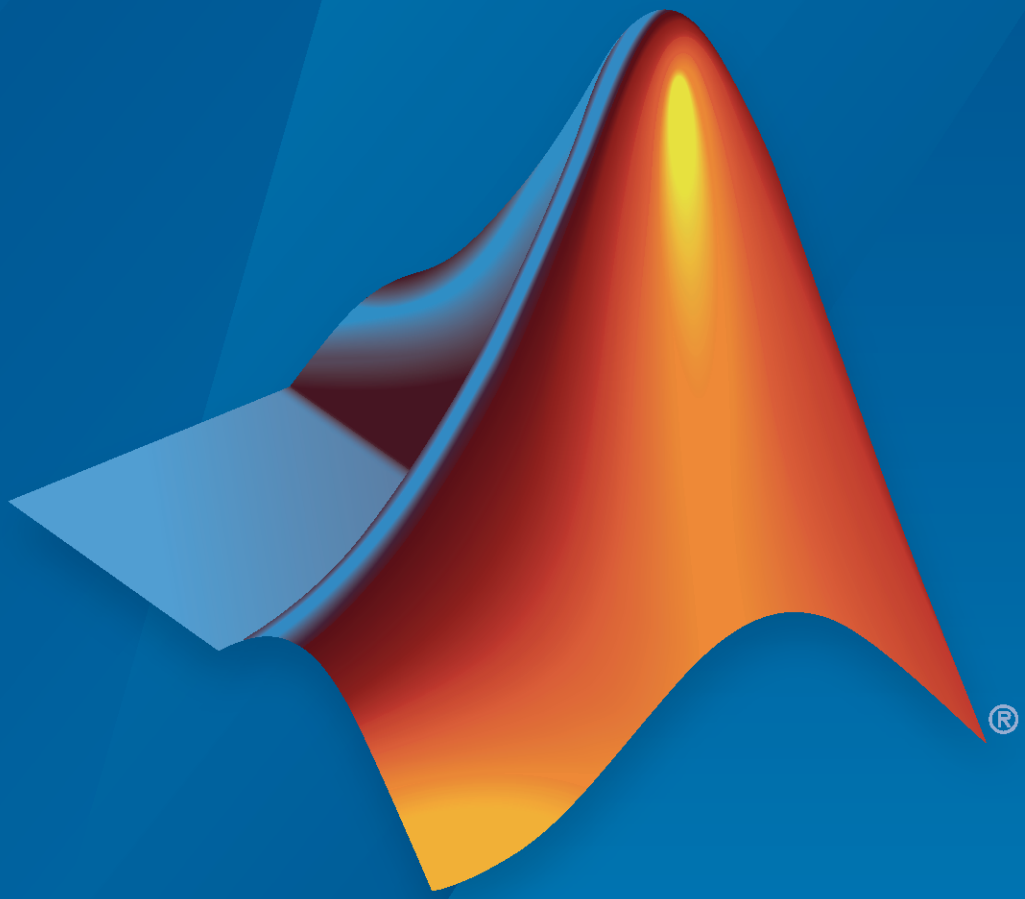
DSP HDL Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2023a

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

DSP HDL Toolbox™ User's Guide

Revision History

March 2022	Online only	New for Version 1.0 (Release 2022a)
September 2022	Online only	Revised for Version 1.1 (Release 2022b)
March 2023	Online only	Revised for Version 1.2 (Release 2023a)

Contents

1 Featured Examples

2 HDL Optimized System Design

3 Block Reference Page Examples

4 HDL Code Generation and Deployment

5 Radar Application Examples

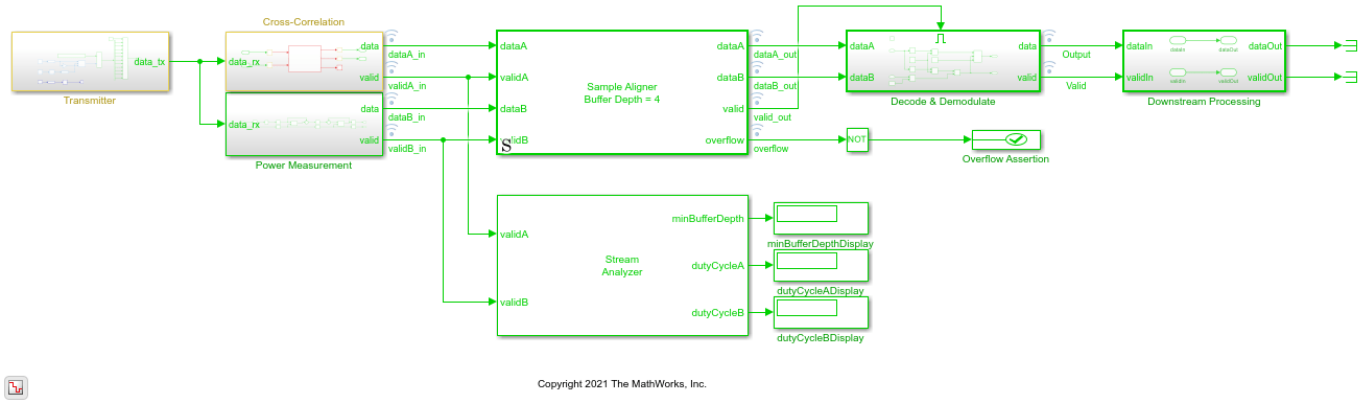
Featured Examples

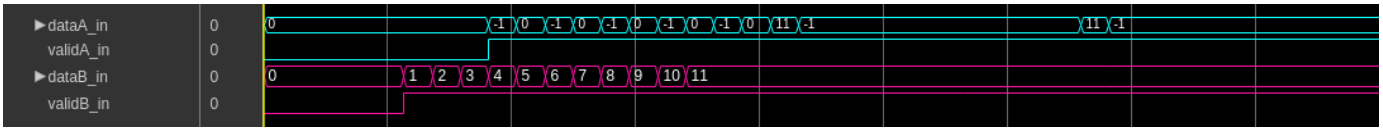
Align Parallel Data Streams

This example shows how to align two commonly sourced data streams with different upstream operation latencies using FIFO-based buffering.

In this example we use the SampleAligner to align two data streams relative to their valid signals. The SampleAligner subsystem uses FIFOs with synchronized read operations. This implementation works only if the input streams have the same number of valid samples in a time period.

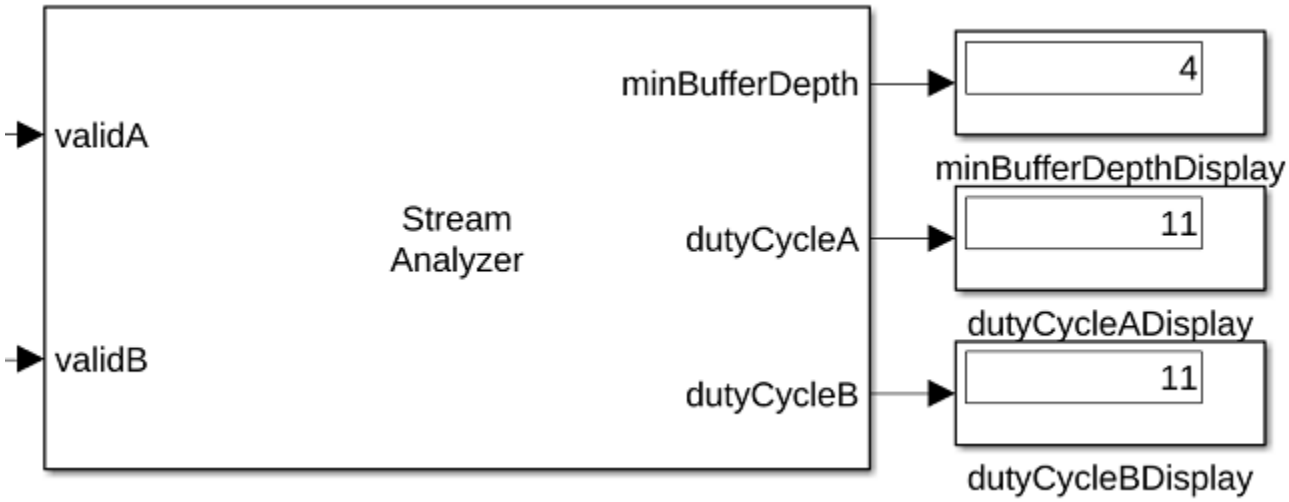
The model generates a random Barker-encoded BPSK input data stream that simulates data from a transmitter. The implemented Barker decoder requires computing the cross-correlation and signal power, and then using those two signals to decode the transmitted samples. Two parallel subsystems perform cross-correlation and measure the signal power. These operations have different latencies, so to recombine the two data streams, they must be re-aligned. The cross-correlation and power measurement operations also do not change the rate or duty cycle of the shared input stream, so they are suitable for a FIFO implementation.





Stream Analyzer

To align the data streams the required buffer depth must be determined and the duty cycle of the data streams be verified as being equal. This is achieved using the StreamAnalyzer which accepts the valid signals of two data streams and determines these required parameters.

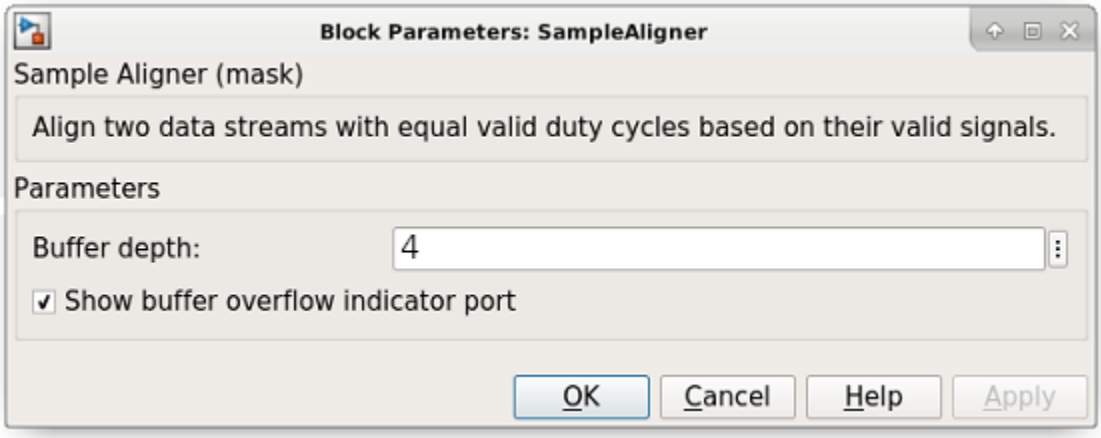


- The **minBufferDepth** output determines the minimum number of samples that must be buffered to synchronize the data streams. This value is determined by taking the maximum modulus value of a counter which is incremented on a valid from stream A and decremented on a valid from stream B.
- The **dutyCycleA** and **dutyCycleB** outputs provide the number of valid samples per second. This calculation starts counting valid samples when it receives the first valid sample on each input stream. The StreamAnalyzer's block sample time is used as the signal period when determining the **dutyCycle**. The duty cycle of both data streams must match to ensure that the internal buffer of the SampleAligner do not overflow. The length of the simulation affects the results of each channel's duty cycle. Longer simulations give more accurate representations of the duty cycles.

The StreamAnalyzer is for use in simulation only and does not support HDL code generation.

Sample Aligner

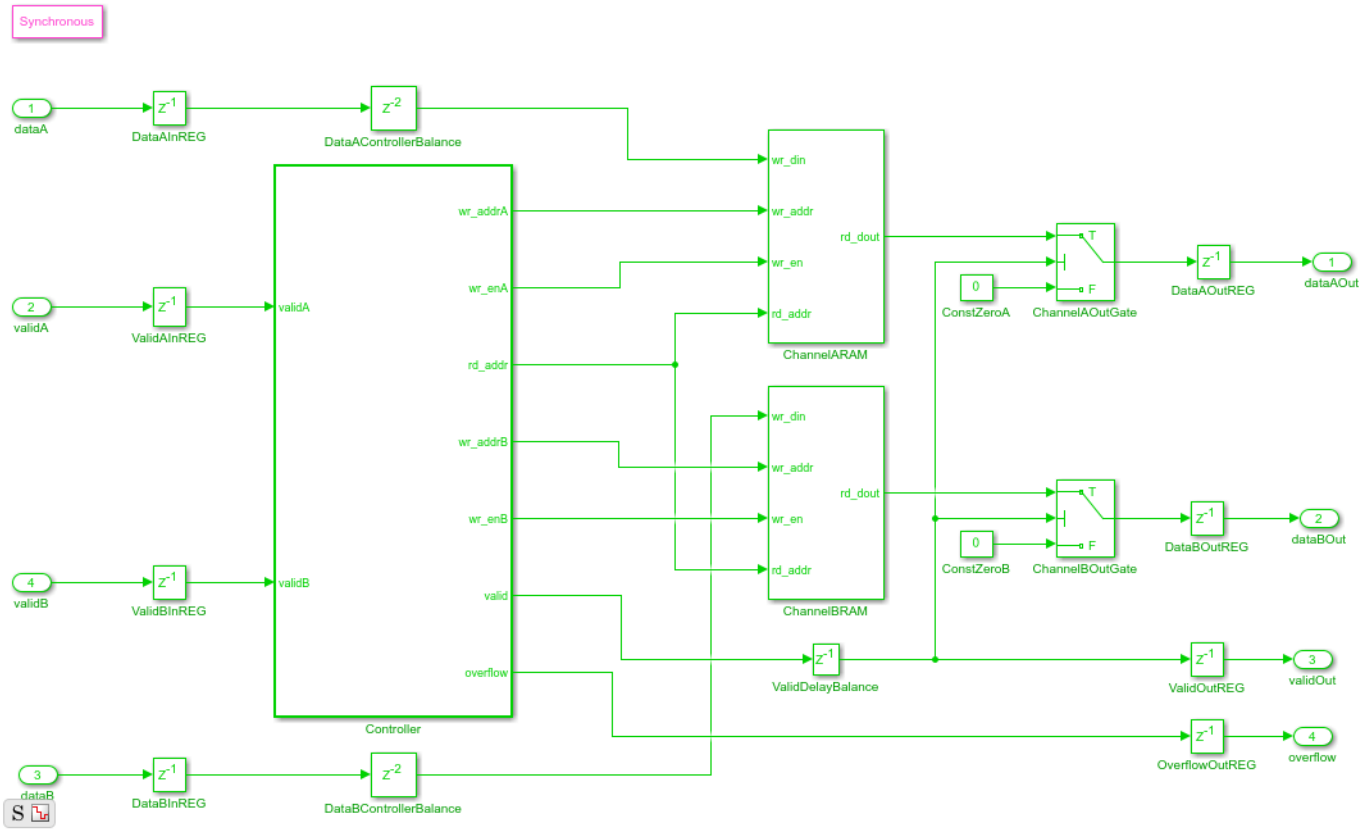
The SampleAligner masked subsystem aligns the samples of two data streams based on their respective valid signals. The subsystem buffers incoming data from either stream until there is at least one valid sample stored from both streams, and then reads both buffers to return a sample for each data stream. The SampleAligner requires that the valid duty cycle of the input data streams match to prevent internal buffer overflows.



The buffer depth parameter sets the internal sample buffer depths. This parameter can be determined from analysis of the latencies for each data stream or from the use of the StreamAnalyzer. The implemented buffer depth is equal to `2^nextpow2(bufferDepth + 5)`. The extra 5 samples accounts for internal control logic latency when operating with continuous valid data streams.

The SampleAligner can be configured to provide an overflow indicator output port which will return a true Boolean value if either of the buffers experience an overflow. In the event of an overflow, no new data is written to the full buffer. This condition leads to loss of data integrity after the buffered samples have been output.

The SampleAligner architecture supports high clock rate applications with its frequency limit imposed by the inferred RAM type. This pipelining means the SampleAligner has a write-to-read latency of 7 clock cycles and a 4 clock cycle latency for the output of an overflow indicator. The input data types for each separate stream does not have to match. Both input streams must be scalar values.



The SampleAligner Controller consists of three parallel operations which handle write and read operations. Pseudo code for these operations are shown below, with more in-depth state diagrams found at the end of example.

```
if (valid_A)
    if (BufferNotFull_A)
        write_A();
    else
        overflow();
    end
end
```

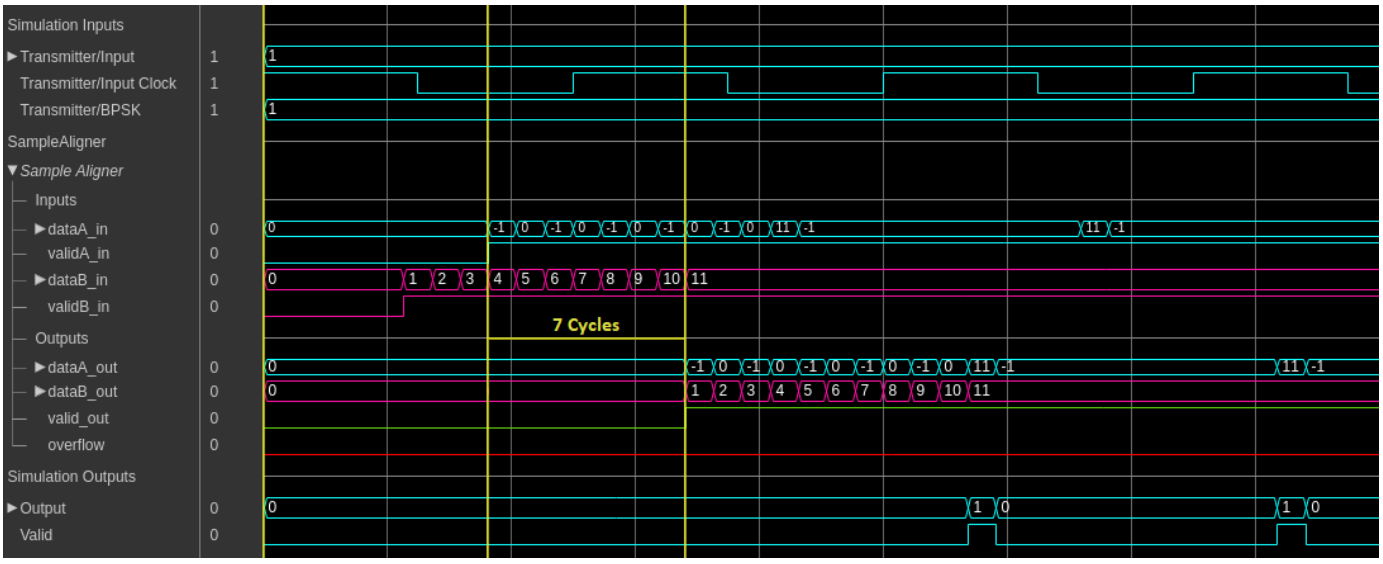
```
if (valid_B)
    if (BufferNotFull_B)
        write_B();
    else
        overflow();
    end
end

if (BufferNotEmpty_A) && (BufferNotEmpty_B)
    read();
end
```

Considerations for Stream Aligning

Performing alignment of parallel data streams is highly specific to the intended use case. The processing applied to both streams prior to the desired alignment stage as well as the downstream processing must be considered to define the alignment constraints. An equal valid duty cycle across both data streams allows for simple FIFO-based buffering. A difference in the valid duty cycles needs to be carefully handled based on the specific use case and may not be possible. Any rate changes across the data streams and the impact of filter architectures on the valid signal must be accounted for. Two data streams can have an equal duty cycle yet different pacing. One could be regular and the other could be a dense bursty stream of data. Bursty data would require a deeper buffer to ensure proper alignment. The present control signals must also be considered, with the introduction of start and end signals imposing additional alignment requirements, deeper buffers, and extended control logic.

Simulation Results

The resulting simulation waveforms can be observed using the Logic Analyzer and show the SampleAligner ensure that both data streams are synchronously output.



HDL Implementation Results

The generated HDL code for the SampleAligner subsystem was synthesized for a Xilinx™ Zynq-7000 ZC706 board and met timing constraints of 649 MHz. The required resources are shown in the table.

```
T =

  3x2 table

    Resource     Usage
    _________    _____

    LUT           39
    LUTRAM        8
    Flip Flop     99
```

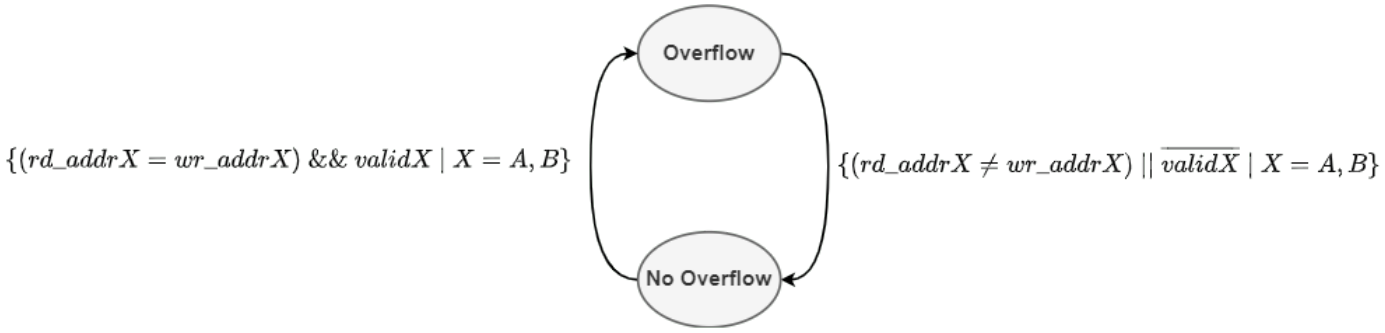
To check and generate the HDL code referenced in this example, you must have the HDL Coder™ product. To generate the HDL code, use this command.

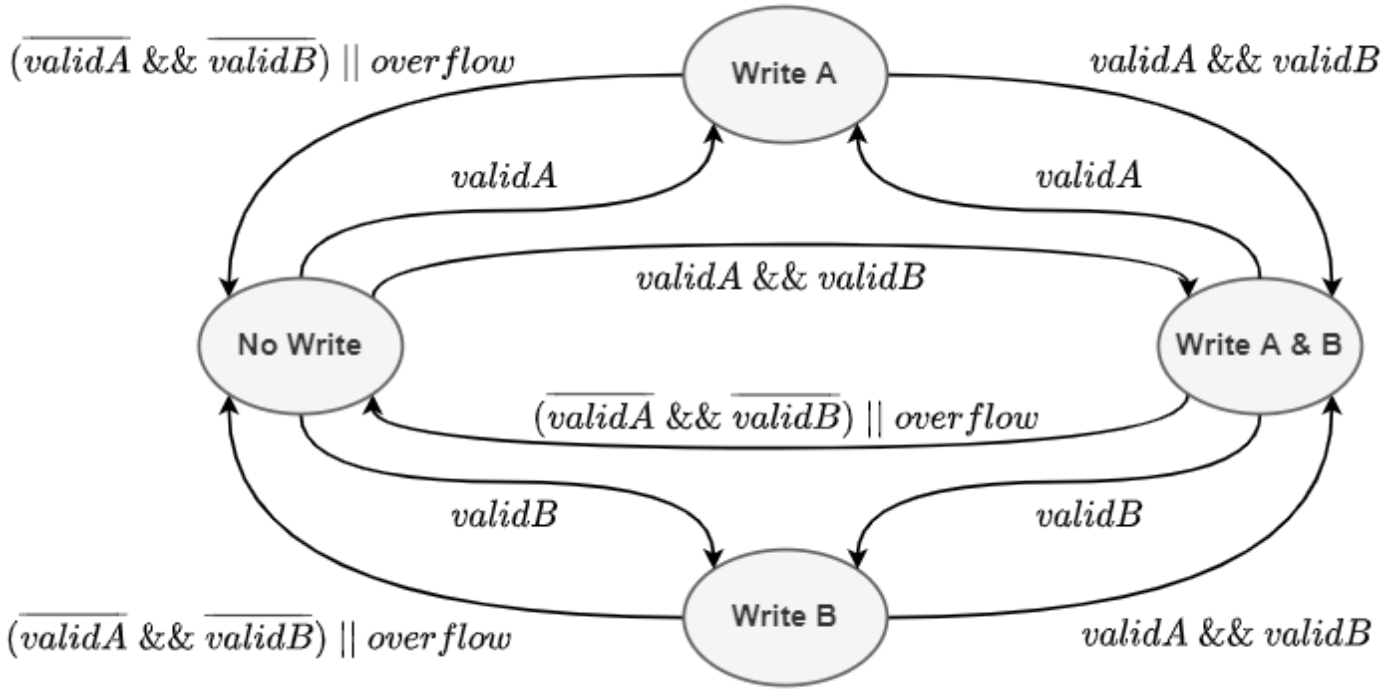
```
makehdl('SampleAlignment/StreamAligner')
```

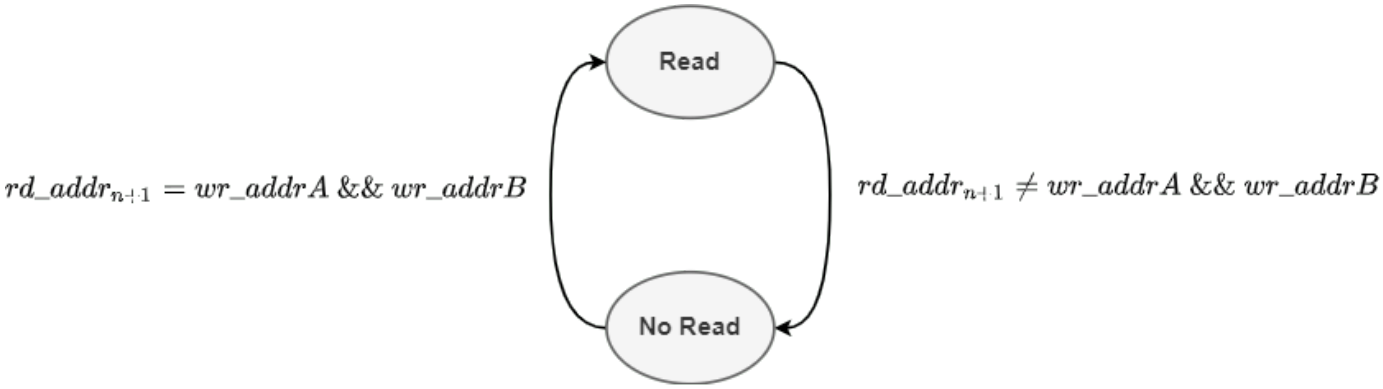
In addition to the HDL generated for this example, the SampleAligner met timing constraints on a Xilinx™ Zynq-7000 ZC706 of 649 MHz and 457 MHz for buffer depths which inferred DRAM and BRAM respectively when operating with two 16-bit data streams. The inferred RAM type is dependent on the implemented buffer size. The SampleAligner controller can see its pipelining reduced or removed to reduce the write-to-read latency if operating at lower frequencies.

Sample Aligner Controller Architecture

The following state diagrams provide a more in-depth description of the SampleAligner's controller operation.







See Also

More About

- “Hardware Control Signals” on page 2-12

HDL Implementation of LMS Filter

This example shows how to implement a fully serial and transpose delayed least mean square (LMS) adaptive filters. The fully serial LMS filter uses standard LMS adaptive filter algorithm and the transpose delayed LMS filter uses delayed LMS filter algorithm with a transpose filter architecture. You can use the fully serial LMS filter for low data-rate applications, such as audio and speech, and the transpose delayed LMS filter for high throughput applications in wireless communication. For high throughput applications, you can also use the LMS Filter block in DSP HDL Toolbox™. The Simulink® model in this example is optimized for HDL code generation and hardware implementation.

The fully serial and transpose delayed LMS filters minimize the error between a desired signal and an observed signal by adjusting the filter weights using mean square error (MSE) criteria. The example model accepts observed, desired, and step-size signals as input and computes the filtered signal, filter error, and filter weights or coefficients. You can use LMS filters for system identification, inverse system identification, noise cancelation, and prediction.

Fully Serial LMS Filter

The block diagram in this section explains the implementation of a fully serial finite impulse response (FIR) filter architecture along with the LMS algorithm filter weights. In this block diagram, the buffered input samples select the data for each tap serially and multiplies with the corresponding coefficients. This architecture reduces the hardware resource utilization, which generates less number of multipliers with additional logic elements such as multiplexers and registers, when compared to fully parallel implementation of LMS filter. Implementation of this architecture requires a clock rate that is faster than the input sample, which depends on the filter length and the number of delays in the feedback path. For example, for a filter length of 16, the block accepts valid input samples for every 35 clock cycles, which is the filter length plus 19 clock cycles.

This section explains the LMS equations corresponding to this architecture. The observed signal $x\left(n\right)$ is passed as an input to the filter and buffered for every filter length of samples. This buffered data $X\left(n\right)$ is multiplied serially with the corresponding filter coefficients W_n and accumulated in a register to get the filtered output $y\left(n\right)$ at timestep n.

$$y\left(n\right) = W_n^T X\left(n\right)$$

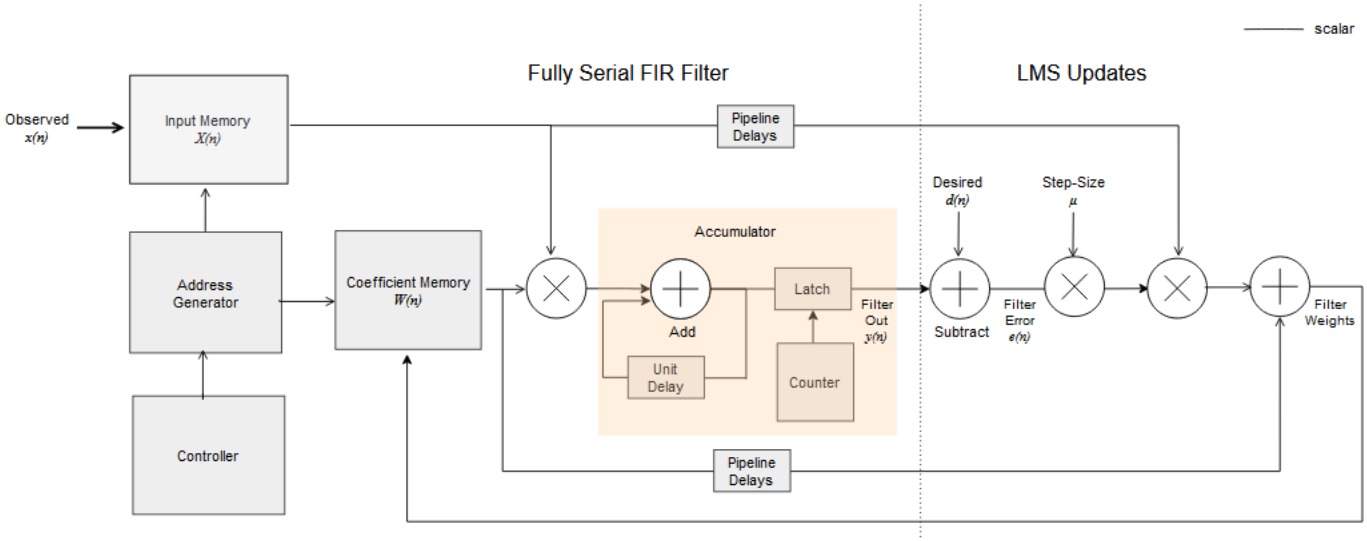
To get the filter error output at timestep n, subtract the filtered output $y\left(n\right)$ from the desired signal $d\left(n\right)$.

$$e\left(n\right) = d\left(n\right) - y\left(n\right)$$

To update the LMS filter weights W_n for the next timestep $n+1$, LMS uses the filter error $e\left(n\right)$ and step size (μ).

$$W_{n+1} = W_n \ + \mu e\left(n\right) X^*\left(n\right)$$

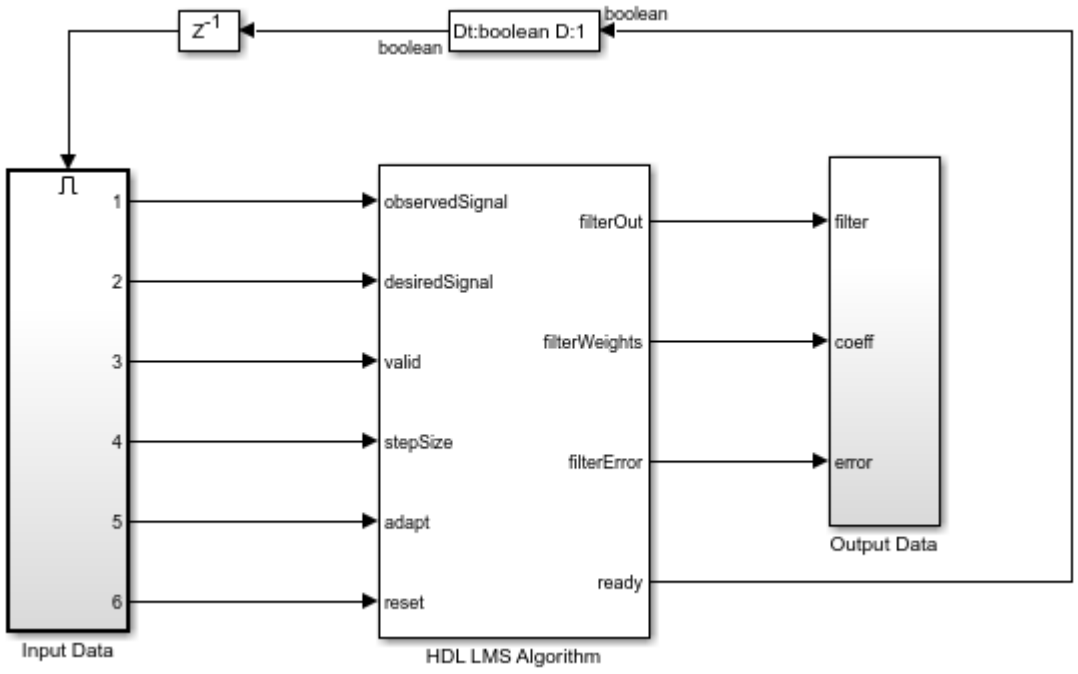
This architecture diagram consists of the fully serial FIR part, $y\left(n\right)$ and the LMS update part, $e\left(n\right)$ and W_{n+1}. The left-side of the diagram shows the fully serial FIR part, and the right-side of the diagram shows the LMS filter weight update part. The input and coefficient memories are used to buffer the observed data and store the filter coefficients.



Model Architecture

Open the example model by entering this command at the MATLAB® command prompt.

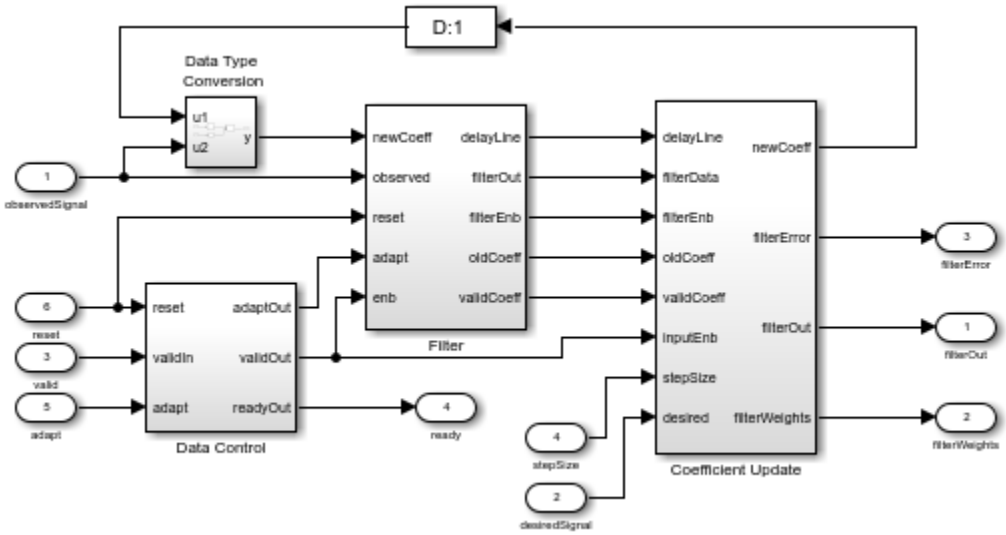
```
modelname = 'HDLFullySerialLMSModel';
open_system(modelname);
```



The `HDL LMS Algorithm` subsystem has Data Control, Filter, and Coefficient Update blocks. The Data Control block generates control signals to control the flow of data. The `ready` signal indicates when to provide the next valid data. The Filter block implements the serial FIR filter and writes the observed input samples to the RAM. The control logic reads the buffered data serially in such a way

that these samples get multiplied with their corresponding filter weights and that the output product is added serially to a register for every filter length of samples to get the filtered output. The Filter block stores the LMS filter weight updates in the RAM and resets the block using a separate reset control logic (when required). The Coefficient Update block implements $e(n)$ and W_{n+1} of the LMS filter. This example starts with a set of zeros as initial values for the estimates of the filter weights.

```
open_system([modelname '/HDL LMS Algorithm'],'force')
```



Transpose Delayed LMS Filter

This section explains the equations corresponding to the delayed LMS algorithm. These equations are approximations to the standard LMS algorithm. The transpose delayed LMS filter uses these equations with transpose filter structure to fit the hardware characteristics. The observed signal $x(n)$ is passed as an input to the filter. The number of delays (*D*) in the feedback path are independent of filter length because the filter part is implemented using a transposed filter architecture, which improves the throughput by minimizing the pipelined delays in the feedback path of LMS filter.

The filtered output $y(n)$ at timestep n is given by the equation.

$$y(n) = W_{n-D}^{T} X(n)$$

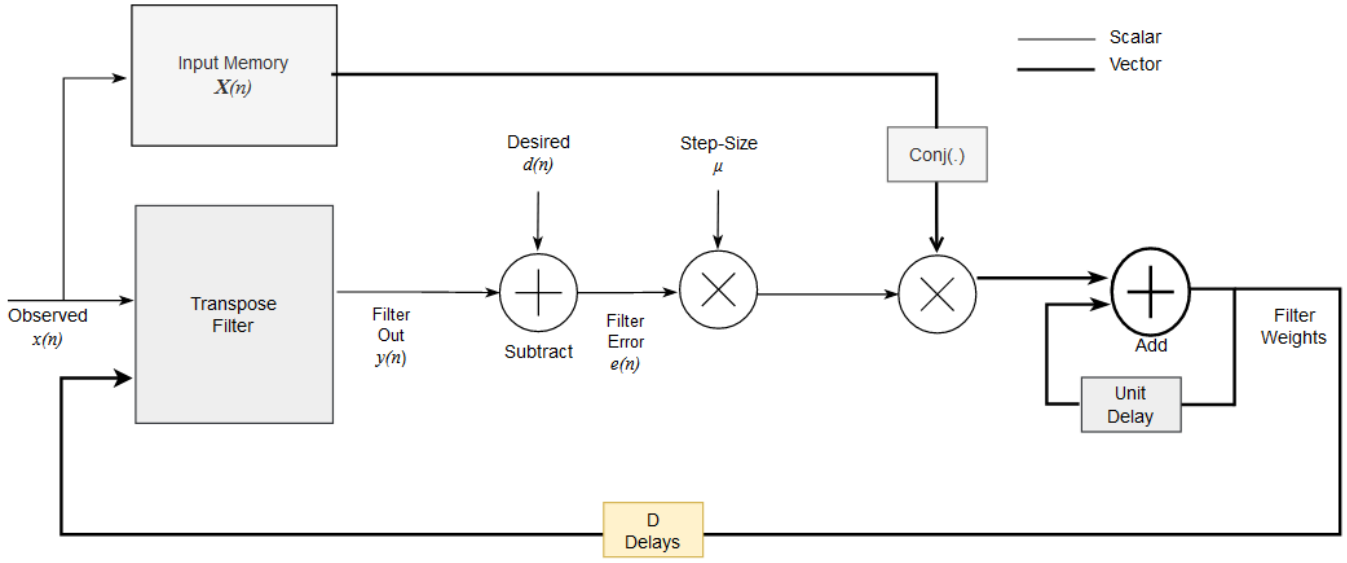
To get the filter error output at timestep n, subtract the filtered output $y(n)$ from the desired signal $d(n)$.

$$e(n) = d(n) - y(n)$$

To update the filter weights W_n for the next timestep $n+1$, LMS filter uses the filter error $e(n)$ and step size (μ).

$$W_{n+1} = W_n + \mu e(n) X^*(n)$$

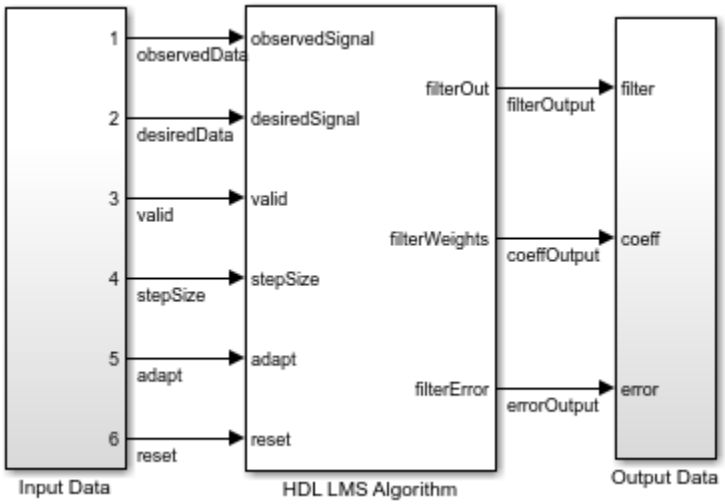
This architecture diagram shows the transpose delayed LMS filter.



Model Architecture

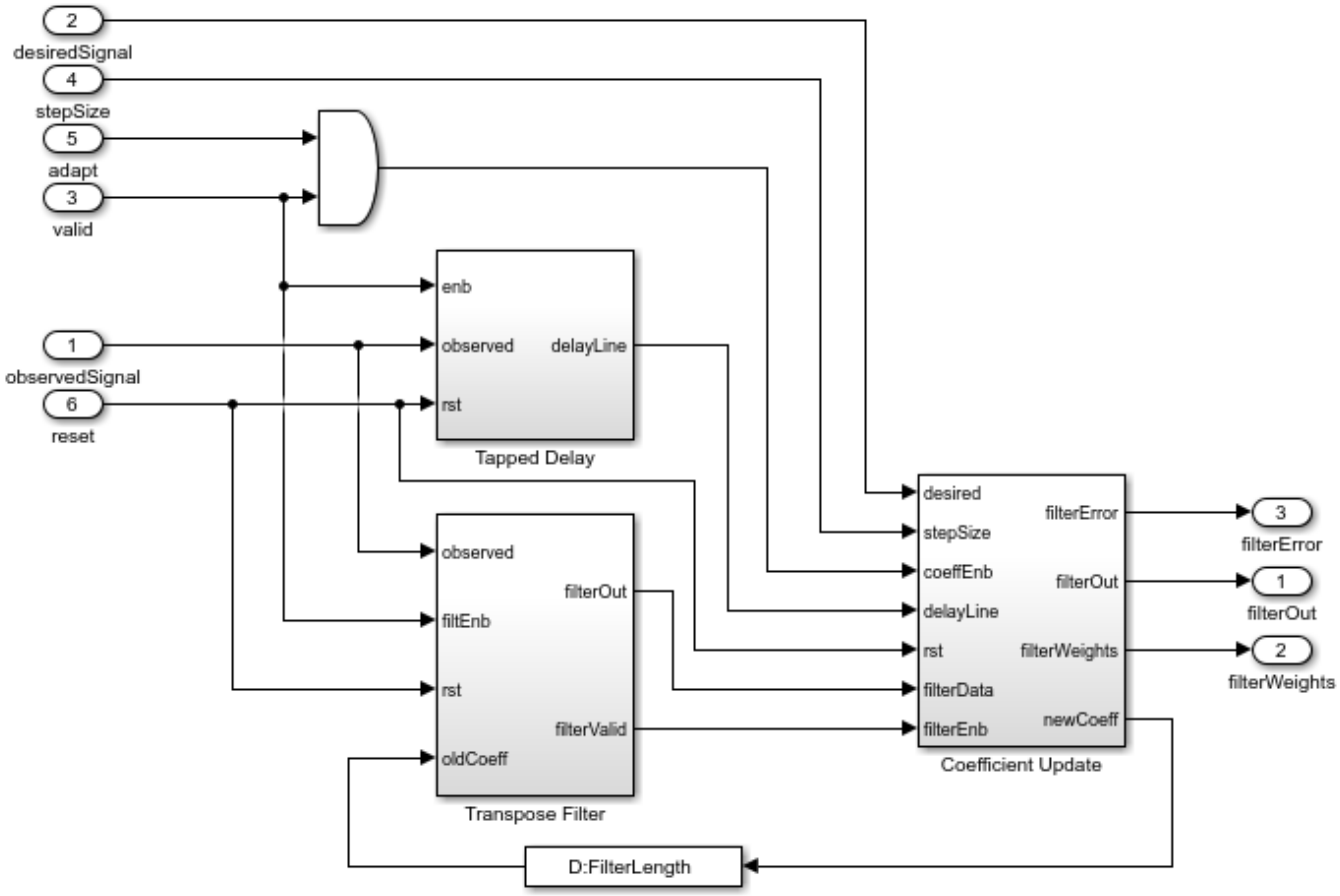
Open the example model by entering this command at the MATLAB® command prompt.

```
modelname = 'HDLDelayedLMSModel';
open_system(modelname);
```



The `HDL LMS Algorithm` subsystem has Transpose Filter, Tapped Delay, and Coefficient Update blocks. The Transpose filter implements the transpose filter structure and outputs $y(n)$. The Tapped delay block buffers the input data $X(n)$. The Coefficient Update block implements $e(n)$ and W_{n+1} of the LMS filter. This example starts with a set of zeros as initial values for the estimates of the filter weights.

```
open_system([modelname '/HDL LMS Algorithm'],'force')
```



Port and Parameter Descriptions

This section explains the port and parameter descriptions for the `HDLFullySerialLMSModel.slx` and `HDLDelayLMSFilterModel.slx` models. The input *observedSignal* and *desiredSignal* signals must be scalars and of the same data type. To support HDL code generation, inputs must be scalar fixed-point data types.

Input Ports

- **observedSignal** — Input observed signal $x(n)$ to the LMS filter, specified as a real- or complex-valued scalar. The block supports `double`, `single`, and signed fixed-point data types.
- **desiredSignal** — Input desired signal $d(n)$ to the LMS filter, specified as a real- or complex-valued scalar. The block supports `double`, `single`, and signed fixed-point data types.
- **stepSize** — Input step size μ to the block, specified as `double`, `single`, or unsigned fixed-point data types. This value determines the convergence rate and stability of the filter and must be less than 1/(power of input signal x filter length) for filter stability. For more information, see [1].
- **reset** — Input reset signal, specified as a Boolean scalar. When **reset** is `true`, the block stops the current calculation and clears the internal states.
- **valid** — Input valid signal, specified as a Boolean scalar to validate the input data.

- **adapt** — Input adapt signal, specified as a Boolean scalar. When this signal is `1` (high), the LMS filter updates its filter weights. When this signal is `0` (low), the filter weights remain the same as what they were previously. This signal is sampled only when the valid signal is `1` (high).

Output Ports

- **fiteredOut** — Filtered output $y(n)$, returned as a scalar. The LMS filter converges this signal with desired signal $d(n)$ by adjusting filter weights.
- **filterError** — Filter error $e(n)$, returned as a scalar. This value is the difference between filtered output $y(n)$ and desired signal $d(n)$.
- **filterWeights** — Filter weights W_n, returned as a scalar for the `HDLFullySerialLMSModel.slx` model and a vector of the same size as filter length for the `HDLDelayLMSFilterModel.slx` model.
- **ready** — Ready signal, returned as a Boolean scalar. This port is applicable for the `HDLFullySerialLMSModel.slx` model. This signal indicates when to provide the next data.

This LMS filter model accepts the filter length as a parameter.

For a fixed-point implementation, internal word length calculations are based on the assumption that the input is a unit power signal. The recommended data type for the inputs and the observed and desired signals is `fixdt(1,16,14)`. The recommended data type for the step size is `fixdt(0,18,18)`.

System Identification of FIR Filter

System identification is a process to find unknown system filter coefficients using an adaptive filter. This section shows how to find unknown system coefficients.

Create a filter object for an unknown system.

```
filterLength = 16;
filterObj = dsp.FIRFilter;
filterObj.Numerator = fircband(filterLength-1,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],...
    {'w' 'c'});
filterObj %#ok
```

```
filterObj =

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: [-0.0346 0.0069 0.0766 0.0156 -0.0785 -0.0448 ... ]
    InitialConditions: 0

  Use get to show all properties
```

Specify a model.

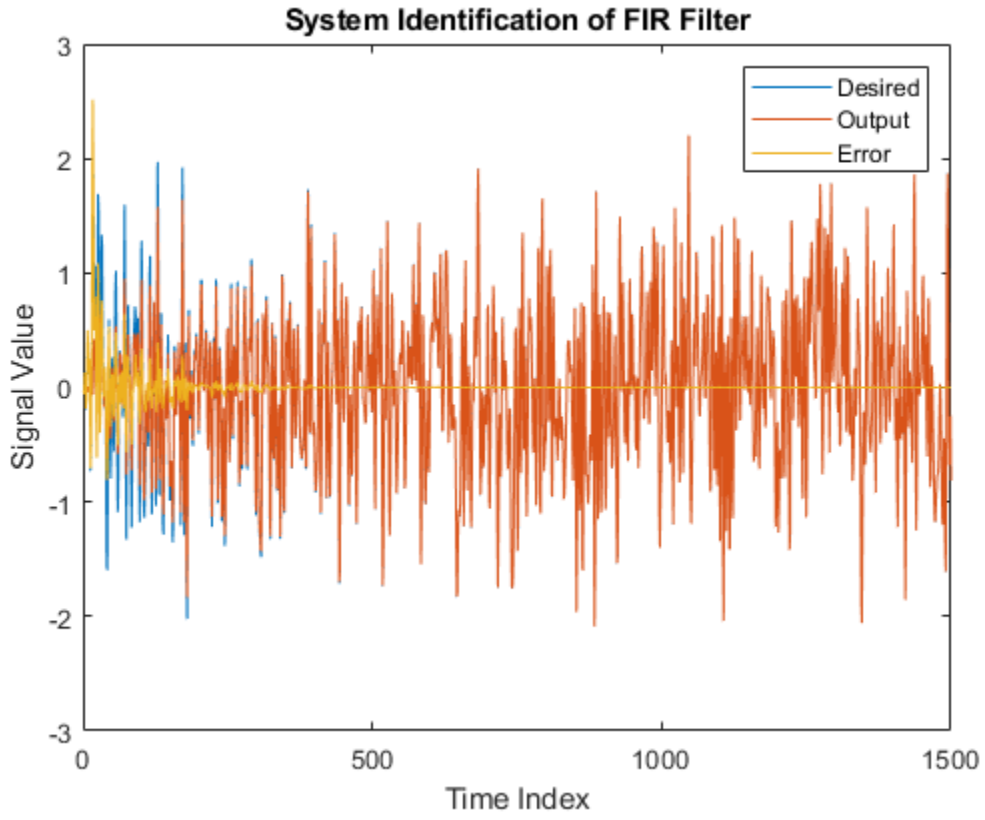
```
% For the fully serial LMS filter, use 'HDLFullySerialLMSModel', for
% delayed LMS filter use 'HDLDelayedLMSModel'
modelname = 'HDLFullySerialLMSModel';
```

Pass the unit power signal, `observedSignal` to the FIR filter. The `desiredSignal` is the sum of the output of the unknown system (FIR filter) and the additive noise signal `noise`. The LMS filter adapts its coefficients until its transfer function matches the transfer function of the unknown system.

```
observedSignal = randn(1500,1);
noise = 0.001*randn(1500,1);
desiredSignal = filterObj(observedSignal) + noise;
stepSize = 0.01;

% Create input and control signals
adaptIn = true(length(observedSignal),1);
resetIn = false(length(observedSignal),1);
validIn = true(length(observedSignal),1);

if strcmpi(modelname,'HDLFullySerialLMSModel')
    simtime = length(observedSignal)*(filterLength+19) + 10;
else
    simtime = length(observedSignal) + 100;
end
out = sim(modelname);

% Capture valid data outputs
actFilterOut = out.filterOut;
actCoeffOut = out.coeffOut;
actErrorOut = out.errorOut;
```

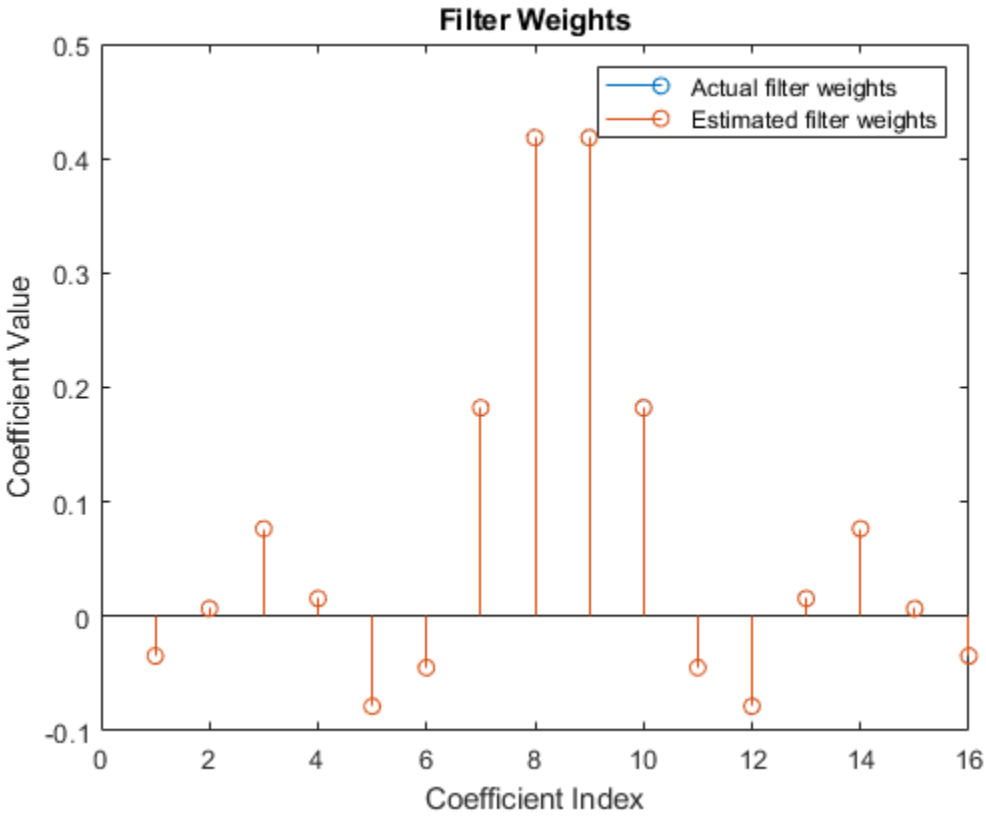
Plot the outputs.

```
figure;
plot(1:1:length(observedSignal),double([desiredSignal,actFilterOut,actErrorOut]))
title('System Identification of FIR Filter')
legend('Desired','Output','Error')
xlabel('Time Index')
ylabel('Signal Value')
```



Because the output weights for the `HDLFullySerialLMSModel.slx` model are serial, you must convert the weights to a vector to represent the coefficients of the LMS filter, which is adapted to resemble the unknown system (FIR filter). To confirm the convergence, compare the numerator of the FIR filter with the estimated weights of the LMS filter.

```
figure;
% Compare actual filter weights with estimated filter weights
if strcmpi(modelname,'HDLFullySerialLMSModel')
    val = (filterLength)*(length(observedSignal)-1);
    stem([(filterObj.Numerator).' double(actCoeffOut((val+1):(val+filterLength)))]);
else
    val = (length(observedSignal)-1);
    stem([(filterObj.Numerator).' double(actCoeffOut((val+1),:)).']);
end
xlabel('Coefficient Index')
ylabel('Coefficient Value')
title('Filter Weights')
legend('Actual filter weights','Estimated filter weights', ...
    'Location','NorthEast')
```



Verification and Results

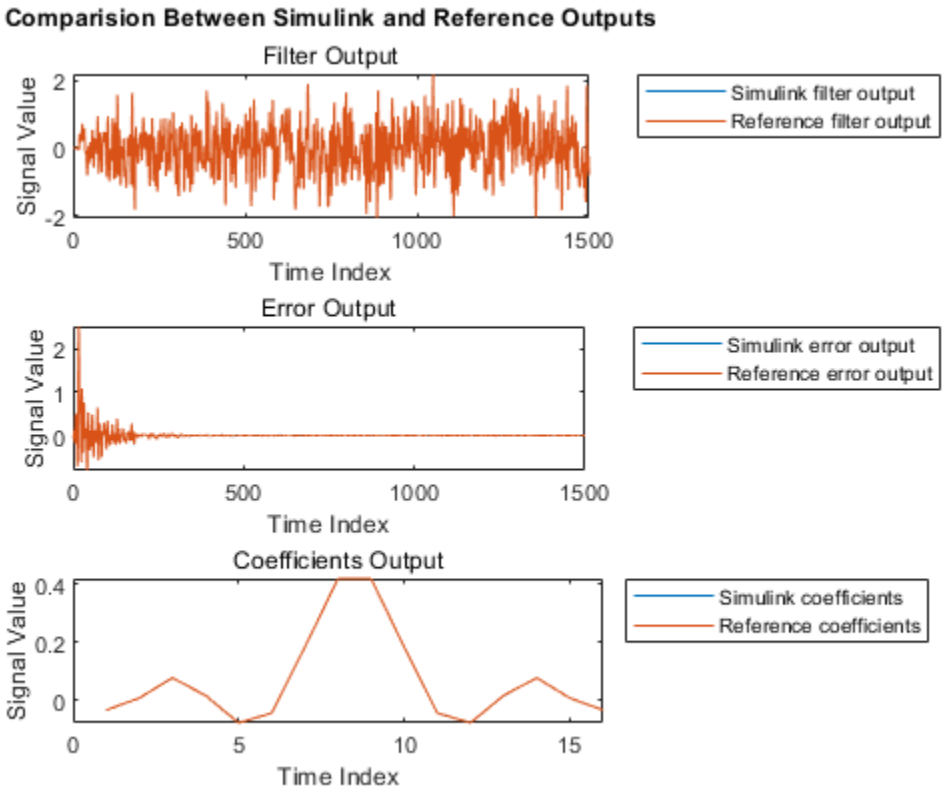
For verification, compare the outputs from this example with the outputs from the `dsp.LMSFilter` function.

```
lms = dsp.LMSFilter(filterLength,'StepSize',stepSize);
[y,e,w] = lms(observedSignal,desiredSignal);
```

Plot the Simulink and reference outputs and verify the difference between them.

```
figure;
% Compare Simulink and reference filtered out
subplot(3,1,1)
plot(1:length(observedSignal),[actFilterOut,y])
title('Comparision Between Simulink and Reference Outputs')
subtitle('Filter Output')
legend('Simulink filter output','Reference filter output','Location','bestoutside')
xlabel('Time Index')
ylabel('Signal Value')

% Compare Simulink and reference filter error out
subplot(3,1,2);
plot(1:length(observedSignal),[actErrorOut,e])
subtitle('Error Output')
legend('Simulink error output','Reference error output','Location','bestoutside')
xlabel('Time Index')
ylabel('Signal Value')
```

```
% Compare Simulink and reference filtered coefficients
subplot(3,1,3);
if strcmpi(modelname,'HDLFullySerialLMSModel')
    plot(1:filterLength,[double(actCoeffOut((val+1):(val+filterLength))),w])
else
    plot(1:filterLength,[double(actCoeffOut((val+1),:)).',w])
end
subtitle('Coefficients Output')
legend('Simulink coefficients','Reference coefficients','Location','bestoutside')
xlabel('Time Index')
ylabel('Signal Value')
```

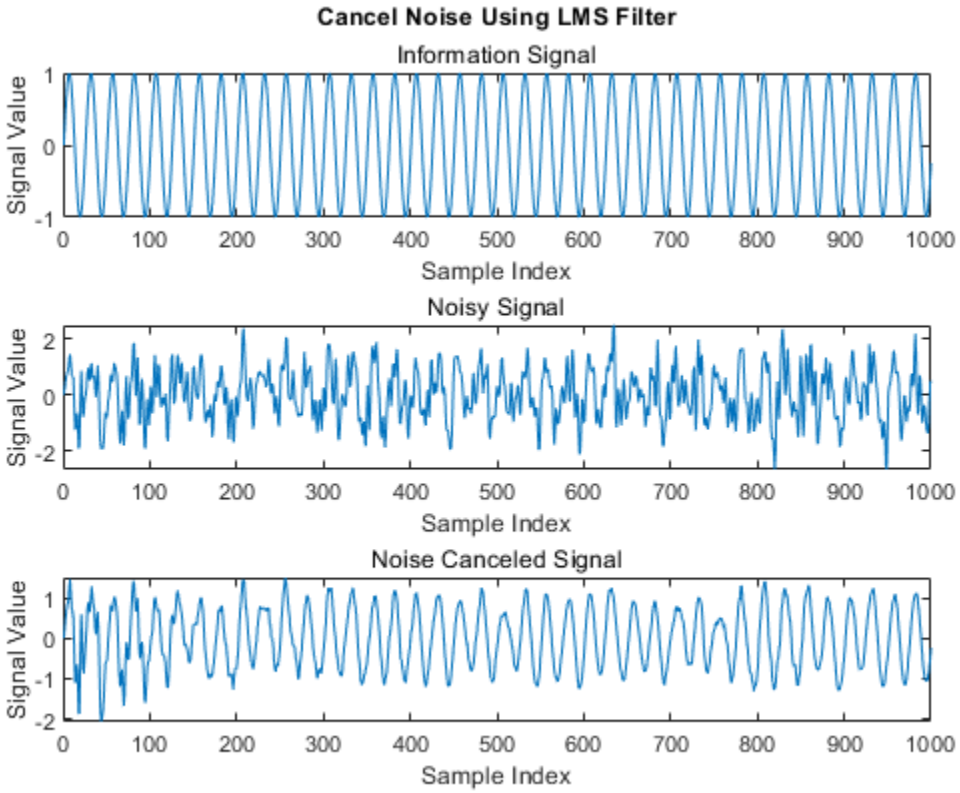


Noise Cancellation Using LMS Filter

Run the `HDLLMSNoiseRemoval.m` script to remove noise from the information signal. The information signal is a pure sine wave corrupted with additive noise, which is considered as a desired signal. The observed signal is correlated with noise data and fed to the LMS filter. When the filter converges, it provides the noise-canceled information signal from the **filterError** port. This plot shows the information signal, noisy signal, and the output noise corrected signal from the fully serial LMS filter in the `HDLFullySerialLMSModel.slx` model. You can generate similar plot for the transpose delayed LMS filter by specifying the model `HDLDelayedLMSModel.slx` in the `HDLLMSNoiseRemoval.m` script.

>> HDLLMSNoiseRemoval

`HDLLMSNoiseRemoval`

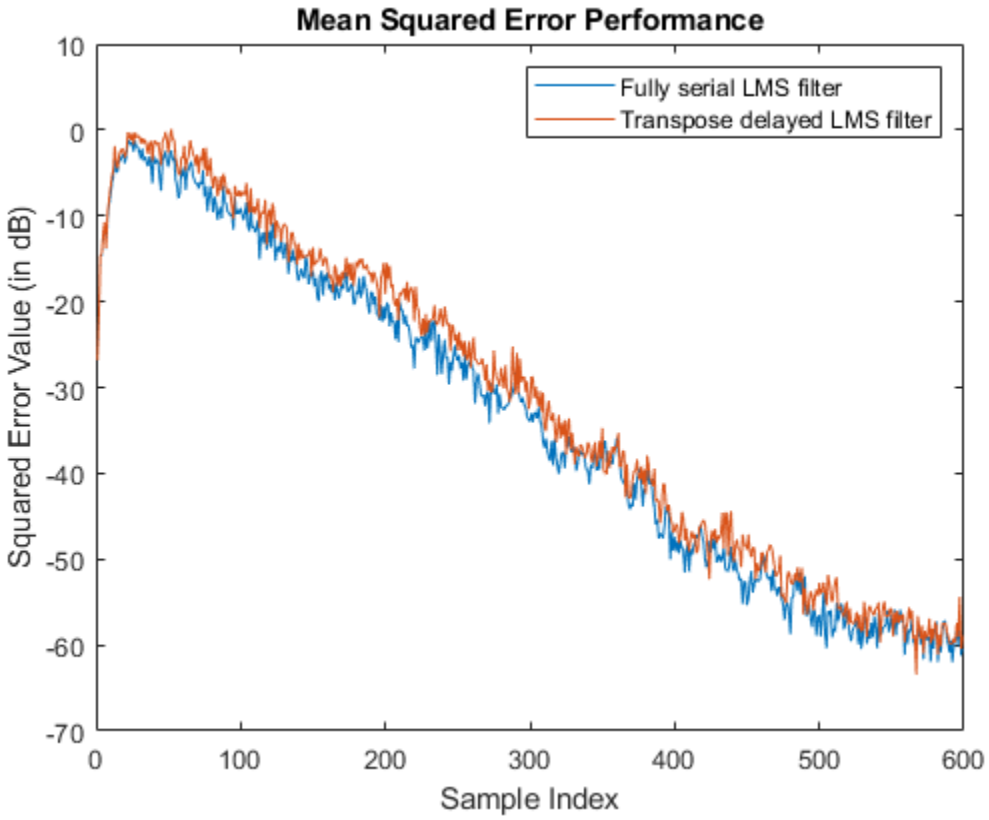


Mean Square Performance

Run the `HDLLMSConvergencePlot.m` script to plot the mean square error (MSE) performance of the LMS algorithm. The MSE measures the average squares of the errors between the desired signal and the primary input signal to the adaptive filter. This plot is the convergence plot for the fully serial LMS filter in the `HDLFullySerialLMSModel.slx` model and the transpose delayed LMS filter in the `HDLDelayedLMSModel.slx` model. The convergence plot of the transpose delayed LMS filter for the small step sizes closely matches with the fully serial LMS filter.

>> HDLLMSConvergencePlot

```
HDLLMSConvergencePlot
```



Throughput

The throughput for the fully serial LMS filter in the `HDLFullySerialLMSModel.slx` model is calculated as (*fMax* / (*FL* + *t*)). The throughput for transpose delayed LMS filter in the `HDLDelayedLMSModel.slx` model is equivalent to *fMax* , which is independent of the filter length. In this calculation: *fMax* is the maximum operating frequency, *FL* is the filter length, and *t* is the number of pipeline delays required in feedback path of fully serial LMS implementation. *t* is 19 in this implementation.

HDL Code Generation and Implementation Results

To generate HDL code for this example, you must have HDL Coder™ product. To generate HDL code and HDL test bench, use the `makehdl` and `makehdltb` commands. The block is synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board.

For the fully serial LMS filter in the `HDLFullySerialLMSModel.slx` model, this example achieves a clock frequency of 470 MHz. For the transpose delayed LMS filter in the `HDLDelayedLMSModel.slx` model, this example achieves a clock frequency of 450 MHz. These tables show the post place and route resource utilization results for the real-valued input word length of 16 and step-size word length of 18 for a filter length of 16 for both models. The resource utilization and synthesis frequency vary with the filter length and input word lengths.

Display the resource utilization for the fully serial LMS filter in the `HDLFullySerialLMSModel.slx` model.

```
F = table( ...
    categorical({'Slice Registers'; 'Slice LUT';'DSP'}), ...
```

```
    categorical({'551'; '412'; '3'}), ...
    categorical({'437200'; '218600'; '900'}), ...
    categorical({'0.13'; '0.19'; '0.33'}), ...
    'VariableNames',{'Resources','Utilized','Available','Utilization (%)'});
disp(F)
```

Resources	Utilized	Available	Utilization (%)
Slice Registers	551	437200	0.13
Slice LUT	412	218600	0.19
DSP	3	900	0.33

Display the resource utilization for the transpose delayed LMS filter in `HDLDelayedLMSModel.slx` model.

```
F = table( ...
    categorical({'Slice Registers'; 'Slice LUT';'DSP'}), ...
    categorical({'3395'; '2914'; '33'}), ...
    categorical({'437200'; '218600'; '900'}), ...
    categorical({'0.78'; '1.33'; '3.67'}), ...
    'VariableNames',{'Resources','Utilized','Available','Utilization (%)'});
disp(F)
```

Resources	Utilized	Available	Utilization (%)
Slice Registers	3395	437200	0.78
Slice LUT	2914	218600	1.33
DSP	33	900	3.67

References

[1] Hayes,M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

See Also

Blocks
LMS Filter

Objects
`dsphdl.LMSFilter` | `dsp.LMSFilter`

High Performance DC Blocker for FPGA

This example shows how to create a DC blocking filter for a communications system by using the Biquad Filter block in DSP HDL Toolbox™.

A DC blocking filter is used in many communications system designs to simplify other filtering requirements. A DC blocking filter is a overall high-pass filter with a very low frequency transition frequency that typically achieves the highpass response by subtracting a lowpass version of the input from the input, leaving just the portions of the signal above DC.

This application is a natural fit for infinite impulse-response (IIR) filters since they are able to provide a steep transition bandwidth with fewer resources than a more common finite impulse response (FIR) filter. A drawback of an IIR filter is that stability of the filter is not guaranteed, so generally a second-order section (SOS), also known as a biquad filter, design is used. Biquad filters can guarantee stability even after fixed-point quantization.

This example shows three implementations of a DC blocker algorithm for hardware. The Biquad Filter block in DSP HDL Toolbox provides options to select different SOS filter architectures to achieve different tradeoffs of resources and throughput in your design.

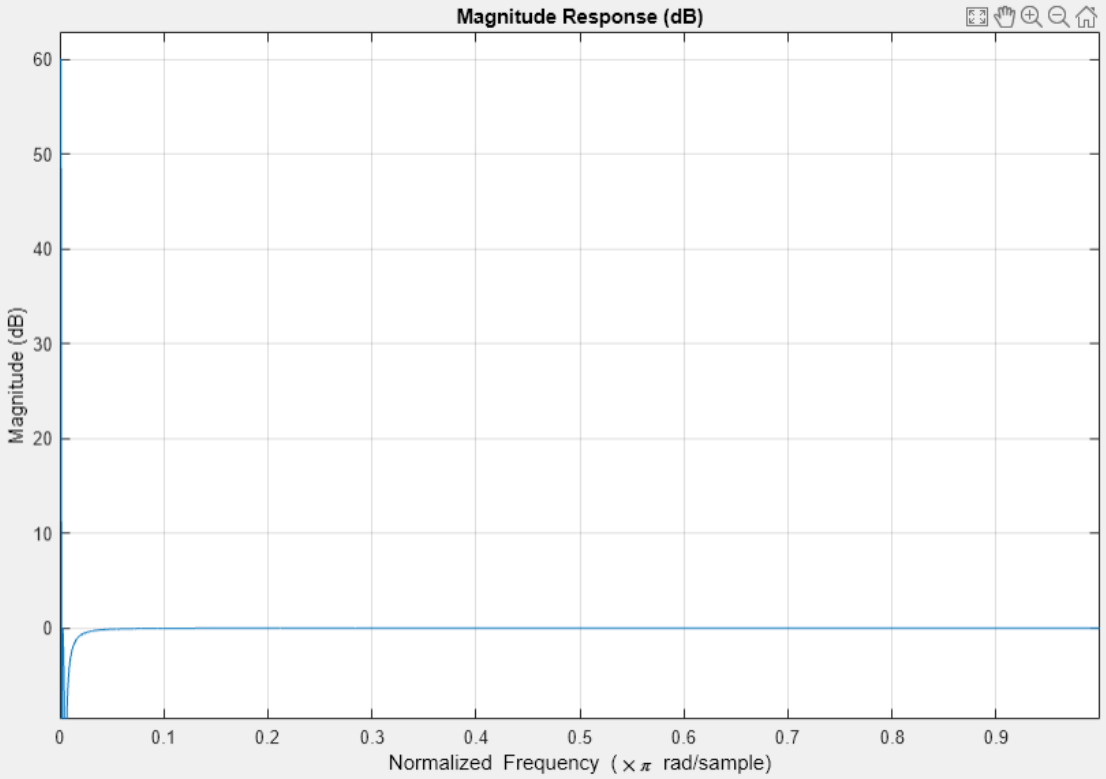
Reference DC Blocker Block

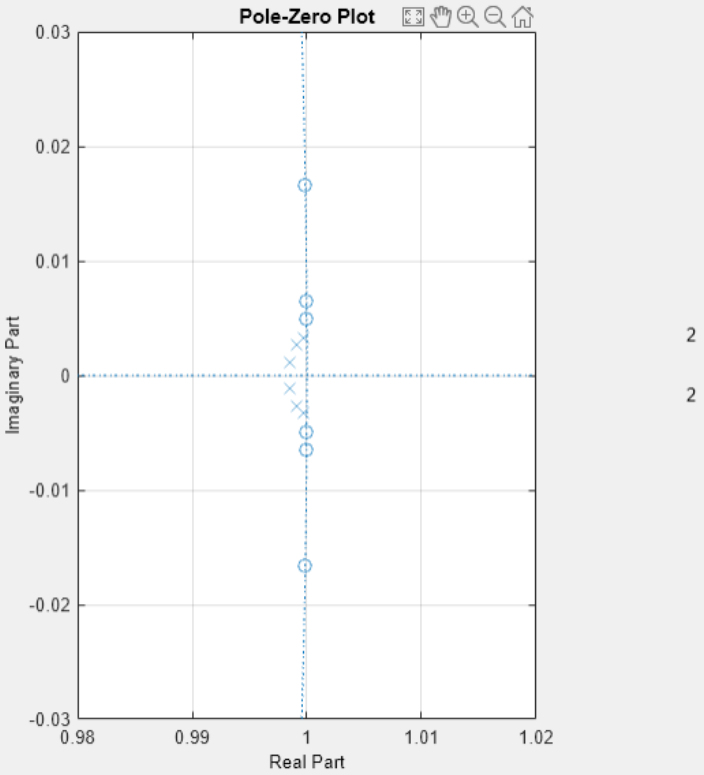
The Communications Toolbox™ DC Blocker Simulink block can provide a behavioral reference for your hardware design. It supports four types of filters: IIR, FIR, cascaded integrator comb, and subtract mean. This example focuses on IIR filters. The block is designed for floating-point operation and has two parameters for an IIR DC blocking filter: the normalized bandwidth of the lowpass stopband and the order of the filter. The block automatically designs an SOS or biquad filter from these parameters with a passband ripple of 0.1 dB and a stopband attenuation of 60 dB.

The block uses an elliptical filter design method, which provides steeper roll-off transition bandwidth characteristics than Butterworth or Chebyshev filter designs and often allows the lowest order for a given frequency response. The downside to this type of filter is that the poles and zeros are often very close to the unit circle and the stability of a quantized filter can sometimes be challenging.

When you design SOS or biquad filters, use a design method that returns zeros, poles, and gain, `(z,p,k)`, rather than the multiplied-out transfer function, `(b,a)`, since the transfer function polynomial can have numerical instability. The DC Blocker block returns zeros, poles, and gain similar to this filter design code. The poles in the pole-zero plot (represented by x values) are very near but still inside the unit circle, making the double-precision filter stable.

```
filterOrder = 6;                          % user set, default value
normalizedBandwidth = 0.001;              % user set, default value
passbandRipple = 0.1;                     % non-settable, in dB
stopbandAtten  = 60;                      % non-settable, in dB
[z,p,k] = ellip(filterOrder,passbandRipple,stopbandAtten,normalizedBandwidth);
[sos,g] = zp2sos(z,p,k);
f1 = fvtool(sos);
f2 = fvtool(sos,'polezero');
zoom(f2,[0.98 1.02 -0.03 0.03]);
```

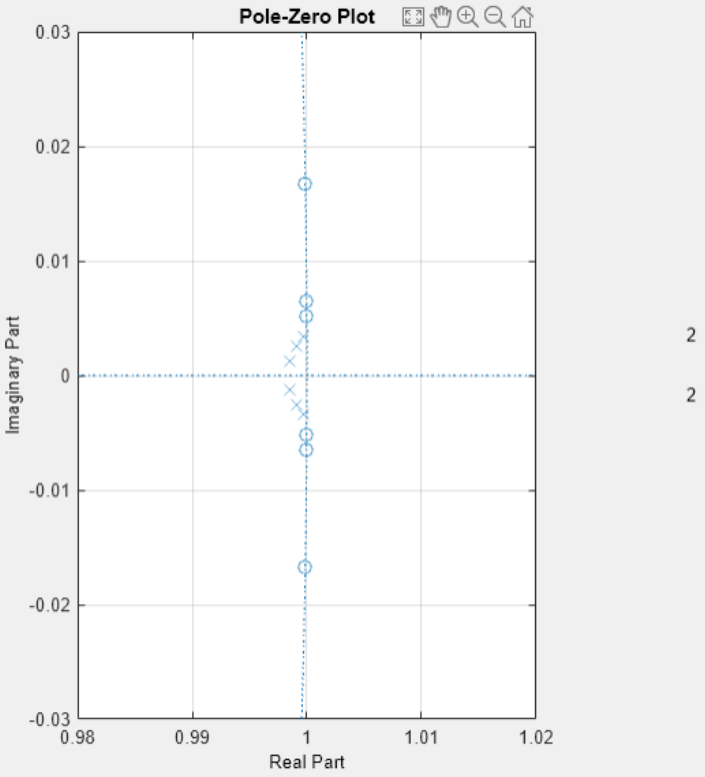

Magnitude Response (dB)
Magnitude (dB)
60
50
40
30
20
10
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Normalized Frequency (×π rad/sample)



Design Filter for Hardware

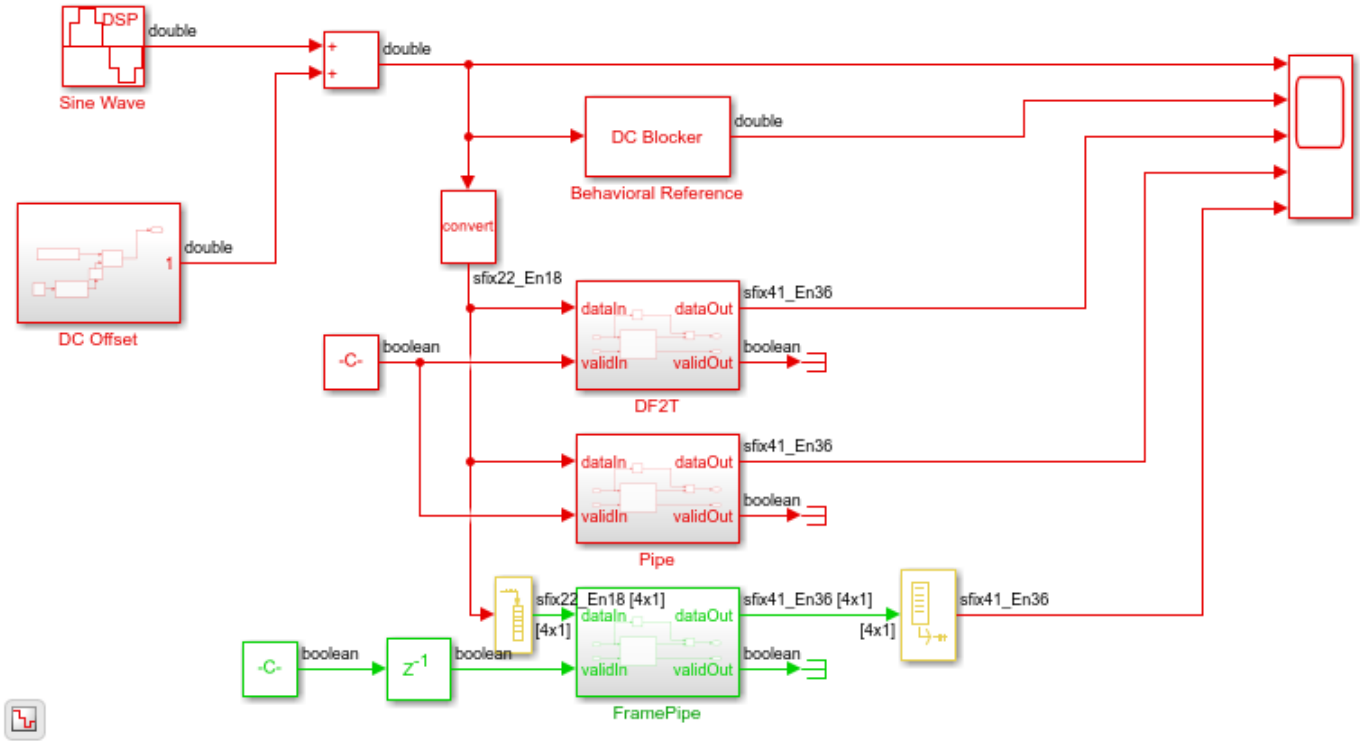
Quantizing the numerator and denominator coefficient independently allows for better results, so this code separates the numerator and denominator before quantization. This design has 12-bit input and targets a typical DSP element in an FPGA, so the example uses 20-bits for numerator and denominator coefficients. The pole-zero plot shows the quantized fixed-point numerator and denominator coefficients after they are converted back to double data type for analysis. You can confirm that the quantized filter is stable by changing the Analysis setting in the Filter Visualization Tool to Filter Info.

```
numerator = sos(:,1:3);
denominator = sos(:,4:6);
scaleValues = g;
scaledDoubleSOS = [double(fi(numerator,1,20)),double(fi(denominator,1,20))];
f3 = fvtool(scaledDoubleSOS,'polezero');
zoom(f3,[0.98 1.02 -0.03 0.03]);
```



The example model shows three different modes of the Biquad Filter block as well as the original behavioral model of the DC Blocker from the Communications Toolbox library.

```
modelname = 'DCBlockerHDL';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
```



The `DF2T` subsystem contains a Biquad Filter block with the **Filter structure** parameter set to `Direct form II transposed`.

The `Pipe` subsystem contains a Biquad Filter block with the **Filter structure** parameter set to `Pipelined feedback form`.

The `FramePipe` subsystem also contains a Biquad Filter block with the **Filter structure** parameter set to `Pipelined feedback form`. This subsystem uses 4-sample vector input to parallelize the filter operation and increase throughput.

In each of the subsystems, the result of the Biquad Filter is subtracted from the input data delayed to match the latency of the filter. All of the subsystems compute the same results, but each filter uses different hardware resources and synthesizes to a different clock rate. The rest of this example compares the resources, clock rate, and throughput of each of the biquad filter structures.

Direct Form II Transposed Filter Structure

To create a unique folder for the generated HDL code for each filter implementation, add the `TargetDirectory` name-value argument to the `makehdl` function call. Use this command to generate HDL code for the direct-form transposed filter.

```
makehdl('DCBlockerHDL/DF2T','TargetDirectory','df2t');
```

This figure shows the Xilinx® Vivado® synthesis results for the DF2T filter implementation. The synthesis tool was run with a 5 ns clock period to target 200 MHz operation, but the post-synthesis timing report shows that the minimum achievable clock period is only 6.754 ns or around 148 MHz. The critical path in this filter implementation goes through a DSP block, which is efficient use of hardware resources, but does not meet the 5 ns target.

Summary

Name	Path 1
Slack	-1.885ns
Source	u_Biquad_Filter/u_BiquadSection2_inst/numPostPipe1_reg[34]/C (rising edge-triggered cell FDCE clocked by clk {rise@
Destination	u_Biquad_Filter/u_BiquadSection2_inst/state1_reg[61]/D (rising edge-triggered cell FDCE clocked by clk {rise@0.000ns
Path Group	clk
Path Type	Setup (Max at Slow Process Corner)
Requirement	5.000ns (clk rise@5.000ns - clk rise@0.000ns)
Data Path Delay	6.754ns (logic 4.442ns (65.768%) route 2.312ns (34.232%))
Logic Levels	20 (CARRY8=7 DSP_A_B_DATA=1 DSP_ALU=2 DSP_M_DATA=1 DSP_MULTIPLIER=1 DSP_OUTPUT=2 DSP_PREADD_DA
Clock Path Skew	-0.123ns
Clock Uncertainty	0.035ns
Clock Net Delay (Source)	1.470ns (routing 0.185ns, distribution 1.285ns)
Clock Net Del...Destination)	1.268ns (routing 0.170ns, distribution 1.098ns)

The resource utilization for this filter structure, which has three second-order sections and one gain value, is 24 DSP elements, with 6 in the denominator and 18 in the numerator. The synthesis tool implemented the gain using shift-and-add logic.

Name	CLB LUTs (1182240)	CLB Registers (2364480)	CARRY8 (147780)	CLB (147780)	LUT as Logic (1182240)	LUT as Memory (591840)	DSPs (6840)	Bonded IOB (676)	HPIOB_M (312)	HPIOB_S (312)	GLOBAL CLOCK BUFFERs (1800)
DF2TDUT	1699	1346	203	392	1674	25	24	69	35	34	1
u_Biquad_Filter (Biquad_Filter)	1666	1254	203	377	1663	3	24	0	0	0	0

Pipelined Feedback Filter Structure for Higher Clock Rate

The pipelined structure creates a new denominator that has higher powers in Z (that is, more delay). This new denominator introduces more poles in the filter but these poles are canceled by a modified numerator. The new poles are created from the roots of the original denominator raised to a power determined by the amount of pipelining required. Since the poles are less than one for a stable filter, the new poles are smaller than the starting values, which adds to filter stability. This structure usually results in higher synthesis clock rates because of the additional pipelining. Use this command to generate HDL code for the pipelined feedback filter.

```
makehdl('DCBlockerHDL/Pipe','TargetDir','pipe');
```

The post-synthesis timing report shows that the design has a minimum clock period of 3.142 ns or around 318 MHz, which is more than twice the speed of the DF2T version. Since this filter processes one sample per clock, this rate corresponds to 318 megasamples/s.

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Des
Path 1	-1.716	7	2	u_Biquad_Filter...reg_reg[0][5]/C	u_Biquad_Filte...ST/ALU_OUT[0]	3.142	2.736	0.406	1.5	clk	clk
Path 2	-1.716	7	2	u_Biquad_Filter...reg_reg[0][5]/C	u_Biquad_Filt...T/ALU_OUT[10]	3.142	2.736	0.406	1.5	clk	clk
Path 3	-1.716	7	2	u_Biquad_Filter...reg_reg[0][5]/C	u_Biquad_Filt...T/ALU_OUT[11]	3.142	2.736	0.406	1.5	clk	clk
Path 4	-1.716	7	2	u_Biquad_Filter...reg_reg[0][5]/C	u_Biquad_Filt...T/ALU_OUT[12]	3.142	2.736	0.406	1.5	clk	clk
Path 5	-1.716	7	2	u_Biquad_Filter...reg_reg[0][5]/C	u_Biquad_Filt...T/ALU_OUT[13]	3.142	2.736	0.406	1.5	clk	clk
Path 6	-1.716	7	2	u_Biquad_Filter...reg_reg[0][5]/C	u_Biquad_Filt...T/ALU_OUT[14]	3.142	2.736	0.406	1.5	clk	clk
Path 7	-1.716	7	2	u_Biquad_Filter...reg_reg[0][5]/C	u_Biquad_Filt...T/ALU_OUT[15]	3.142	2.736	0.406	1.5	clk	clk

The resource utilization for this filter structure uses 84 DSP elements. There are 6 DSPs in the denominator and 18 DSPs in the numerator as before, and the new numerators (one per section) that cancel the added poles in the denominator use 60 DSP elements in total.

Name	CLB LUTs (1182240)	CLB Registers (2364480)	CARRY8 (147780)	CLB (147780)	LUT as Logic (1182240)	LUT as Memory (591840)	DSPs (6840)	Bonded IOB (676)	HPIOB_M (312)	HPIOB_S (312)	GLOBAL CLOCK BUFFERs (1800)
PipeDUT	6928	8119	761	1558	6572	356	84	69	35	34	1
u_Biquad_Filter (Biquad_Filter)	6863	7965	761	1526	6551	312	84	0	0	0	0

Pipelined Filter with Frame Input for Highest Clock Rate

This filter implementation increases throughput by using the frame input and output mode. This mode goes by many names including frame-based, vector, and super-sample-rate (SSR). All of these names mean that the algorithm processes more than one sample per clock cycle. You can use the buffer block in Simulink to switch between input sizes and try different designs. The model in this example is configured for four samples per clock. Use this command to generate HDL code for the pipelined feedback filter with frame-based input.

```
makehdl('DCBlockerHDL/FramePipe','TargetDir','framepipe');
```

The post-synthesis timing report shows that the achievable clock rate for the four-sample-per-clock implementation is around 275 MHz.

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Des
Path 1	-0.161	15	3	u_Biquad_Filter...[0]_replica_1/C	u_Biquad_Filter...1low_reg[66]/D	3.822	1.649	2.173	3.8	clk	clk
Path 2	-0.160	15	3	u_Biquad_Filter...[0]_replica_1/C	u_Biquad_Filter...1low_reg[64]/D	3.821	1.648	2.173	3.8	clk	clk
Path 3	-0.159	15	37	u_Biquad_Filter...o2high_reg[3]/C	u_Biquad_Filter...2low_reg[77]/D	3.877	1.679	2.198	3.8	clk	clk
Path 4	-0.158	15	37	u_Biquad_Filter...o2high_reg[3]/C	u_Biquad_Filter...2low_reg[75]/D	3.876	1.678	2.198	3.8	clk	clk
Path 5	-0.158	17	33	u_Biquad_Filte...Zero1_reg[1]/C	u_Biquad_Filter...high_reg[80]/D	3.894	1.760	2.134	3.8	clk	clk
Path 6	-0.152	15	31	u_Biquad_Filter...o2high_reg[7]/C	u_Biquad_Filter...2high_reg[80]/D	3.823	1.521	2.302	3.8	clk	clk
Path 7	-0.151	16	23	u_Biquad_Filter...o1high_reg[4]/C	u_Biquad_Filter...1high_reg[80]/D	3.866	1.879	1.987	3.8	clk	clk

The critical path for this implementation is in the pipelined addition of the modified numerator, which has large word lengths due to the fixed-point requirements of this particular filter design.

Delay Type	Incr (ns)	Path ...	Location	Chip	Netlist Resource(s)
FDCE (Prop_FFF2_SLICEL_C_Q)	(r) 0.099	4.083	Site: SLICE_X115Y391	S...1	u_Biquad_Filter/u_BiquadSection3_inst/delayline13NumZero3_reg[4]/Q
net (fo=19, routed)	0.358	4.441			u_Biquad_Filter/u_BiquadSection3_inst_n_3184
			Site: SLICE_X116Y398	S...1	u_Biquad_Filter/numPost15NumZero3high[8]_i_38/I1
LUT2 (Prop_E6LUT_SLICEM_I1_O)	(r) 0.148	4.589	Site: SLICE_X116Y398	S...1	u_Biquad_Filter/numPost15NumZero3high[8]_i_38/O
net (fo=1, routed)	0.015	4.604			u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high[8]_i_12
			Site: SLICE_X116Y398	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[8]_
CARRY8 (Prop_CARRY8_SLICEM_S[4]_CO[7])	(r) 0.198	4.802	Site: SLICE_X116Y398	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[8]_
net (fo=1, routed)	0.028	4.830			u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[8]_
			Site: SLICE_X116Y399	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[33]
CARRY8 (Prop_CARRY8_SLICEM_CI_CO[7])	(r) 0.023	4.853	Site: SLICE_X116Y399	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[33]
net (fo=1, routed)	0.028	4.881			u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[33]
			Site: SLICE_X116Y400	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[41]
CARRY8 (Prop_CARRY8_SLICEM_CI_O[6])	(r) 0.129	5.010	Site: SLICE_X116Y400	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[41]
net (fo=3, routed)	0.480	5.490			u_Biquad_Filter/u_BiquadSection3_inst/delayline13NumZero3_reg[20]_2[6
			Site: SLICE_X114Y393	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high[41]_i_5
LUT3 (Prop_G6LUT_SLICEL_I2_O)	(r) 0.114	5.604	Site: SLICE_X114Y393	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high[41]_i_5
net (fo=1, routed)	0.371	5.975			u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high[41]_i_5
			Site: SLICE_X114Y397	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[41]
CARRY8 (Prop_CARRY8_SLICEL_DI[2]_CO[7])	(r) 0.135	6.110	Site: SLICE_X114Y397	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[41]
net (fo=1, routed)	0.028	6.138			u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[41]
			Site: SLICE_X114Y398	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[49]
CARRY8 (Prop_CARRY8_SLICEL_CI_CO[7])	(r) 0.023	6.161	Site: SLICE_X114Y398	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[49]
net (fo=1, routed)	0.028	6.189			u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[49]
			Site: SLICE_X114Y399	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[57]
CARRY8 (Prop_CARRY8_SLICEL_CI_O[4])	(r) 0.109	6.298	Site: SLICE_X114Y399	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[57]
net (fo=3, routed)	0.382	6.680			u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high_reg[57]
			Site: SLICE_X114Y410	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high[49]_i_2
LUT3 (Prop_B6LUT_SLICEL_I2_O)	(r) 0.100	6.780	Site: SLICE_X114Y410	S...1	u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high[49]_i_2
net (fo=3, routed)	0.224	7.004			u_Biquad_Filter/u_BiquadSection3_inst/numPost15NumZero3high[49]_i_2

Even with the lower clock rate, the throughput per clock is four times larger so this design processes about 1.1 gigasample/s.

The resource utilization for this filter structure shows that this filter uses 510 DSP elements and many more CLB elements as well. This increase is because the size of the new numerator added to cancel poles in the expanded denominator scales with the number of samples per cycle.

Name	CLB LUTs (1182240)	CLB Registers (2364480)	CARRY8 (147780)	CLB (147780)	LUT as Logic (1182240)	LUT as Memory (591840)	DSPs (6840)	Bonded IOB (676)	HPIOB_M (312)	HPIOB_S (312)	GLOBAL CLOCK BUFFERs (1800)
FramePipeDUT	60208	59506	6959	10643	58166	2042	510	258	131	127	2
u_Biquad_Filter (Biquad_Filter)	59025	59119	6939	10534	57335	1690	510	0	0	0	0

The pipeline depth for the Biquad Filter block in this mode is 4 pipeline stages per multiply and one pipeline stage for every addition. For a given number of samples per clock, `FrameSize`, the size of the new numerator is `2*FrameSize*4-1`. For a frame size of 4, the new numerator has 31 taps.

Going Further

If you explore filter stability by quantizing the numerator and denominator using 16 bits instead of 20 bits, you can see that the filter becomes unstable. The filter coefficients in this example actually require a minimum word length of 19 bits for a stable filter.

You can try different frame sizes to increase throughput. As the frame size increases, the denominator coefficients get smaller and the filter numerics become more challenging. If the

coefficients are quantized to zero, the Biquad Filter block issues a warning. Sometimes, increasing the word length for the denominator is the only way to avoid numeric problems since the scale of the two internally computed coefficients can differ by more than the word length you select.

At some point, the size of the logic added to handle a larger frame size exceeds the size of an equivalent FIR filter. In this design, you can experiment with the transition bandwidth for an FIR filter, starting from the normalized frequency used to design the IIR filter of 0.001. An FIR filter that matches that frequency specification has over 5000 taps and consumes considerably more resources than the IIR filter in the Biquad Filter block.

See Also

Blocks
Biquad Filter

Frequency-Domain Filtering in HDL

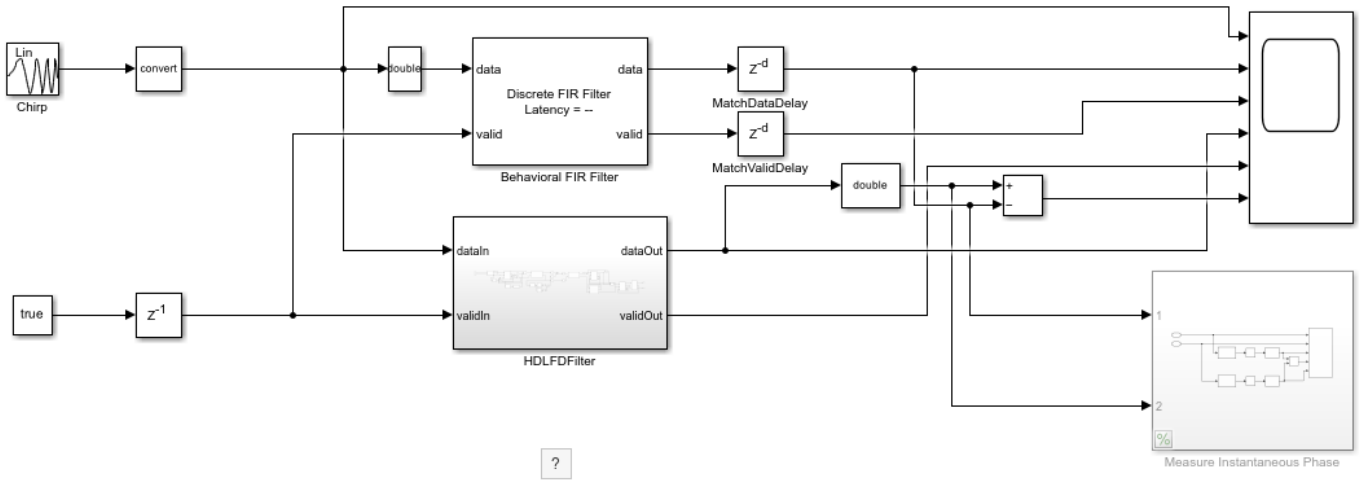
This example shows how to implement a filter in the frequency domain.

A sinusoidal input is filtered with the overlap-add method using a frequency-domain FIR filter built using the DSP HDL Toolbox FFT and IFFT blocks. The overlap-add and overlap-save filtering methods are area efficient ways to compute the discrete convolution of a signal with a finite-impulse response (FIR filter). The efficiency gain is particularly noticeable for high-order filters with a large number of taps. Match filters in communications systems, high-order FIR filters, and channel models on FPGA/ASIC hardware are all good examples of systems where frequency-domain filtering is a good choice.

Open and run the model.

The overlap-add algorithm [1] filters the input signal in the frequency domain. The input is divided into non-overlapping blocks which are linearly convolved with the FIR filter coefficients. The linear convolution of each block is computed by multiplying the discrete Fourier transforms (DFTs) of the block and the filter coefficients, and computing the inverse DFT of the product. For filter length M and FFT size N, the last M-1 samples of the linear convolution are added to the first M-1 samples of the next input sequence. The first N-M+1 samples of each summation result are output in sequence.

```
modelname = 'FreqDomainFiltHDL';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
```



Selecting the FFT Size

In selecting an FFT size for a particular filter, you must take care to make a hardware-friendly choice. If the filter length is an exact power of two, then your FFT size must be exactly double your filter length. In this case with a power of two size, there will be no time, except for the first filter length of

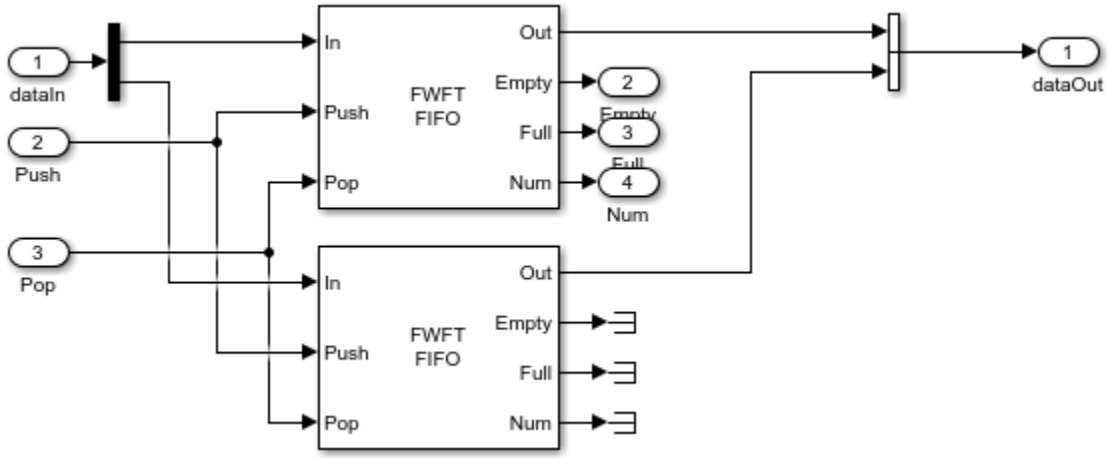
samples, where you will not be adding overlapping results. This simplifies the timing of the design since only the starting period must be handled specially.

If your filter length is not an exact power of two, then you must choose an FFT size that is larger than the next power of two from the filter length. In this example, the filter has 300 coefficients, then the next power of two would be 512, but you must choose 1024 for the FFT size. If you try to use the smaller size of 512 and check the timing, you will find that there are three overlapping regions, not two, and the hardware design becomes more complicated and area intensive.

Converting to Two-samples at a Time

In hardware, computing the FFT of N-(M-1) samples must be done by zero padding the input to the FFT block after the block of samples is sent in. This padding takes time away from processing the input samples, so somehow you must gain the time it takes to pad the input. One way to accomplish this is to buffer sample into a larger frame of say two samples and process these in a frame-based FFT block. Since you also need to buffer samples for a time, a FIFO is a natural buffer to use at the input.

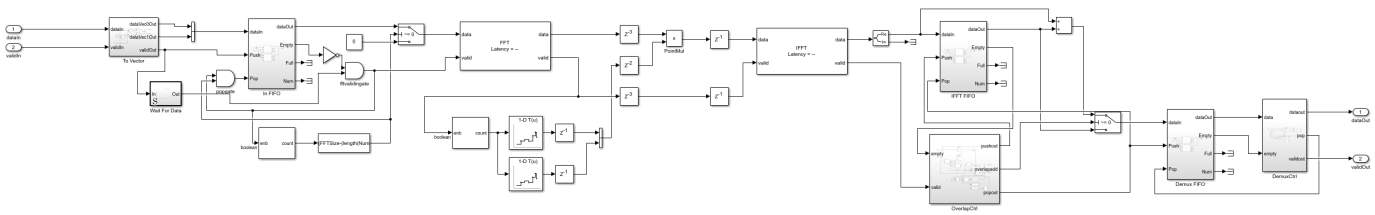
```
open_system([modelname '/HDLFDFilter/In FIFO'],'force');
```



Point Multiplication

The output of the FFT is then multiplied point-by-point, using a pipelined multiplier, with the FFT of the filter coefficients. In this example, the FFT of the filter coefficients is precomputed but for dynamic filter coefficients, but you can add another FFT to compute the coefficent transform. The result of the point multiply is then sent to an IFFT block to be transformed back into the time domain.

```
open_system([modelname '/HDLFDFilter'],'force');
```



Design Considerations

Another important consideration in making the frequency-domain filter hardware-friendly is that the shift from one sample at a time to two (or other power of 2) samples at a time means that the point at which the design switches from the non-overlapped region to the overlapped region must happen on a boundary that is divisible by two (or the selected power of 2) to avoid difficult timing in the design. You would like to switch between the non-overlapped region and the overlapped region on boundary that lines up with the change to a vector of samples. This adds a condition that your filter length must be divisible by your vector size. Also note that odd-order filters (with an even number of taps) will be required.

In this example, the filter transfer function is designed to be a low-pass filter so using a chirp or swept tone input signal allows you to easily visualize the frequency response of the filter directly in Simulink. You must design this filter with a larger than default Density Factor in order for the filter design to converge to a solution. The designed filter has 300 taps and is fully symmetric and so would require 150 multipliers to implement in the conventional way. By using a frequency-domain filtering approach, you are able to reduce this considerably. Care should be taken when comparing multiplier counts since the FIR implementation uses real-only multipliers but the FFT, point-multiply, and IFFT use complex multipliers and therefore require 3 or 4 real multipliers depending on the architecture selected.

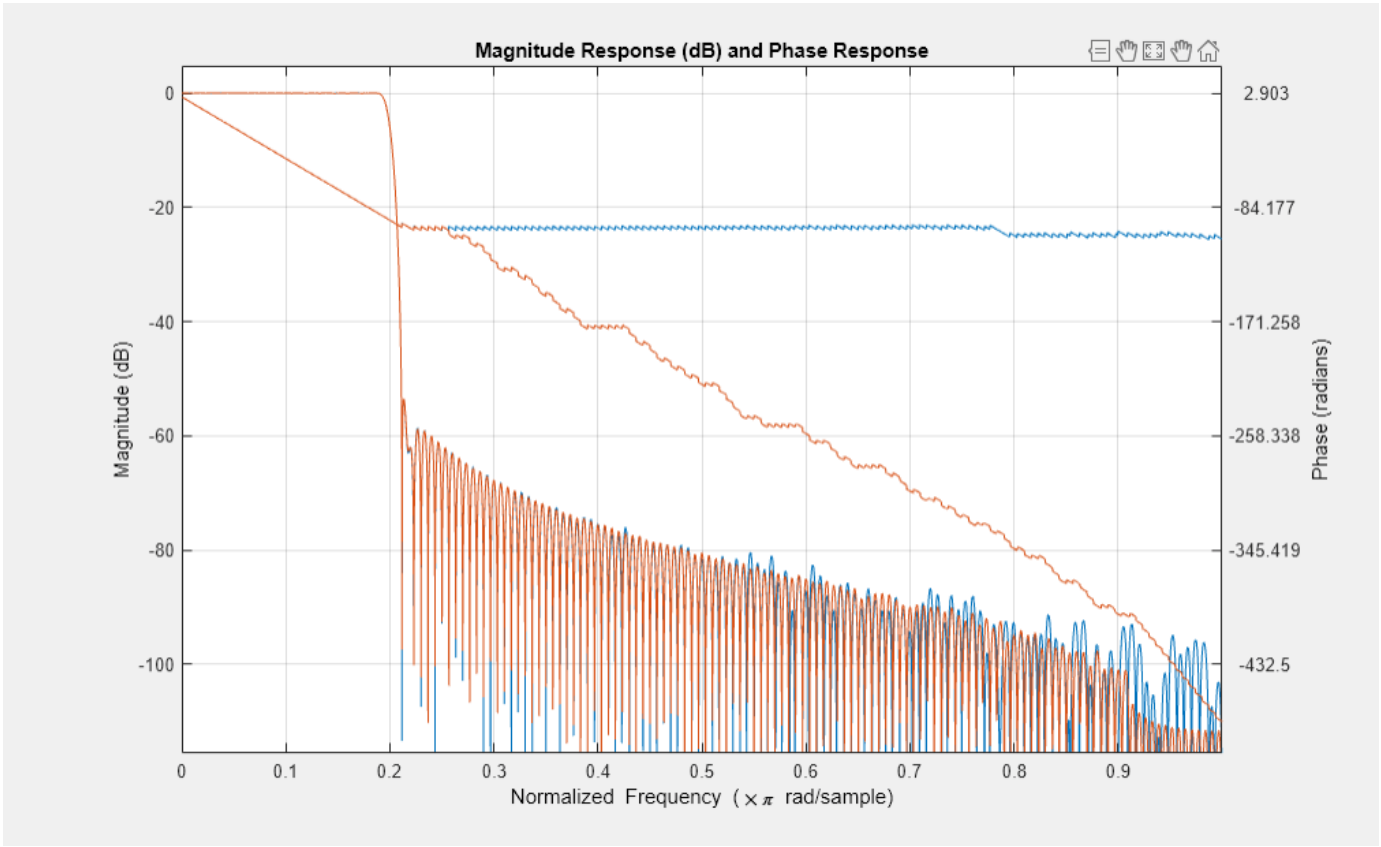
Exploring the Filter Response

You can explore the filter response by looking at the transformed, then quantized, then inverse transformed coefficients versus the original coefficients.

Num is the filter coefficients you calculated and FFTNum is the transform of those coefficients in double precision. You can simulate what the frequency response of the frequency-domain filter will be by quantizing FFTNum and then translating that back into the time domain.

The quantized transformed filter coefficients for the frequency-domain filter require different quantization settings since they can range between above 1 and below -1. Use the fixed-point tools to find the best precision binary point for the data by supplying only the signedness and the word length. Using fvtool for analysis allows you to overlay the data for both the original and the transformed filter coefficients.

```
QNum = fi(Num,1,18);
QFDNum = ifft(double(fi(FFTNum,1,18)));
QFDNum = QFDNum(1:300);
h = fvtool(double(QNum),1,double(QFDNum),1);
set(h,'OverlayedAnalysis','phase');
```



As you can see in fvtool, the magnitude responses of the two filters match very closely with some minor variations in the stopband up until frequencies very near Nyquist. The phase response matches very closely in the passband but varies significantly in the stopband. The original filter shows some ripples but nearly constant phase in the stopband while the frequency-domain filter shows a quantized phase response that falls away rapidly. The phase response of the stopband is not normally an factor in filter design, but it is good to know that the frequency-domain filter does not match the original here.

Going further

You can also see the change in stopband phase response by computing the instantaneous phase for each both the behavioral FIR and the frequency-domain filter. To do this you can uncomment the subsystem Measure Instantaneous Phase and re-run the simulation. The instantaneous phase of the real outputs is computed using Analytic Signal block followed by a Complex to Magnitude-Angle and finally an Unwrap block to make the phase continuous.

The DSP System Toolbox block Frequency-Domain FIR Filter has an option to partition the numerator to reduce latency. Using this option, the filter performs overlap-save or overlap-add on each partition, and combines the partial results to form the overall output. The latency is now reduced to the partition length. Using this technique on an HDL frequency-domain filter is also possible but you will likely use more multipliers depending on the size of your filter.

The transformed filter coefficients are computed from the FFT of a real sequence. This produces conjugate symmetric output so the upper bins of the FFT are the same values as the lower bins but reversed and with the sign of the imaginary part changed. This conjugate symmetry could allow you

to save half of the storage in the ROM used to store the transformed coefficients since you can map the upper bins to the lower bins in reverse order and conjugate the result. Note that the first bin representing the DC value of the sequence is not used in the reverse upper bins.

You can compare the synthesis results from an FIR filter with the results from the frequency-domain filter. These results can be hard to predict since they depend on the quantization settings used and since the FFT, point-multiply, and IFFT all operated on complex data, each multiplier is either 3 or 4 real multipliers depending on the block settings. Finding the area cross-over point between the two implementations is challenging to predict as well. In this implementation, you used two words at time in order gain the time needed for the overlap regions and allow continuous input, but if your application does not require continuous input, a lower area implementation is possible.

References

[1] Overlap-Add Algorithm: Proakis and Manolakis, Digital Signal Processing, 3rd ed, Prentice-Hall, Englewood Cliffs, NJ, 1996, pp. 430 - 433.

[2] Overlap-Save Algorithm: Oppenheim and Schafer, Discrete-Time Signal Processing, Prentice-Hall, Englewood Cliffs, NJ, 1989, pp. 558 - 560.

HDL Implementation of Four Channel Synthesizer and Channelizer

This example shows how to synthesize a series of four stereo signals into a broadband signal using a channel synthesizer and how to split the synthesized broadband signal into four individual narrowband signals using a channelizer. The Simulink® model in this example contains a `Synthesizer Channelizer` subsystem with Channel Synthesizer and Channelizer blocks. These blocks support HDL code generation.

Data Source

Use these stereo signals as model inputs.

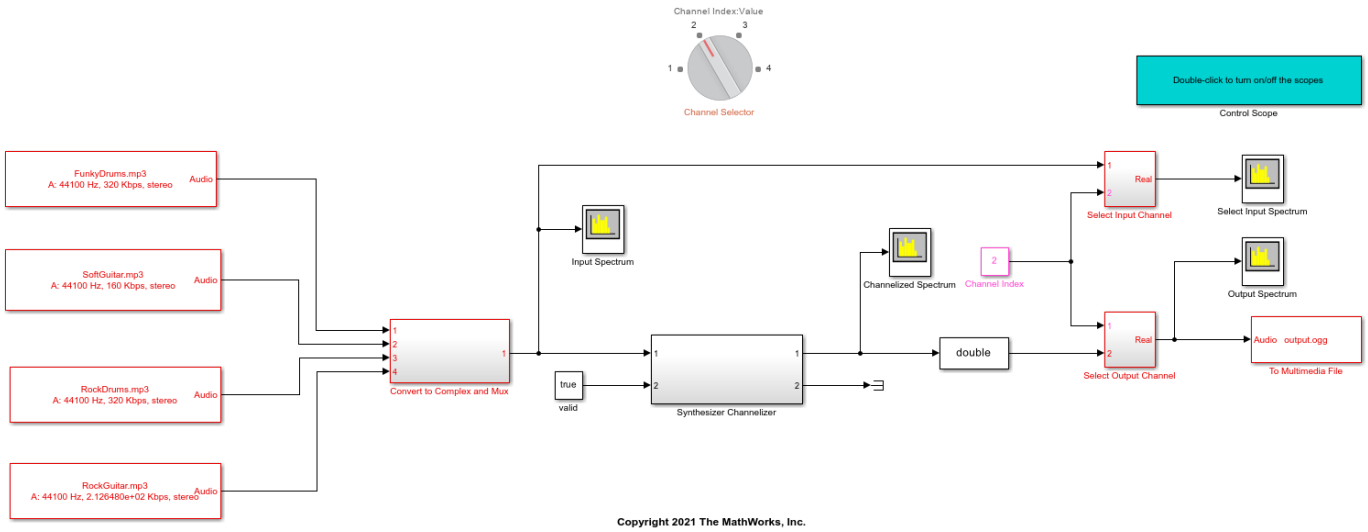
- `FunkyDrums.mp3`
- `SoftGuitar.mp3`
- `RockDrums.mp3`
- `RockGuitar.mp3`

Each stereo signal has two samples. The two channels in the model represent the left and right channels of a stereo signal. To store the channels of a stereo signal, convert each signal into a complex signal, multiplex, and then transpose to form a 1-by-2 vector.

Model Structure

Open the model.

```
model = 'HDLSynthesizerChannelizer';
open_system(model);
```



Model Parameters

The `InitFcn` callback sets up the model. To access the `InitFcn` callback, right-click the model, click **Model Properties**, open the **Callbacks** tab, and then click `InitFcn`. In the Channelizer block, set

the **Number of frequency bands** parameter to 4. Each band has one FIR filter and each FIR filter has 24 taps, so the model has total of 96 filter taps. The `designMultirateFIR` function designs a multirate FIR filter and generates the filter coefficients. Set the number of taps per band and stopband attenuation. Calculate the filter coefficients by using the `designMultiRateFIR` function.

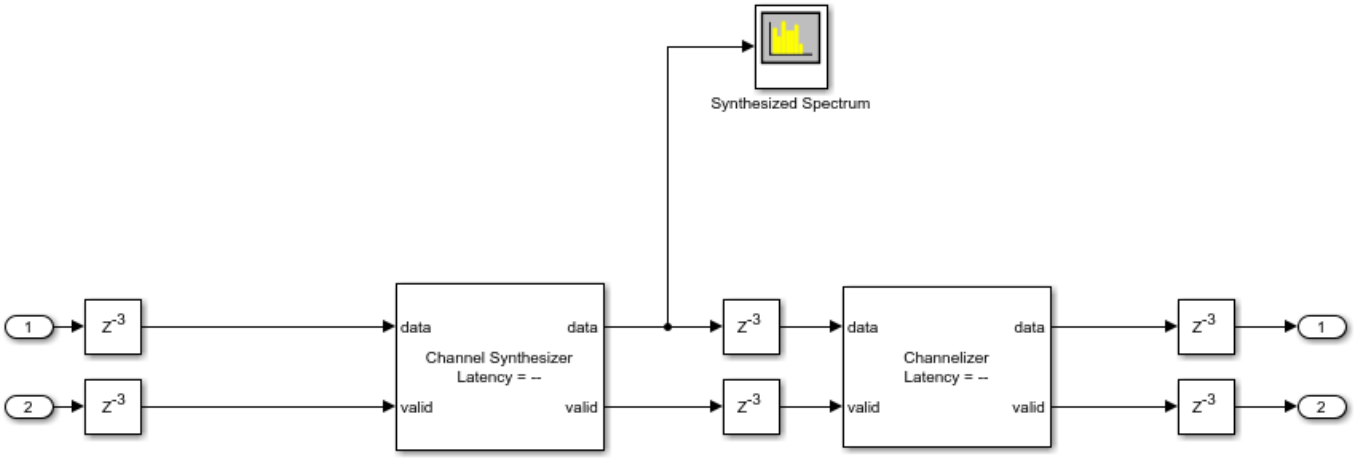
```
M = 4;                                          % number of frequency bands
P = 24;                                         % number of taps per band
Astop = 120;                                    % stop band attenuation
b = designMultirateFIR(1,M,ceil(P/2),Astop); % filter coefficients
```

Synthesizer and Channelizer

The Channel Synthesizer block synthesizes the input signals into a single broadband signal of size 4-by-1 by using an FFT-based synthesis filter bank. The model passes this signal to the Channelizer block. The Channelizer block splits the broadband input signal into four narrowband output signals. The output of the Channelizer block is a 1-by-4 vector, where each channel represents a narrowband.

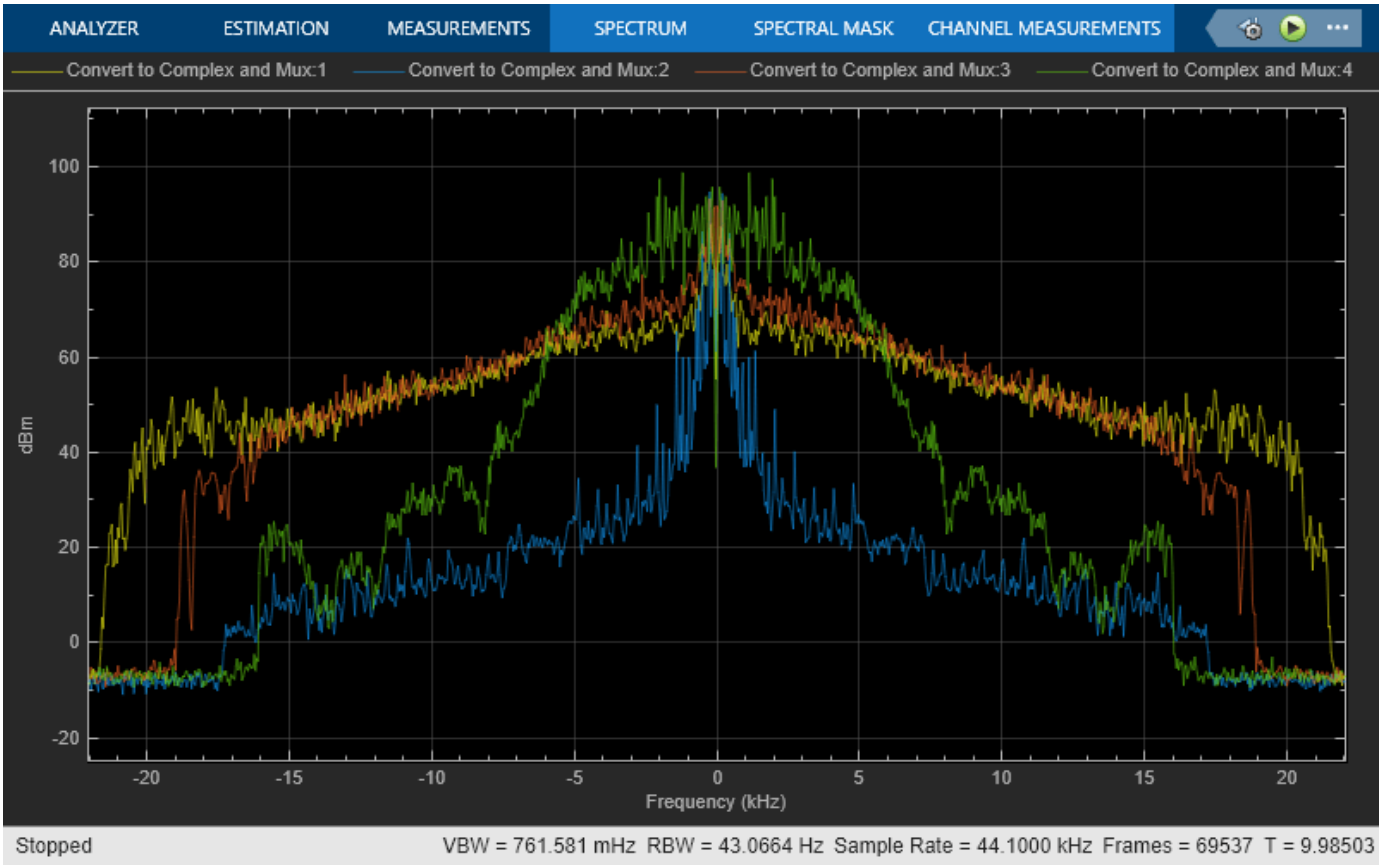
Open the `Synthesizer Channelizer` subsystem.

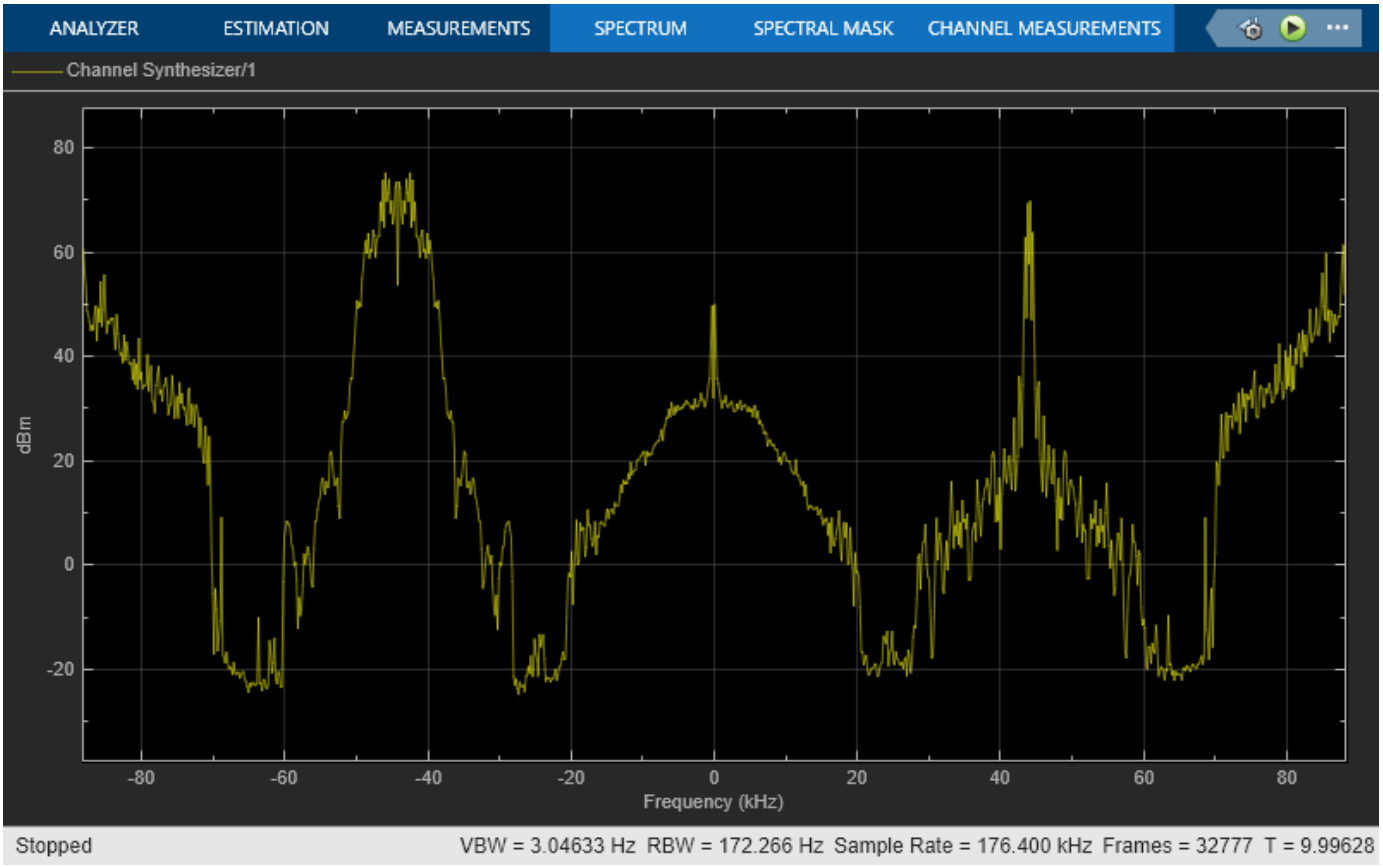
```
open_system([model '/Synthesizer Channelizer']);
```

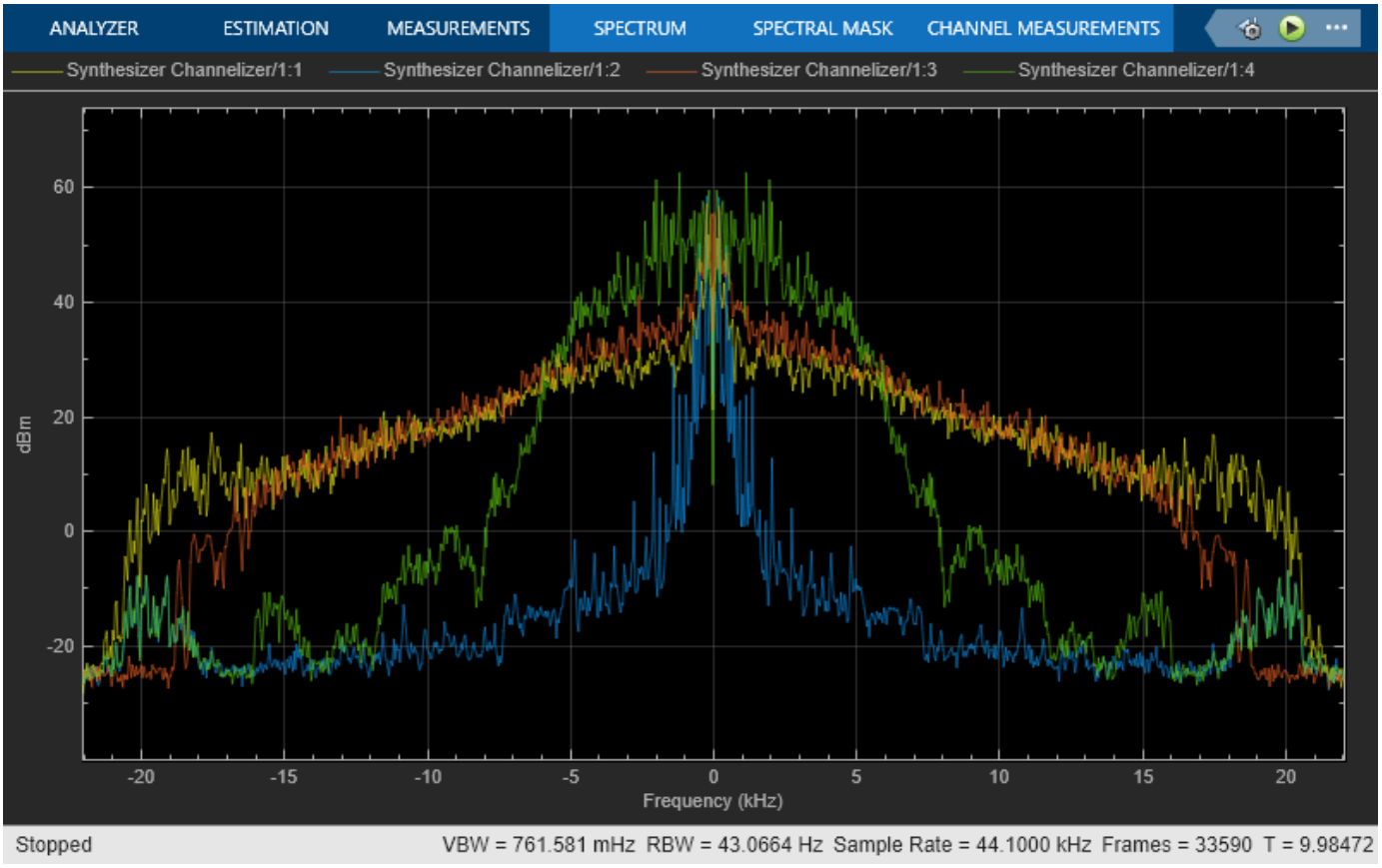


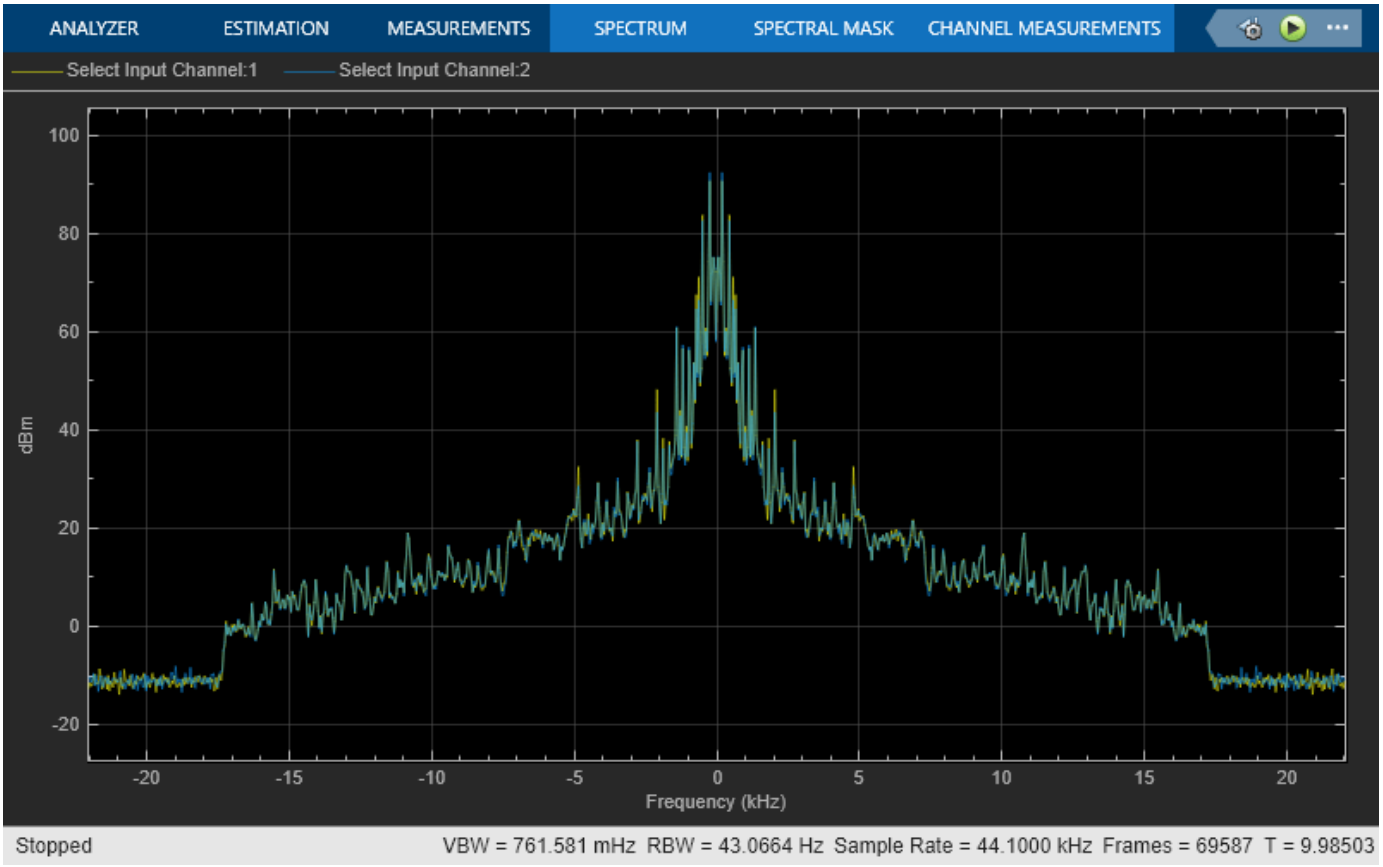
View the spectra of the input, synthesized signal, channelized signal, and synthesized-channelized output signal. The **Synthesized Spectrum** window shows the spectrum of the broadband signal. The **Channelized Spectrum** window shows the spectra of the four narrowband signals. The input and output spectra match for any selected signal.

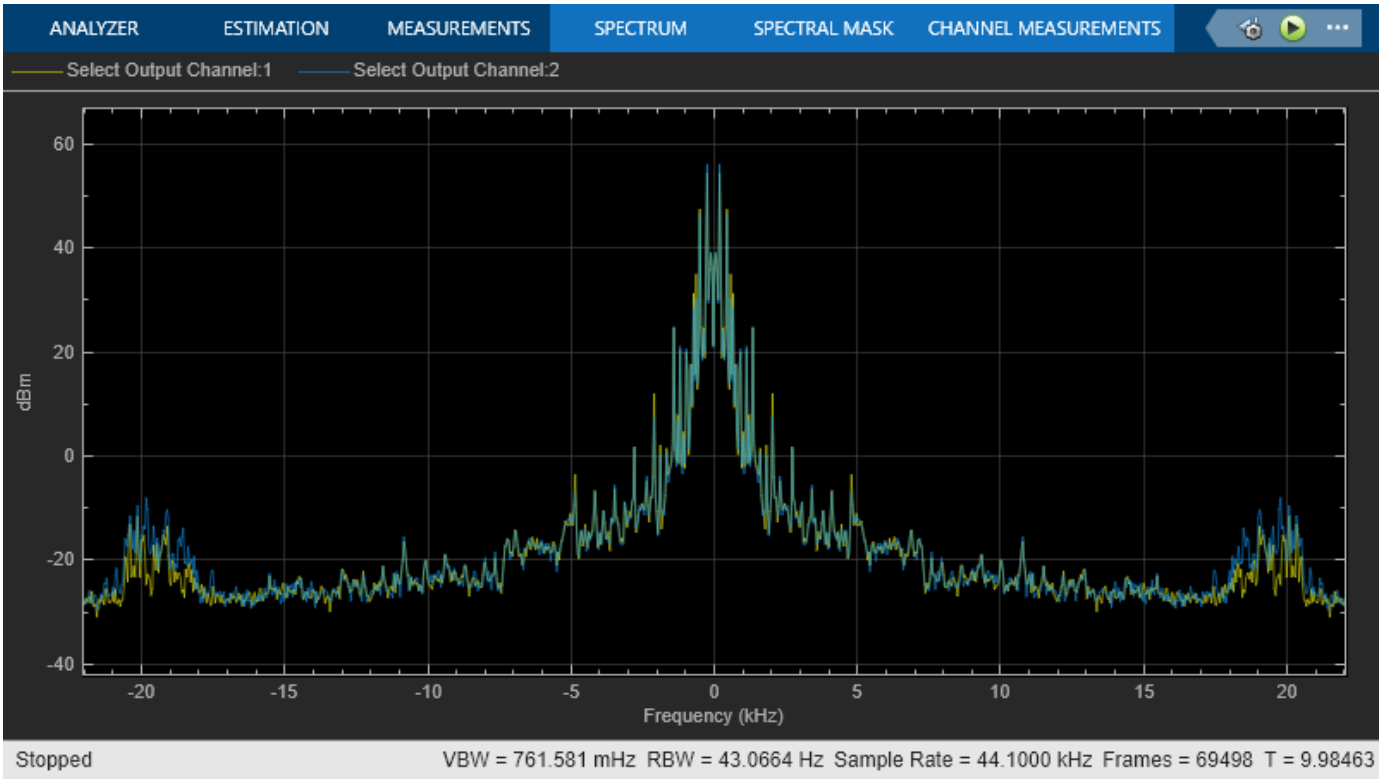
```
open_system([model '/Control Scope']);
sim(model)
```


ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
Convert to Complex and Mux:1
Convert to Complex and Mux:2
Convert to Complex and Mux:3
Convert to Complex and Mux:4
dBm
100
80
60
40
20
0
-20
-20
-15
-10
-5
0
5
10
15
20
Frequency (kHz)
Stopped
VBW = 761.581 mHz RBW = 43.0664 Hz Sample Rate = 44.1000 kHz Frames = 69537 T = 9.98503


ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
Channel Synthesizer/1
80
60
40
20
0
-20
dBm
-80
-60
-40
-20
0
20
40
60
80
Frequency (kHz)
Stopped
VBW = 3.04633 Hz RBW = 172.266 Hz Sample Rate = 176.400 kHz Frames = 32777 T = 9.99628


ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
Synthesizer Channelizer/1:1
Synthesizer Channelizer/1:2
Synthesizer Channelizer/1:3
Synthesizer Channelizer/1:4
60
40
20
0
-20
dBm
-20
-15
-10
-5
0
5
10
15
20
Frequency (kHz)
Stopped
VBW = 761.581 mHz RBW = 43.0664 Hz Sample Rate = 44.1000 kHz Frames = 33590 T = 9.98472


ANALYZER
ESTIMATION
MEASUREMENTS
SPECTRUM
SPECTRAL MASK
CHANNEL MEASUREMENTS
Select Input Channel:1
Select Input Channel:2
100
80
60
40
20
0
-20
dBm
-20
-15
-10
-5
0
5
10
15
20
Frequency (kHz)
Stopped
VBW = 761.581 mHz RBW = 43.0664 Hz Sample Rate = 44.1000 kHz Frames = 69587 T = 9.98503



Audio Capture

Collect the output samples and reproduce the audio.

```
% Create a |dsp.AudioFileReader| object with default settings.
fileReader = dsp.AudioFileReader('output.ogg');

% Return a structure containing information about the audio file.
fileInfo = audioinfo('output.ogg');

% Create an |audioDeviceWriter| object and specify the sample rate.
deviceWriter = audioDeviceWriter('SampleRate',fileInfo.SampleRate);

% Reduce the computational load of initialization in the audio stream loop.
setup(deviceWriter,zeros(fileReader.SamplesPerFrame,fileInfo.NumChannels));

% In the audio stream loop, read the audio signal frame from the file and
% write the frame to your device.
while ~isDone(fileReader)
audioData = fileReader();
deviceWriter(audioData);
end

% Close the input file and release the device.
release(fileReader);
release(deviceWriter);

% Save and close the model.
open_system([model '/Control Scope']);
close_system(model,0)
```

HDL Code Generation and FPGA Implementation

To generate the HDL code for this example, you must have the HDL Coder™ product. Generate HDL code and an HDL testbench for the `Synthesizer Channelizer` subsystem. Synthesize this subsystem on a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post-place-and-route resource utilization results. The design meets the timing requirement with a clock frequency of 191.4 MHz. The resources and frequencies vary with the parameter values that you select in the block mask.

```
T = table(...
    categorical({'LUTs'; 'Slice Registers'; 'DSPs'}),...
    categorical({'1215'; '2861'; '384'}),...
    'VariableNames',{'Resource','Usage'})
```

```
T =

  3x2 table

       Resource        Usage
    _______________    _____

    LUTs               1215
    Slice Registers    2861
    DSPs               384
```

See Also

Channelizer | Channel Synthesizer

Related Examples

- “High-Throughput Channelizer for FPGA”

Gigasamples-per-Second Correlator and Peak Detector

This example shows how to implement a high-throughput frame-based correlator and peak detector. The system is suitable for applications such as lidar and mm-wave radar.

Lidar and radar systems operate by transmitting pulses, receiving the sent pulse in a stream of data, and using signal processing techniques to determine where in the receiver stream the pulse is located. When you design such a system, one of the main considerations is the pulse width or pulse duration. Pulse width is a measure (in seconds) of how long each pulse transmission is. Longer pulses have more energy and can therefore increase the range of the system. Shorter pulses cannot travel as far, but they can achieve greater accuracy in resolving the distance between objects. The pulse width determines the signal bandwidth. For example, a pulse width of 2 ns results in a signal bandwidth of 500 MS/s. The signal bandwidth is then used to determine the minimum distance where separate objects can be resolved from one another. This distance is the *range ambiguity* and is equal to *c/(2*B)*, where *c* is the speed of light, and *B* is the signal bandwidth.

In high-precision lidar systems, the pulse width can often be as short as 4 ns. This width corresponds to a signal bandwidth of 250 MS/s and range ambiguity of 0.6 m. This calculation does not assume any additional signal processing, such as pulse compression, which could improve the accuracy. To meet the Nyquist rate, the received signal must be sampled at a rate of at least 500 MS/s. In practice, systems often oversample to improve performance. Typically, FPGAs run at up to 500 MHz. To process data with sample rates greater than the maximum clock rate, designs use frame-based processing, where each block operates on a vector of input data every clock cycle. In this way, the processing is parallel and sample rate is higher without an increase in clock rate.

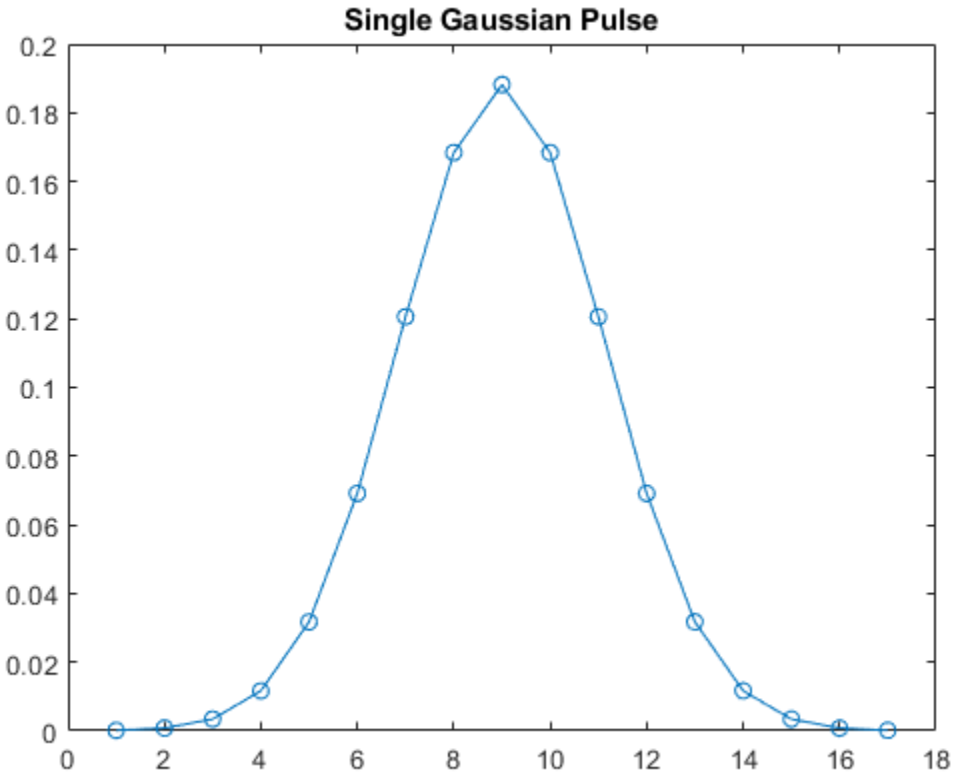
This example describes a correlation and peak detection system that uses a 250 MHz clock and an input frame of 16 samples. These parameters enable the system to process a 4 GS/s input stream oversampled by a factor of 16.

Waveform Generation and Matched Filter Design

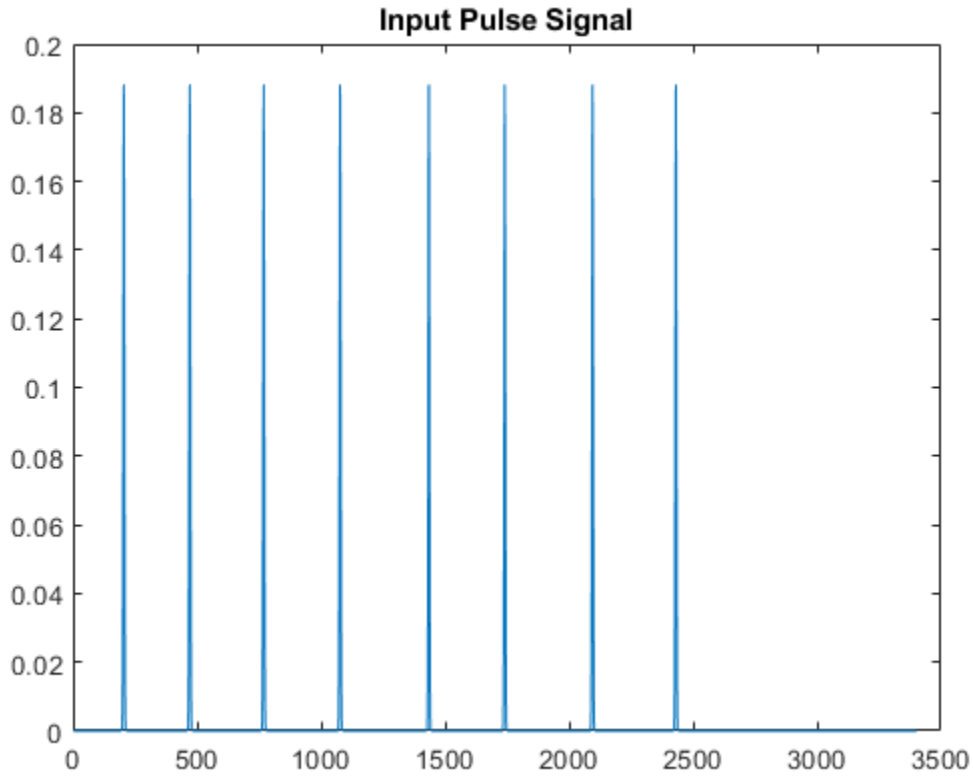
Broadly, lidar and radar systems can be split into pulsed waveform systems and continuous waveform systems. Pulsed waveform systems transmit bursts of data and then wait for a period, whereas continuous waveform systems are always transmitting. In each kind of system, you can apply different kinds of modulation to the waveform to enhance different properties such as range and resolution. This example shows a pulsed laser lidar system without signal modulation.

An ideal pulse has a rectangular shape in the time domain, corresponding to a sinc function in the frequency domain. The physical properties of laser systems mean that there is a ramp-up period to peak output, followed by a ramp-down period. Model this input by using a Gaussian function, then generate a stream of zeros and place pulses in the stream.

```
bt = 1; % 3 dB bandwidth-symbol time
sps = 16; % 16 times oversampled
span = 1; % 1 symbol
pulse = gaussdesign(bt,span,sps); % Pulse shape
plot(pulse,'o-');
title('Single Gaussian Pulse')
```



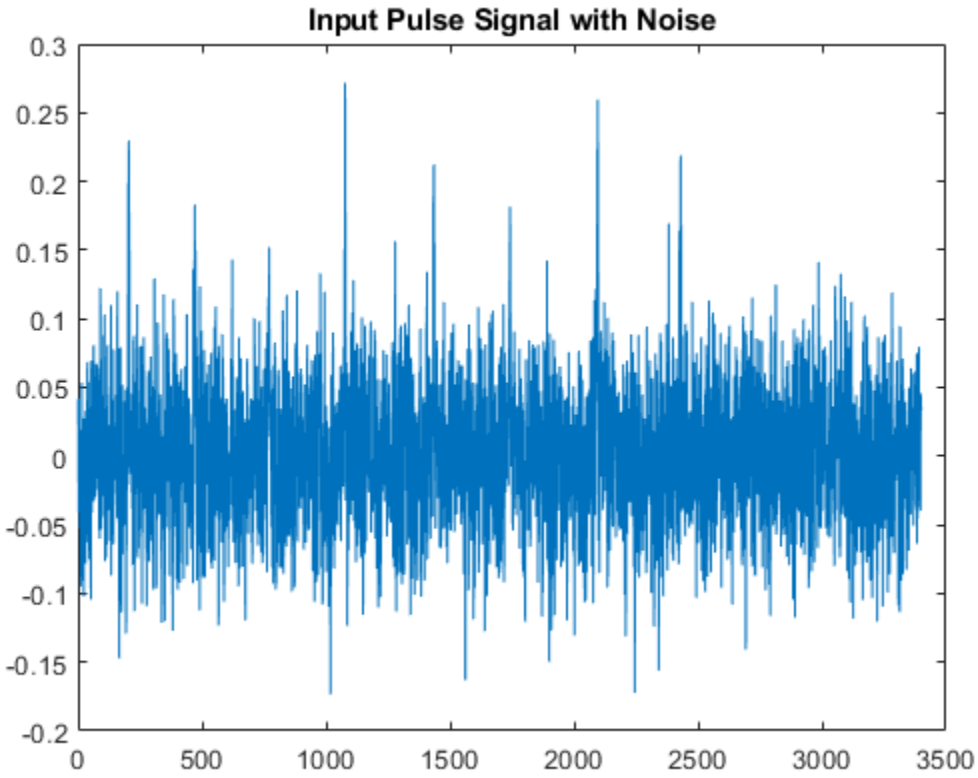

```
pulseLength = 17; % Number of samples per symbol
N = pulseLength * 200;  % 200 symbols
tx = zeros(N,1);
temp = primes(round(.80*N)); % Place pulses at a few scattered locations. Offset is a prime numbe
locations = temp(45:45:end);
for index = 1:length(locations)
    tx(locations(index):locations(index) + pulseLength - 1) = pulse;
end
figure
plot(tx);
title('Input Pulse Signal')
```



Now, add noise to simulate the channel and design a matched filter, which is the time-reversed conjugate of the pulse. Measure the noise inserted to make sure the calculation was correct. The pulse is symmetric and equivalent to the matched filter.

```
snr = 3;
pulseStream = awgn(tx,snr,10*log10(cov(pulse)),1);  % Add in AWGN.
figure
plot(pulseStream);
title('Input Pulse Signal with Noise')
noise = pulseStream - tx;
fprintf('Computed SNR is %3.2f \n',10*log10(cov(pulse)/cov(noise)));
h = flipud(conj(pulse));
isequal(h,pulse)
```

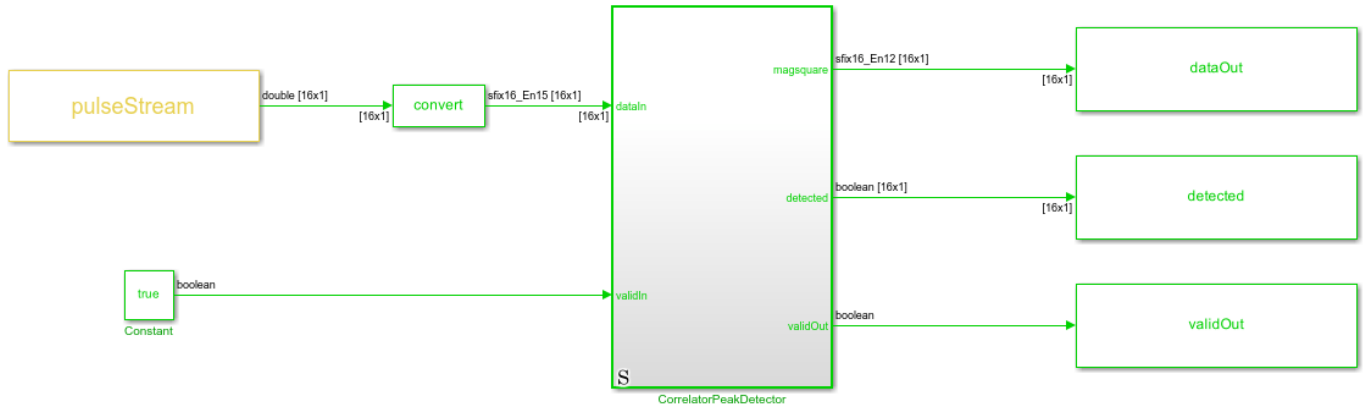
```
Computed SNR is 3.02

ans =

  logical

   1
```



Simulink Design

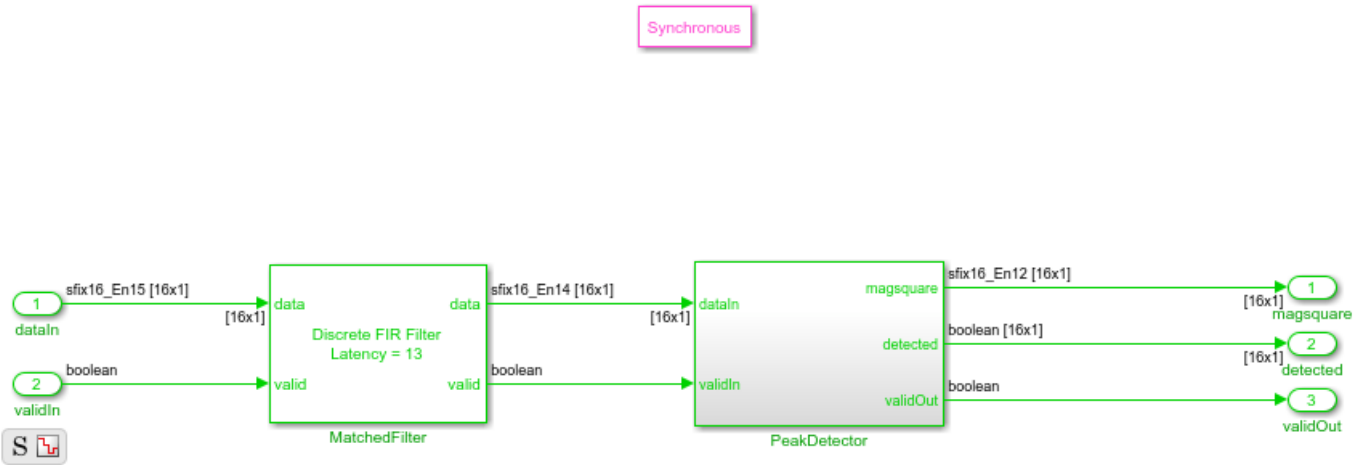
The example model implements a frame-based correlator and peak detector, using the input waveform and filter coefficients from the previous section. The `CorrelatorPeakDetector` subsystem has three outputs. The magnitude-squared matched filter output shows the boost in the signal-to-noise ratio (SNR) from the correlator. The detected output is a stream of Boolean values, which indicates when a pulse is detected. The valid output indicates when the output data is available.

```
vectorSize = 16;                                                %#ok<NASGU>
windowLength = 19;
model = 'CorrelationandPeakDetection';
load_system(model)
set_param(model,'SimulationCommand','Update')
open_system(model)
```



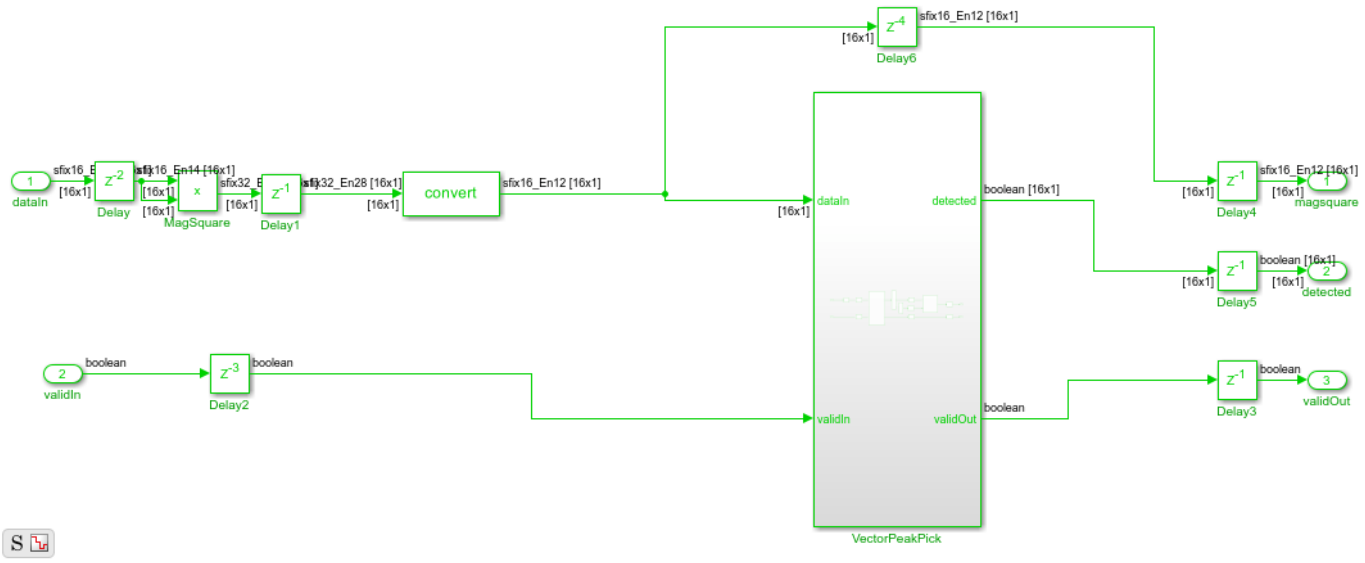
The DUT consists of a correlator or matched filter implemented using a Discrete FIR Filter block and a `PeakDetector` subsystem. The Discrete FIR Filter convolutes the input stream with the matched filter coefficients and passes the result to the `PeakDetector` subsystem. The `PeakDetector` uses a windowing method to determine local maxima.

```
model = 'CorrelationandPeakDetection/CorrelatorPeakDetector';
open_system(model)
```



The `PeakDetector` subsystem forms a sliding window of the FIR results, which is [19x1] for each element of the [16x1] input. An overall vector of [34x1] forms each subwindow. Inside the `VectorPeakPick` subsystem, the `VectorTappedDelay` subsystem forms this window and passes it to the `subtract_midpoint` subsystem, which implements the peak detection algorithm. The peak detection algorithm assumes that peaks are present when all values in the window subtracted by the middle value are less than or equal to 0. A For Each subsystem repeats this calculation 16 times to check each subwindow.

```
model = 'CorrelationandPeakDetection/CorrelatorPeakDetector/PeakDetector';
open_system(model)
```



Verification

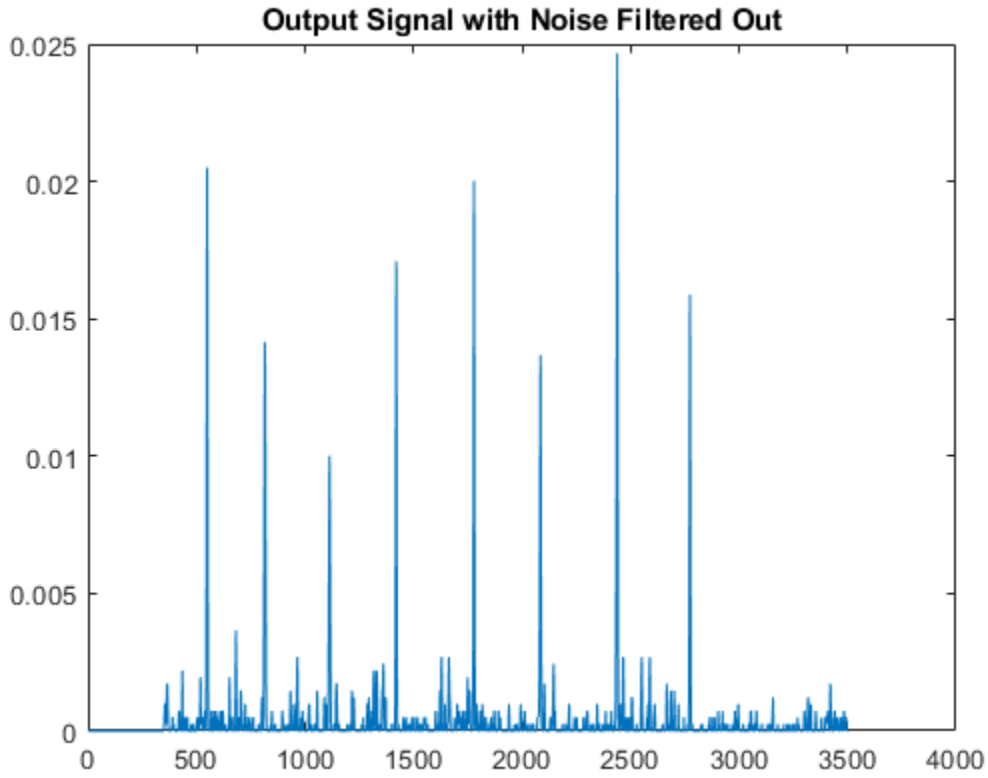
Next, run the simulation and verify that the model detects pulses where you expect them, using the information from waveform generation.

```
sim('CorrelationandPeakDetection.slx')
xlocations = find(detected==1); % Find locations where peaks were detected.
prevxlocation = 0; % Check for multiple points for the same peak in the loop below.
addr = 1;
locationsHDL = zeros(length(locations),1);
for ii = 1:1:length(xlocations)  % If there are multiple points for the same peak, pick one.
    if xlocations(ii) ~= prevxlocation+1
        locationsHDL(addr) = xlocations(ii);
        addr = addr + 1;
        prevxlocation = xlocations(ii);
    end
end
latencyHDL = round(mean(locationsHDL - locations')); % Latency is constant, and is the difference
locationsDetected = locationsHDL - latencyHDL                    %#ok<NOPTS,NASGU>
```

```
locationsDetected =

         197
         463
         761
        1069
        1427
        1733
        2087
        2422
```

Observe the magnitude squared output to see that the matched filter has significantly boosted the SNR.

```
plot(dataOut);
title('Output Signal with Noise Filtered Out')
```



HDL Implementation Results

To generate HDL code from this example model, you must have the HDL Coder™ product. HDL was generated for the `CorrelatorPeakDetector` subsystem and synthesized with Xilinx® Vivado™ targeting a Xilinx Zynq®-7000 SoC ZC706 evaluation board. The design meets timing with a constraint of 400 MHz. The table shows the post place-and-route resource utilization results.

```
T = table(...
    categorical({'DSP';'LUT';'Flip Flop';'BRAM'}), ...
    categorical({'288'; '9549'; '9092';'0'}), ...
    'VariableNames',{'Resource','Usage'})                    %#ok<NOPTS>
```

```
T =

  4x2 table

    Resource     Usage
    _________    _____

    DSP          288
    LUT          9549
    Flip Flop    9092
    BRAM         0
```

Sample Rate Modification Using Scalar Processing

You can adapt the model to process input with different sample rates. For example, you can process an input with 25 MS/s oversampled by a factor of 10 and, therefore, with a throughput of 250 MS/s using scalar rather than frame-based input. DSP HDL Toolbox™ library blocks automatically switch between frame and scalar algorithms according to the dimension of the data at the input port. In this example, you can choose to process frame or scalar input by modifying a single parameter, `vector_size`. The model automatically determines the correct dimensions for frame or scalar input and processes the data accordingly.

```
vectorSize = 1; % Scalar processing
sim('CorrelationandPeakDetection.slx') % Run simulation.
xlocations = find(detected==1); % Find locations where peaks are detected.
prevxlocation = 0; % Check for multiple points for the same peak in the loop below.
addr = 1;
locationsHDL = zeros(length(locations),1);
for ii = 1:1:length(xlocations)  % If there are multiple points for same peak, pick one.
    if xlocations(ii) ~= prevxlocation + 1
        locationsHDL(addr) = xlocations(ii);
        addr = addr + 1;
        prevxlocation = xlocations(ii);
    end
end
latencyHDL = round(mean(locationsHDL-locations')); % Latency is constant, and is the difference b
locationsDetected = locationsHDL-latencyHDL                    %#ok<NOPTS>
```

```
locationsDetected =

         197
         463
         761
        1069
        1427
        1733
        2087
        2422
```

NFC Digital Downconverter

This example shows how to decimate a 100 MS/s ADC signal down to 424 kS/s for a near-field communication (NFC) system.

Near-field communication systems operate at data rates of 424, 212, or 106 kS/s. Typical ADCs used in test and measurement (T&M), software-defined radio (SDR), and other prototyping equipment have much higher sample rates in the MS/s range. When you stream data from this equipment to a host computer or processor, the high sample rate can use a lot of memory and processor resources. Lowering the sample rate on the FPGA before streaming data to the host can save host resources. This example includes a digital downconverter (DDC) that converts a 100 MS/s input stream down to 424 kS/s output. The overall rate change is 235.8409566. This example differs from the "Implement Digital Downconverter for FPGA" on page 1-66 example in two ways. First, the decimation factor is much larger in this example, which requires different decimation filters. Second, this example must compute a fractional rate change, rather than a strictly integer rate change.

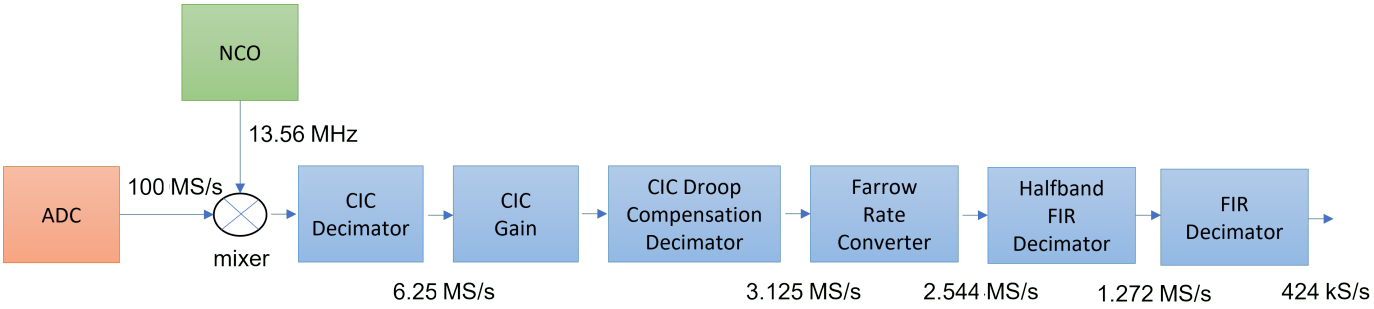
DDC Structure

The DDC consists of a numerically controlled oscillator (NCO), mixer, and decimating filter chain. The filter chain consists of a cascade integrator-comb (CIC) decimator, CIC gain correction, CIC compensation decimator (FIR), Farrow rate converter, halfband FIR decimator, and a final FIR decimator.

The overall response of the filter chain is equivalent to that of a single decimation filter with the same specification. However, splitting the filter into multiple decimation stages results in a more efficient design that uses fewer hardware resources.

The CIC decimator provides a large initial decimation factor, which enables subsequent filters to work at lower rates. The CIC compensation decimator improves the spectral response by compensating for the CIC droop while decimating by two. The halfband is an intermediate decimator, and the final decimator implements the precise `Fpass` and `Fstop` characteristics of the DDC. The lower sampling rates near the end of the chain mean the later filters can optimize resource use by sharing multipliers.

This figure shows a block diagram of the DDC:



DDC Design

To design the DDC, you use floating-point operations and filter-design functions in MATLAB®.

DDC Parameters

This example designs the DDC filter characteristics to meet these specifications for the given input sampling rate and carrier frequency.

```
FsIn = 100e6;       % Sampling rate of DDC input
FsOut = 424e3;      % Sampling rate of DDC output
Fc = 13.56e6;       % Carrier frequency
Fpass = 424e3;      % Passband frequency, equivalent to sampling rate
Fstop = 480e3;      % Stopband frequency
Ap = 0.1;           % Passband ripple
Ast = 60;           % Stopband attenuation
```

CIC Decimator

The first filter stage is a CIC decimator because of its ability to efficiently implement a large decimation factor. The response of a CIC filter is similar to a cascade of moving average filters, but a CIC filter uses no multiplication or division. As a result, the CIC filter has a large DC gain.

```
cicParams.DecimationFactor = 16;
cicParams.DifferentialDelay = 1;
cicParams.NumSections = 3;
cicParams.FsOut = FsIn/cicParams.DecimationFactor;

cicFilt = dsp.CICDecimator(cicParams.DecimationFactor, ...
    cicParams.DifferentialDelay,cicParams.NumSections)       %#ok<*NOPTS>
cicFilt.FixedPointDataType = 'Minimum section word lengths';
cicFilt.OutputWordLength = 18;
cicGain = gain(cicFilt)
```

```
cicFilt =

  dsp.CICDecimator with properties:

      DecimationFactor: 16
     DifferentialDelay: 1
           NumSections: 3
    FixedPointDataType: 'Full precision'


cicGain =

        4096
```

Because the CIC gain is a power of two, a hardware implementation can easily correct for the gain factor by using a shift operation. For analysis purposes, the example represents the gain correction in MATLAB with a one-tap `dsp.FIRFilter` System object™.

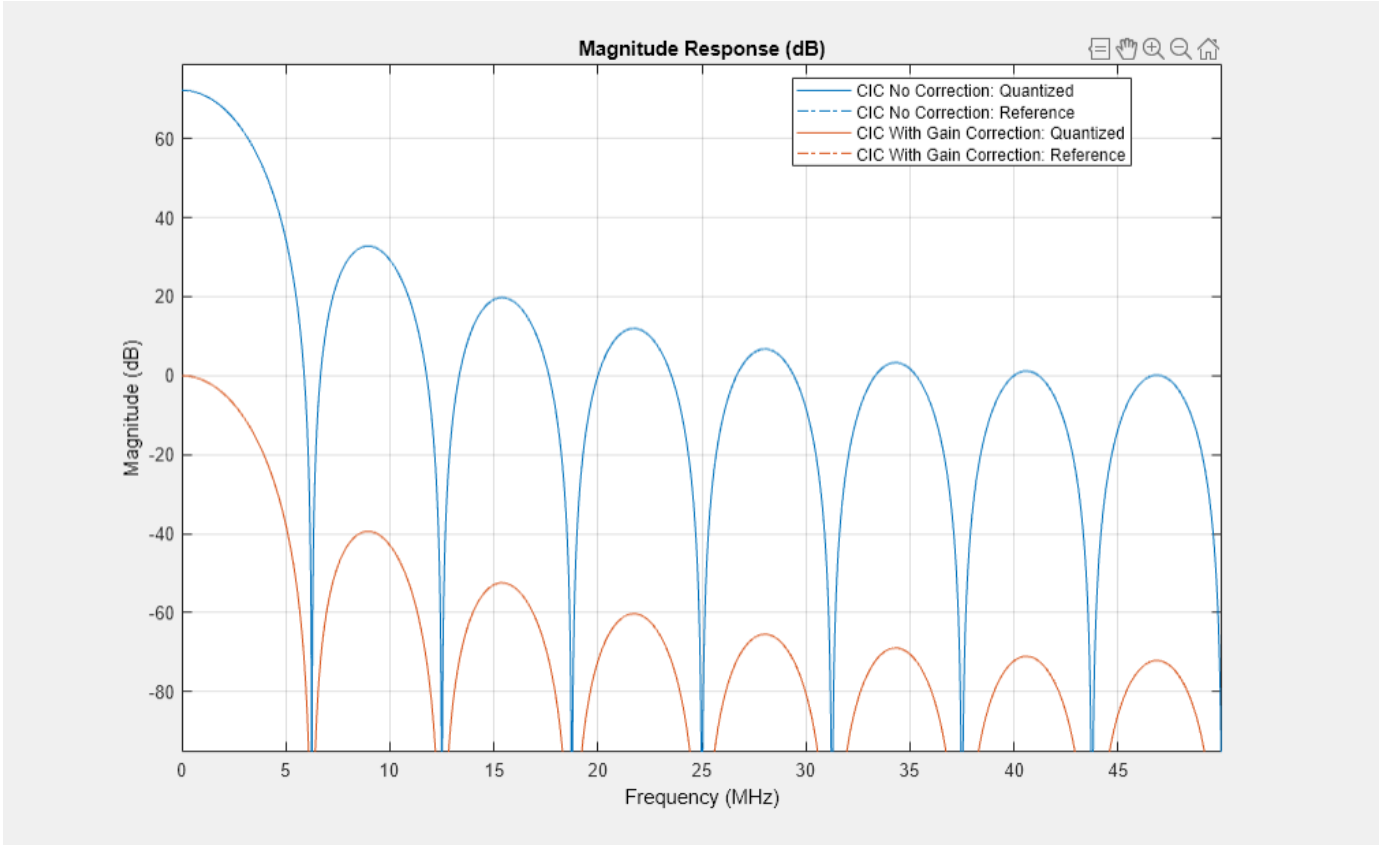
```
cicGainCorr = dsp.FIRFilter('Numerator',1/cicGain)
cicGainCorr.FullPrecisionOverride = false;
cicGainCorr.CoefficientsDataType = 'Custom';
cicGainCorr.CustomCoefficientsDataType = numerictype(fi(cicGainCorr.Numerator,1,16));
cicGainCorr.OutputDataType = 'Custom';
cicGainCorr.CustomOutputDataType = numerictype(1,18,16);
```

```
cicGainCorr =
```

```
  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: 2.4414e-04
    InitialConditions: 0

  Use get to show all properties
```

Display the magnitude response of the CIC filter with and without gain correction by using `fvtool`. For analysis, combine the CIC filter and the gain correction filter into a `dsp.FilterCascade` System object. CIC filters use fixed-point arithmetic internally, so `fvtool` plots both the quantized and unquantized responses.

```
ddcPlots.cicDecim = fvtool(...
    cicFilt, ...
    dsp.FilterCascade(cicFilt,cicGainCorr), ...
    'Fs',[FsIn,FsIn]);
legend(ddcPlots.cicDecim, ...
    'CIC No Correction', ...
    'CIC With Gain Correction');
```



CIC Droop Compensation Filter

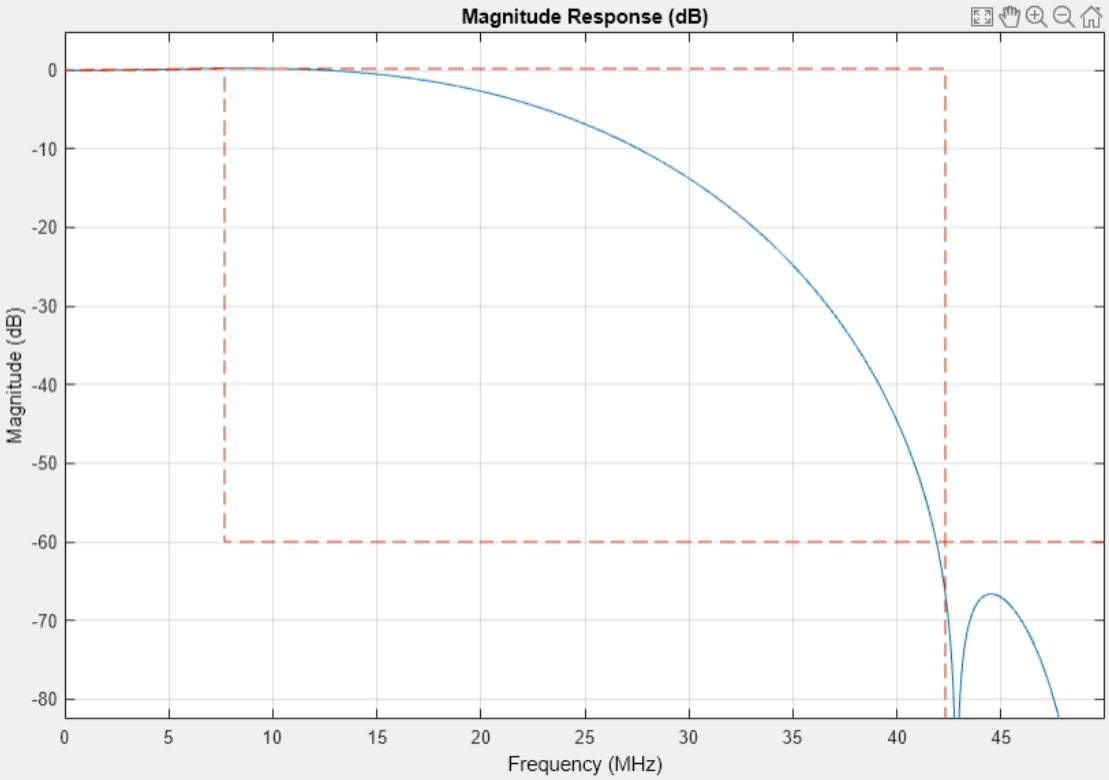
Because the magnitude response of the CIC filter has a significant *droop* within the passband region, the example uses a FIR-based droop compensation filter to flatten the passband response. The droop

compensator has the same properties as the CIC decimator. This filter implements decimation by a factor of two, so you must also specify bandlimiting characteristics for the filter. Use the `design` function to return a filter System object with the specified characteristics.

```
compParams.R = 2;                                % CIC compensation decimation factor
compParams.Fpass = Fstop;                        % CIC compensation passband frequency
compParams.FsOut = cicParams.FsOut/compParams.R; % New sampling rate
compParams.Fstop = compParams.FsOut - Fstop;     % CIC compensation stopband frequency
compParams.Ap = Ap;                              % Same passband ripple as overall filter
compParams.Ast = Ast;                            % Same stopband attenuation as overall filter

compSpec = fdesign.decimator(compParams.R,'ciccomp', ...
    cicParams.DifferentialDelay, ...
    cicParams.NumSections, ...
    cicParams.DecimationFactor, ...
    'Fp,Fst,Ap,Ast', ...
    compParams.Fpass,compParams.Fstop,compParams.Ap,compParams.Ast, ...
    cicParams.FsOut);
compFilt = design(compSpec,'SystemObject',true)


% Fixed point settings
compFilt.FullPrecisionOverride = false;
compFilt.CoefficientsDataType = 'Custom';
compFilt.CustomCoefficientsDataType = numerictype([],16,15);
compFilt.ProductDataType = 'Full precision';
compFilt.AccumulatorDataType = 'Full precision';
compFilt.OutputDataType = 'Custom';
compFilt.CustomOutputDataType = numerictype([],18,16);


compFilt =

  dsp.FIRDecimator with properties:

   Main
    DecimationFactor: 2
     NumeratorSource: 'Property'
           Numerator: [-0.0282 -0.0462 0.1348 0.4382 0.4382 0.1348 ... ]
           Structure: 'Direct form'

  Use get to show all properties
```

Plot the response of the CIC droop compensation filter.

```
ddcPlots.cicComp = fvtool(compFilt, ...
    'Fs',FsIn,'Legend','off');
```



Farrow Rate Converter

Next, a Farrow filter converts the sampling rate into an integer multiple of the output sampling rate. This conversion means the remainder of the decimation filter chain can use integer decimation factors. To choose the output rate of this stage, consider an integer that you can factor easily into a few stages. The sample rate at the input to the Farrow rate converter is 3.125 MS/s, which is approximately a 7.3702 multiple of 424 kS/s (output rate). If the Farrow rate converter output rate was 7*424 kS/s, then the further decimation stages could not factor this rate into integer multiples. This example uses 6*424 kS/s as the output sampling rate of this stage. This rate change is approximately 1.22, and this decimation ratio has acceptable aliasing. Farrow filters with larger rate changes may have increased aliasing.

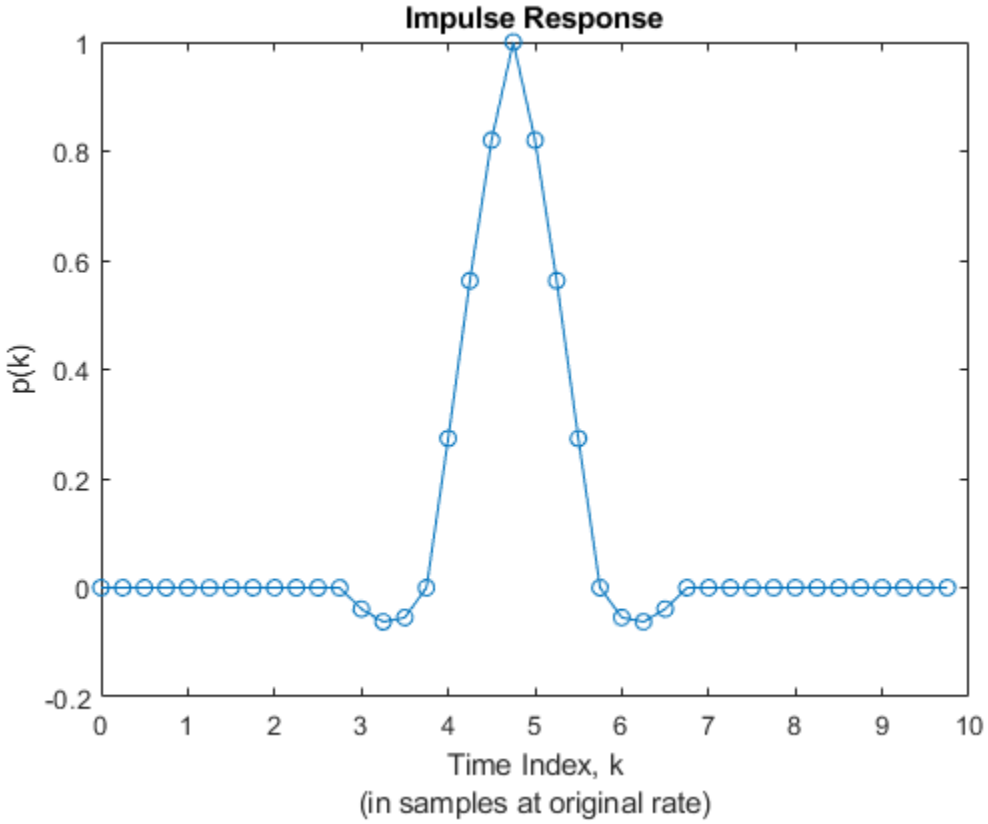
The default 3rd order LaGrange coefficients are derived from a closed-form solution and work for any rate change, so you do not need to design custom coefficients for this filter. The Farrow filter structure is the same as that used in the `dsp.VariableIntegerDelay` and `dsp.FarrowRateConverter` System objects.

These variables define the key parameters of the Farrow rate converter. `FsIn` and `FsOut` are the input and output rates, respectively.

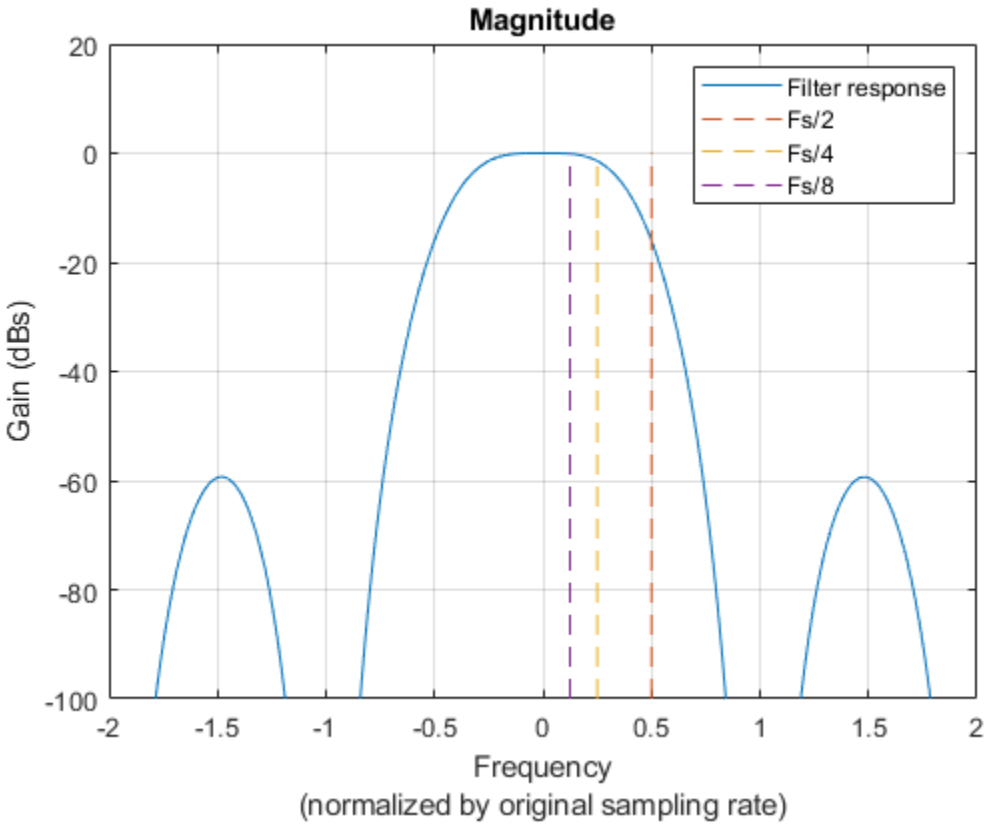
```
farrowParams.FsIn = compParams.FsOut ;
farrowParams.FsOut = 6*424e3;
farrowParams.RateChange = farrowParams.FsIn/farrowParams.FsOut;
farrowFilt = dsp.FarrowRateConverter('InputSampleRate',farrowParams.FsIn, 'OutputSampleRate',farr
```

To evaluate the Farrow rate converter, generate an impulse input with a length of `Lx` samples. Then, calculate the oversampled impulse response of the Farrow-based variable fractional delay object. Pass the impulse response through the object at `N` different fractional delays from `0` through `1-(1/N)`. Store the results in the oversampled response vector `p` and plot the impulse and magnitude response.

```
Lx = 10;
x = zeros(Lx,1);
x(1) = 1;

vfd = dsp.VariableFractionalDelay( ...
    'InterpolationMethod','Farrow');

N = 4;
Lp = N * Lx;
p = zeros(Lp,1);


for n=1:N
    p(n:N:end) = vfd(x,4+(N-n)/N);
end

figure(1); clf;
t = (0:length(p)-1)/N;
plot(t,p,'-o');
title("Impulse Response");
xlabel("Time Index, k" + newline + "(in samples at original rate)");
ylabel("p(k)");
```



```
figure(2); clf;
Lfft = 1024;
Pmag = 20*log(abs(fft(p/N,1024)));
f = (0:Lfft-1) * N / Lfft;
plot(f-N/2,fftshift(Pmag)); hold on;
ax = axis;
plot([1/2 1/2],[ax(3) ax(4)],'--');
plot([1/4 1/4],[ax(3) ax(4)],'--');
plot([1/8 1/8],[ax(3) ax(4)],'--');
axis([ax(1) ax(2) -100 20]);
grid on;
title("Magnitude");
xlabel("Frequency" + newline + "(normalized by original sampling rate)");
ylabel("Gain (dBs)");
legend("Filter response","Fs/2","Fs/4","Fs/8");
```



Halfband Decimator

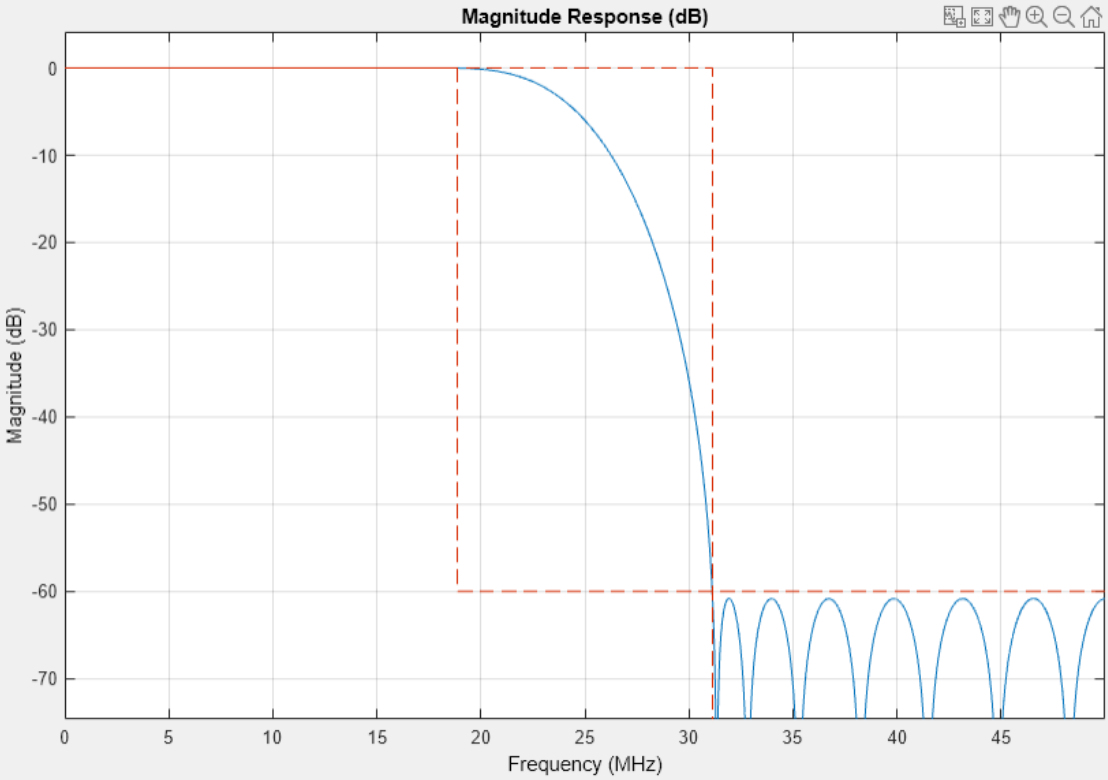
The halfband filter provides efficient decimation by two. Halfband filters are efficient because approximately half of their coefficients are equal to zero, and those multipliers are excluded from the hardware implementation.

```
hbParams.FsOut = farrowParams.FsOut/2;
hbParams.TransitionWidth = hbParams.FsOut - 2*Fstop;
hbParams.StopbandAttenuation = Ast;

hbSpec = fdesign.decimator(2,'halfband',...
```

```
    'Tw,Ast', ...
    hbParams.TransitionWidth, ...
    hbParams.StopbandAttenuation, ...
    farrowParams.FsOut);
hbFilt = design(hbSpec,'SystemObject',true)


% Fixed point settings
hbFilt.FullPrecisionOverride = false;
hbFilt.CoefficientsDataType = 'Custom';
hbFilt.CustomCoefficientsDataType = numerictype([],16,15);
hbFilt.ProductDataType = 'Full precision';
hbFilt.AccumulatorDataType = 'Full precision';
hbFilt.OutputDataType = 'Custom';
hbFilt.CustomOutputDataType = numerictype([],18,16);


hbFilt =

  dsp.FIRDecimator with properties:

   Main
    DecimationFactor: 2
     NumeratorSource: 'Property'
           Numerator: [0.0020 0 -0.0054 0 0.0124 0 -0.0247 0 0.0470 0 ... ]
           Structure: 'Direct form'

  Use get to show all properties
```

Plot the response of the halfband filter output.

```
ddcPlots.halfbandFIR = fvtool(hbFilt, ...
    'Fs',FsIn,'Legend','off');
```



Final FIR Decimator

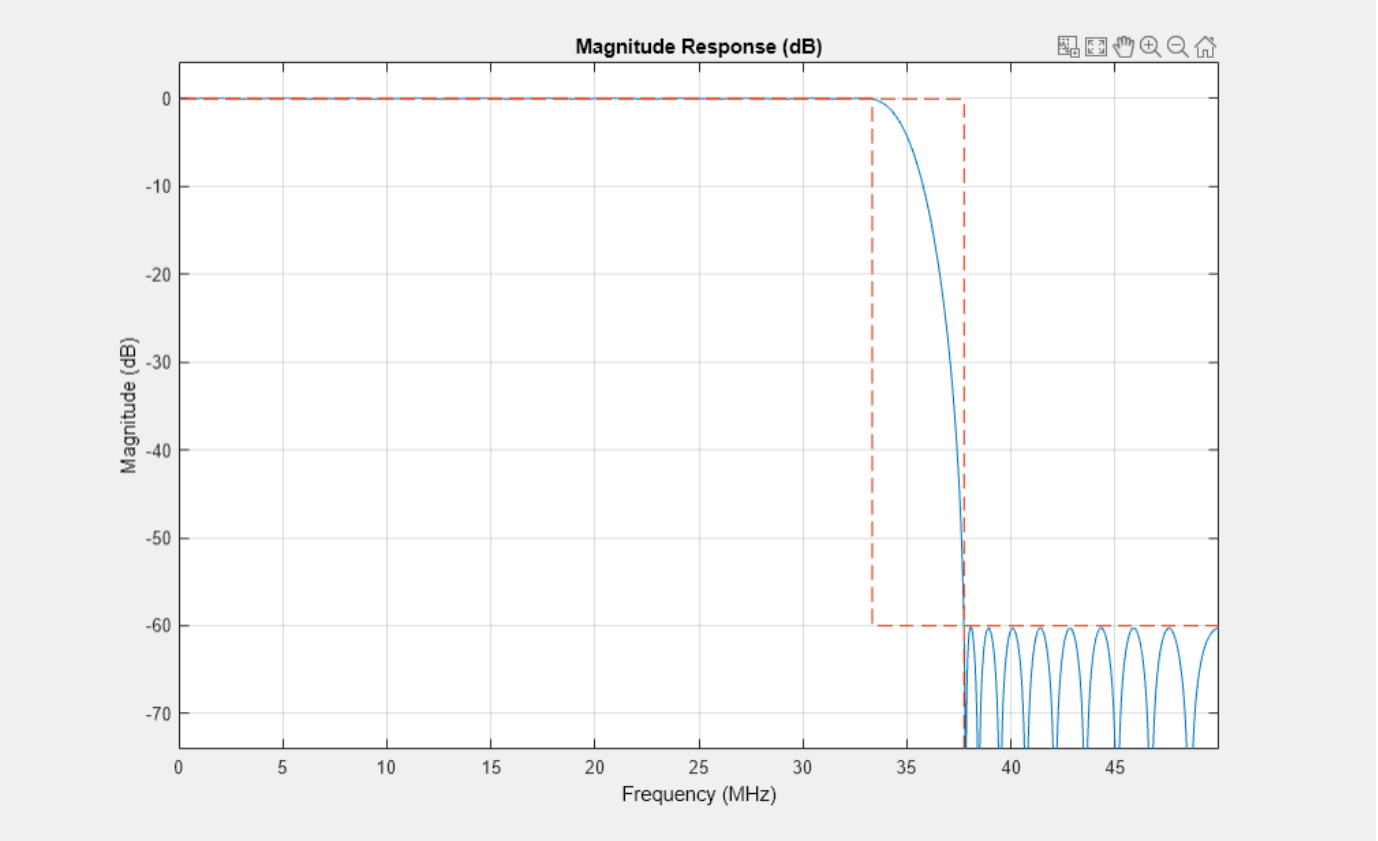
The final FIR implements the detailed passband and stopband characteristics of the DDC. This filter has more coefficients than the earlier FIR filters. However, because the filter operates at a lower sampling rate, it can use resource sharing for an efficient hardware implementation.

```
finalSpec = fdesign.decimator(2,'lowpass', ...
    'Fp,Fst,Ap,Ast',Fpass,Fstop,Ap,Ast,hbParams.FsOut);
finalFilt = design(finalSpec,'equiripple','SystemObject',true)


% Fixed point settings
finalFilt.FullPrecisionOverride = false;
finalFilt.CoefficientsDataType = 'Custom';
finalFilt.CustomCoefficientsDataType = numerictype([],16,15);
finalFilt.ProductDataType = 'Full precision';
finalFilt.AccumulatorDataType = 'Full precision';
finalFilt.OutputDataType = 'Custom';
finalFilt.CustomOutputDataType = numerictype([],18,16);


finalFilt =

  dsp.FIRDecimator with properties:

   Main
    DecimationFactor: 2
```

```
    NumeratorSource: 'Property'
          Numerator: [-0.0024 -0.0012 0.0020 -0.0017 -2.0236e-04 0.0027 ... ]
          Structure: 'Direct form'

  Use get to show all properties
```

Visualize the magnitude response of the final FIR.

```
ddcPlots.overallResponse = fvtool(finalFilt,'Fs',FsIn,'Legend','off');
```



HDL-Optimized Simulink Model

The next step in the design flow is to implement the DDC in Simulink using blocks that support HDL code generation.

Model Configuration

The model relies on variables in the MATLAB workspace to configure the blocks and settings. It uses the same filter chain variables defined earlier in the example. Next, define the NCO characteristics and the input signal. The example uses these characteristics to configure the NCO block.

Specify the desired frequency resolution and calculate the number of accumulator bits that you need to achieve the desired resolution. Set the desired spurious free dynamic range, and then define the number of quantized accumulator bits. The NCO uses the quantized output of the accumulator to address the sine lookup table. Also, compute the phase increment that the NCO uses to generate the

specified carrier frequency. The NCO applies phase dither to those accumulator bits that it removes during quantization.

```
nco.Fd = 1;
nco.AccWL = nextpow2(FsIn/nco.Fd)+1;
SFDR = 84;
nco.QuantAccWL = ceil((SFDR-12)/6);
nco.PhaseInc = round((-Fc*2^nco.AccWL)/FsIn);
nco.NumDitherBits = nco.AccWL-nco.QuantAccWL;
```

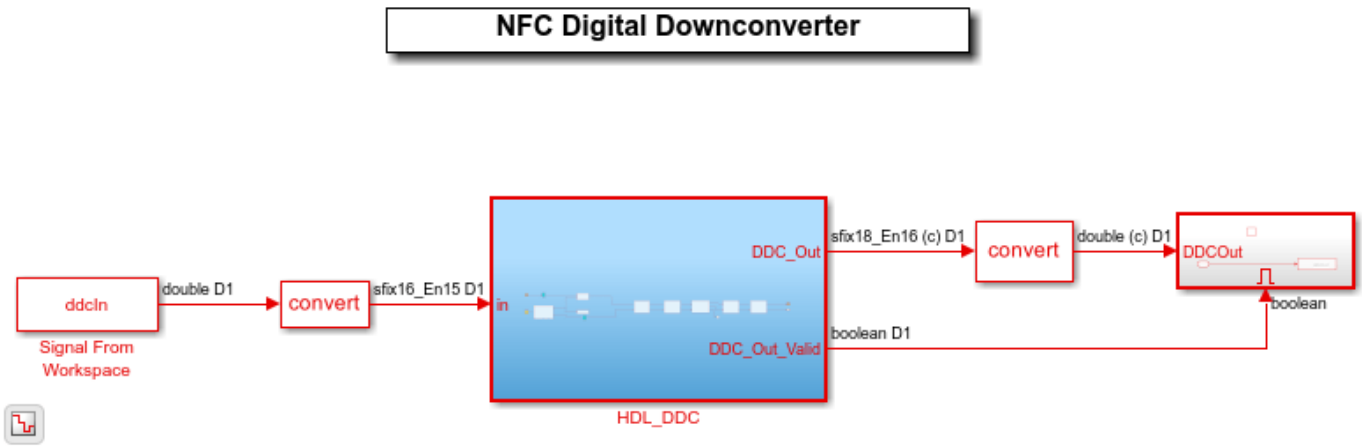
The input to the DDC comes from the `ddcIn` variable. For now, assign a dummy value for `ddcIn` so that the model can compute its data types. During testing, `ddcIn` provides input data to the model.

```
ddcIn = 0; %#ok<NASGU>
```

Model Structure

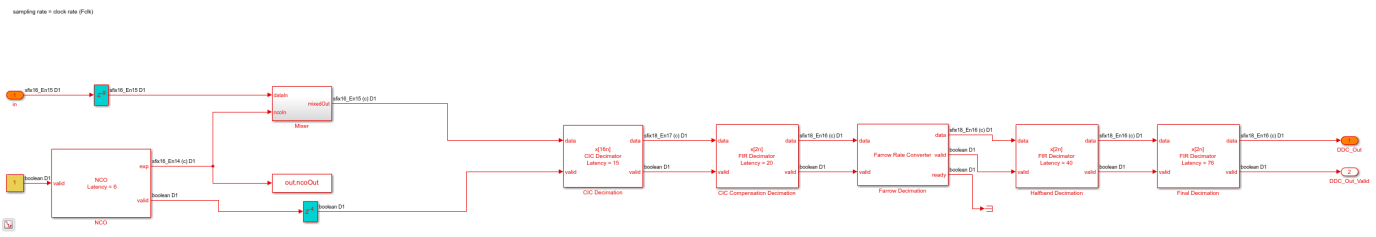
This figure shows the top level of the DDC Simulink model. The model imports the `ddcIn` variable from the MATLAB workspace by using a Signal From Workspace block, converts the input signal to 16-bit values, and applies the signal to the DDC. You can generate HDL code from the `HDL_DDC` subsystem.

```
modelName = 'DDCforNFCHDL';
 open_system(modelName);
 set_param(modelName,'SimulationCommand','Update');
 set_param(modelName,'Open','on');
```



The `HDL_DDC` subsystem implements the DDC filter. First, the NCO block generates a complex phasor at the carrier frequency. This signal goes to a mixer that multiplies the phasor with the input signal. Then, the mixer passes the output to the filter chain, which decimates it to 424 kS/s.

```
set_param([modelName '/HDL_DDC'],'Open','on');
```



NCO Block Parameters

The NCO block in the model is configured with the parameters defined in the `nco` structure.

CIC Decimation and Gain Correction

The first filter stage is a CIC decimator that is implemented with a CIC Decimator block. The block parameters are set to the `cicParams` structure values. To implement the gain correction, the block has the **Gain correction** parameter selected.

The model configures the filters by using the properties of the corresponding System objects. The CIC compensation, halfband decimation, and final decimation filters operate at effective sample rates that are lower than the clock rate by factors of 16, 32, and 64, respectively. The model implements these sample rates by using the **valid** input signal to indicate which samples are valid at each rate. The signals in the filter chain all have the same Simulink sample time.

The CIC Compensation, Halfband Decimation, and Final Decimation filters are each implemented by an FIR Decimator. By setting the **Minimum number of cycles between valid input samples** parameter, you can use the invalid cycles between input samples to share resources. For example, the spacing between every input of CIC Compensation Decimator is 16.

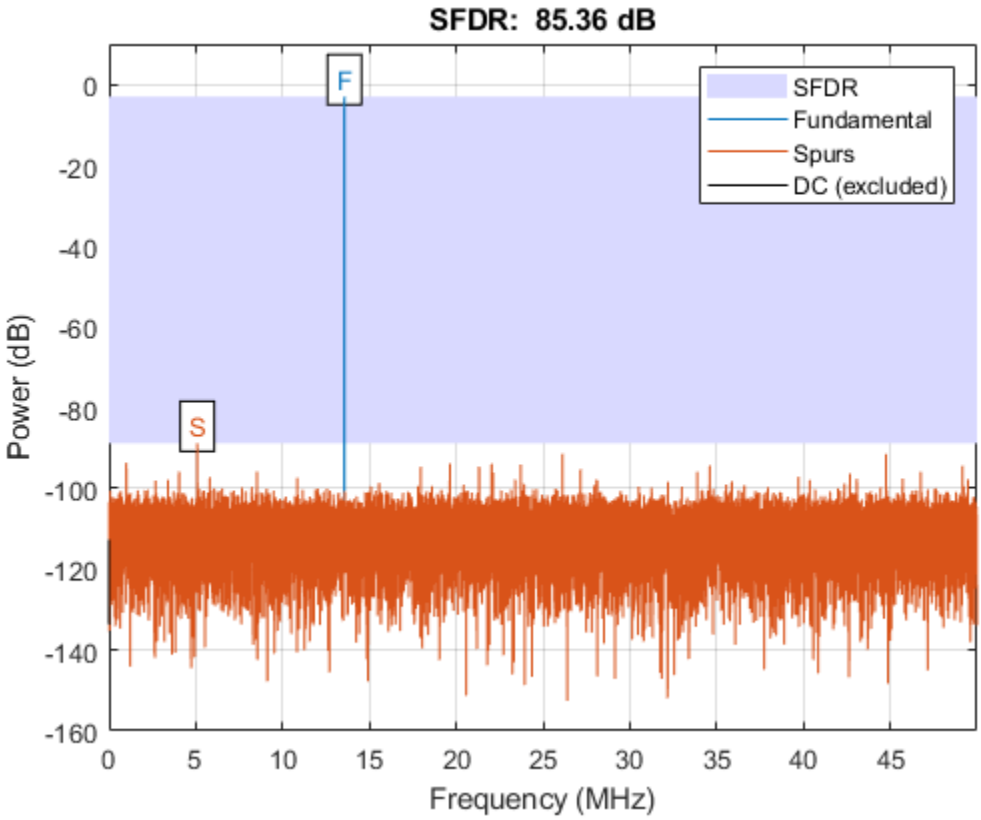
The CIC Compensation Decimation, Halfband Decimation, and Final Decimation filters are fully serial and use one multiplier for the real parts of the samples and one multiplier for the imaginary parts. Each filter uses two multipliers in total. This resource saving is possible because of the large rate change. The Farrow Rate Converter uses one fully serial FIR filter (one multiplier) for each row of the coefficient matrix, for the real part of the samples and the imaginary part. The filter uses eight multipliers in total.

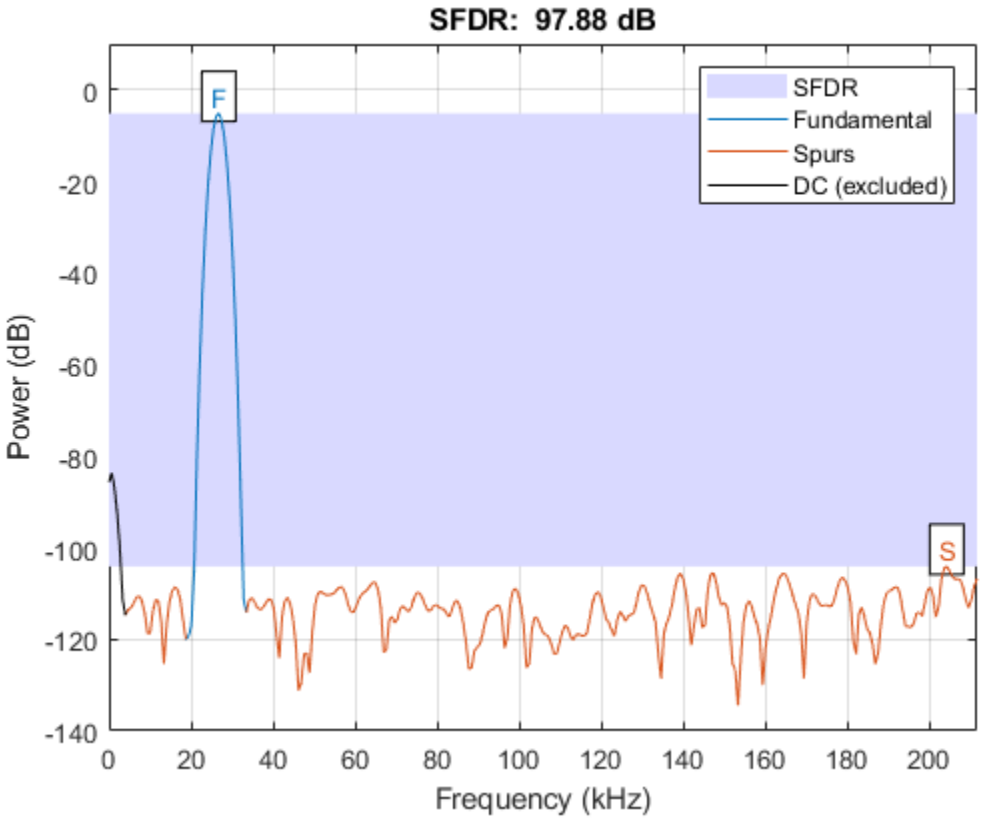
Sinusoid on Carrier Test and Verification

To test the DDC, modulate a 424 kHz sinusoid onto the carrier frequency and pass the modulated sine wave through the DDC. Then, measure the spurious-free dynamic range (SFDR) of the resulting tone and the SFDR of the NCO output. Plot the SFDR of the NCO and the fixed-point DDC output.

```
% Initialize random seed before executing any simulations.
rng(0);

% Generate a 40 kHz test tone, modulated onto the carrier.
ddcIn = DDCTestUtils.GenerateTestTone(40e3,Fc);

% Demodulate the test signal by running the Simulink model.
out = sim(modelName);

% Measure the SFDR of the NCO, floating-point DDC outputs, and fixed-point
% DDC outputs.
results.sfdrNCO = sfdr(real(out.ncoOut),FsIn);
results.sfdrFixedDDC = sfdr(real(out.ddcFixedOut),FsOut);

disp('SFDR Measurements');
disp(['   Fixed-point NCO SFDR: ',num2str(results.sfdrNCO) ' dB']);
disp(['   Optimized fixed-point DDC SFDR: ',num2str(results.sfdrFixedDDC) ' dB']);
fprintf(newline);

% Plot the SFDR of the NCO and fixed-point DDC outputs.
ddcPlots.ncoOutSDFR = figure;
sfdr(real(out.ncoOut),FsIn);
```

```
ddcPlots.OptddcOutSFDR = figure;
sfdr(real(out.ddcFixedOut),FsOut);

SFDR Measurements
   Fixed-point NCO SFDR: 85.3637 dB
   Optimized fixed-point DDC SFDR: 97.8753 dB
```





HDL Code Generation and FPGA Implementation

To generate the HDL code for this example, you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `HDL_DDC` subsystem. For this example, the DDC was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place-and-route resource utilization results. The design met timing with a clock frequency of 340 MHz.

```
T = table(...
    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}), ...
    categorical({'1547'; '386'; '3351'; '2'; '36'}), ...
    'VariableNames',{'Resource','Usage'})
```

```
T =

  5x2 table

    Resource    Usage
    ________    _____

     LUT        1547
     LUTRAM     386
     FF         3351
     BRAM       2
```

DSP 36

Implement Digital Downconverter for FPGA

This example shows how to design a digital downconverter (DDC) for radio communication applications such as LTE, and generate HDL code.

Introduction

DDCs are widely used in digital communication receivers to convert radio frequency (RF) or intermediate frequency (IF) signals to baseband. The DDC operation shifts the signal to a lower frequency and reduces its sampling rate to facilitate subsequent processing stages. The DDC in this example performs complex frequency translation followed by sample rate conversion using a four-stage filter chain. The example starts by designing the DDC with DSP System Toolbox™ functions in floating point. Then, each stage is converted to fixed point, and used in a Simulink® model that generates synthesizable HDL code. The example uses these two test signals to demonstrate and verify the DDC operation:

- A sinusoid that is modulated onto a 32 MHz IF carrier.
- An LTE downlink signal with a bandwidth of 1.4 MHz modulated onto a 32 MHz IF carrier.

The example compares the signal quality at the output of the floating-point DDC with the signal quality at the output of the fixed-point DDC.

Finally, the example presents an implementation of the filter chain for FPGAs, and synthesis results.

This example uses `DDCTestUtils`, a helper class that contains functions for generating stimulus and analyzing the DDC output. For more information, see the `DDCTestUtils.m` file.

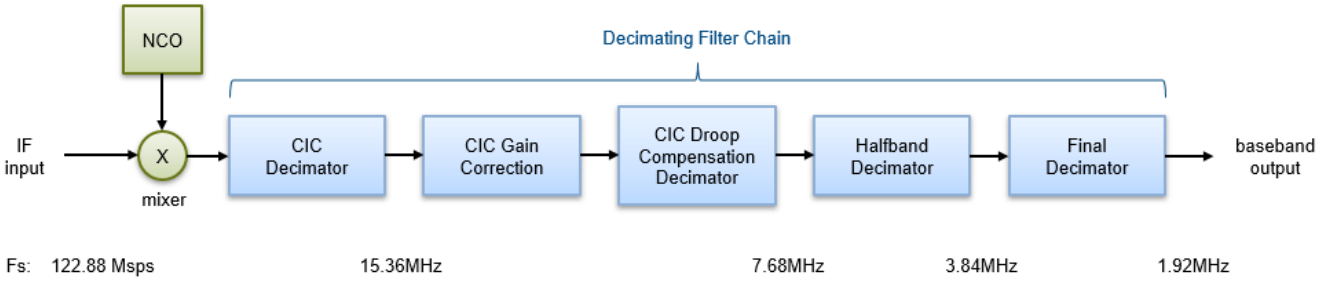
DDC Structure

The DDC consists of a numerically controlled oscillator (NCO), mixer, and decimating filter chain. The filter chain consists of a cascade integrator-comb (CIC) decimator, CIC gain correction, a CIC compensation decimator (FIR), a halfband FIR decimator, and a final FIR decimator.

The overall response of the filter chain is equivalent to that of a single decimation filter with the same specification. However, splitting the filter into multiple decimation stages results in a more efficient design that uses fewer hardware resources.

The CIC decimator provides a large initial decimation factor, which enables subsequent filters to work at lower rates. The CIC compensation decimator improves the spectral response by compensating for the CIC droop while decimating by two. The halfband is an intermediate decimator, and the final decimator implements the precise `Fpass` and `Fstop` characteristics of the DDC. The lower sampling rates near the end of the chain mean the later filters can optimize resource use by sharing multipliers.

This figure shows a block diagram of the DDC.



The sample rate of the input to the DDC is 122.88 Msps, and the output sample rate is 1.92 Msps. These rates give an overall decimation factor of 64. LTE receivers use 1.92 Msps as the typical sampling rate for cell search and master information block (MIB) recovery. The DDC filters are designed to suit this application. The DDC is optimized to run at a clock rate of 122.88 MHz.

DDC Design

This section explains how to design the DDC using floating-point operations and filter-design functions in MATLAB®.

DDC Parameters

This example designs the DDC filter characteristics to meet these specifications for the given input sampling rate and carrier frequency.

```
FsIn = 122.88e6;    % Sampling rate of DDC input
FsOut = 1.92e6;     % Sampling rate of DDC output
Fc = 32e6;          % Carrier frequency
Fpass = 540e3;      % Passband frequency, equivalent to 36x15kHz LTE subcarriers
Fstop = 700e3;      % Stopband frequency
Ap = 0.1;           % Passband ripple
Ast = 60;           % Stopband attenuation
```

CIC Decimator

The first filter stage is a CIC decimator because of its ability to efficiently implement a large decimation factor. The response of a CIC filter is similar to a cascade of moving average filters, but a CIC filter uses no multiplication or division. As a result, the CIC filter has a large DC gain.

```
cicParams.DecimationFactor = 8;
cicParams.DifferentialDelay = 1;
cicParams.NumSections = 3;
cicParams.FsOut = FsIn/cicParams.DecimationFactor;

cicFilt = dsp.CICDecimator(cicParams.DecimationFactor, ...
    cicParams.DifferentialDelay,cicParams.NumSections)       %#ok<*NOPTS>
cicGain = gain(cicFilt)


cicFilt =

  dsp.CICDecimator with properties:

      DecimationFactor: 8
```

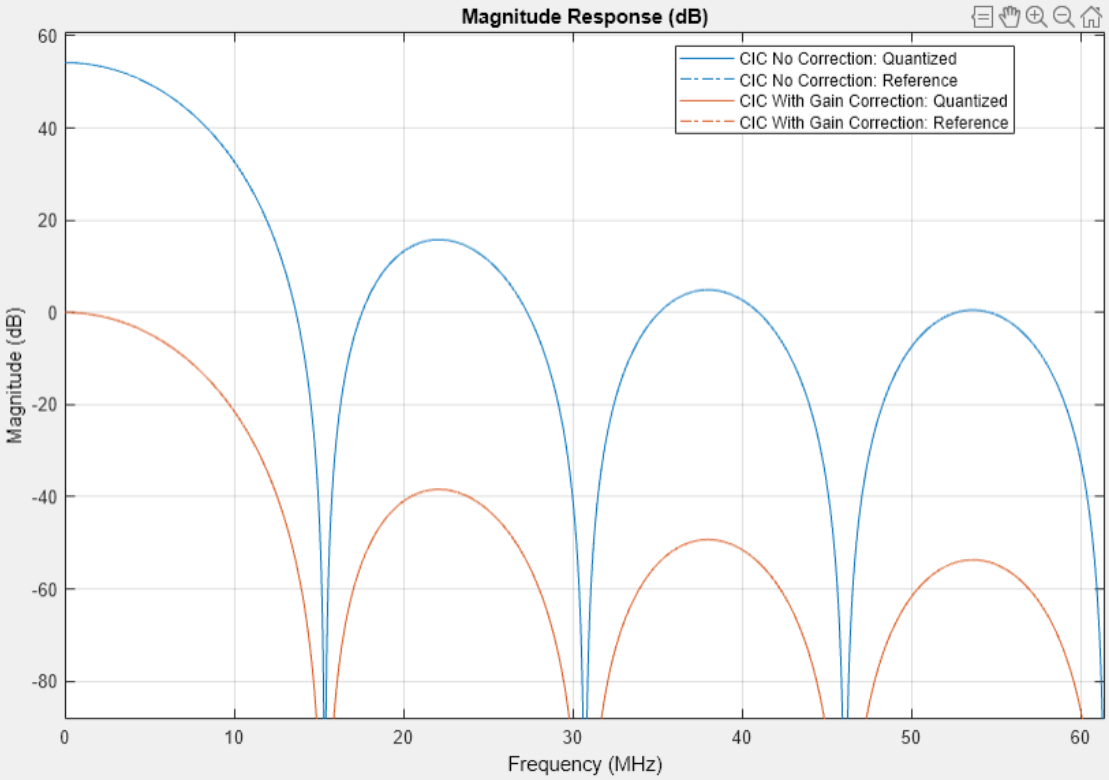
```
     DifferentialDelay: 1
           NumSections: 3
    FixedPointDataType: 'Full precision'


cicGain =

   512
```

Because the CIC gain is a power of two, a hardware implementation can easily correct for the gain factor by using a shift operation. For analysis purposes, the example represents the gain correction in MATLAB with a one-tap `dsp.FIRFilter` System object™.

```
cicGainCorr = dsp.FIRFilter('Numerator',1/cicGain)


cicGainCorr =

  dsp.FIRFilter with properties:

            Structure: 'Direct form'
      NumeratorSource: 'Property'
            Numerator: 0.0020
    InitialConditions: 0

  Use get to show all properties
```

Display the magnitude response of the CIC filter with and without gain correction by using `fvtool`. For analysis, combine the CIC filter and the gain correction filter into a `dsp.FilterCascade` System object. CIC filters use fixed-point arithmetic internally, so `fvtool` plots both the quantized and unquantized responses.

```
ddcPlots.cicDecim = fvtool(...
    cicFilt, ...
    dsp.FilterCascade(cicFilt,cicGainCorr), ...
    'Fs',[FsIn,FsIn]);
legend(ddcPlots.cicDecim, ...
    'CIC No Correction', ...
    'CIC With Gain Correction');
```



CIC Droop Compensation Filter

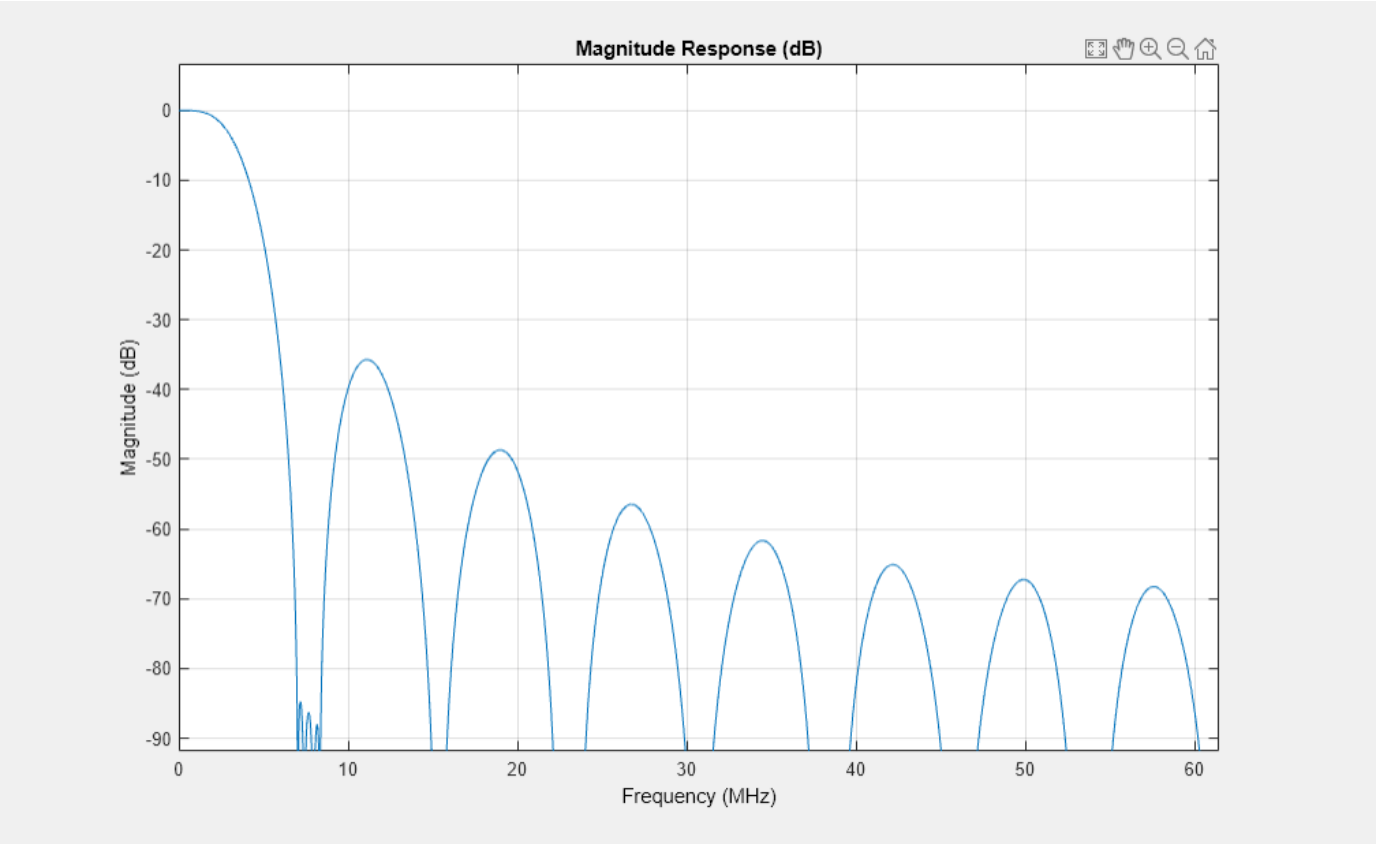
Because the magnitude response of the CIC filter has a significant *droop* within the passband region, the example uses a FIR-based droop compensation filter to flatten the passband response. The droop compensator has the same properties as the CIC decimator. This filter implements decimation by a factor of two, so you must also specify bandlimiting characteristics for the filter. Use the `design` function to return a filter System object with the specified characteristics.

```
compParams.R = 2;                                % CIC compensation decimation factor
compParams.Fpass = Fstop;                        % CIC compensation passband frequency
compParams.FsOut = cicParams.FsOut/compParams.R; % New sampling rate
compParams.Fstop = compParams.FsOut - Fstop;     % CIC compensation stopband frequency
compParams.Ap = Ap;                              % Same passband ripple as overall filter
compParams.Ast = Ast;                            % Same stopband attenuation as overall filter

compSpec = fdesign.decimator(compParams.R,'ciccomp', ...
    cicParams.DifferentialDelay, ...
    cicParams.NumSections, ...
    cicParams.DecimationFactor, ...
    'Fp,Fst,Ap,Ast', ...
    compParams.Fpass,compParams.Fstop,compParams.Ap,compParams.Ast, ...
    cicParams.FsOut);
compFilt = design(compSpec,'SystemObject',true)


compFilt =
```

```
dsp.FIRDecimator with properties:

 Main
  DecimationFactor: 2
   NumeratorSource: 'Property'
         Numerator: [-0.0398 -0.0126 0.2901 0.5258 0.2901 -0.0126 -0.0398]
         Structure: 'Direct form'

Use get to show all properties
```

Plot the combined response of the CIC filter (with gain correction) and droop compensation.

```
ddcPlots.cicComp = fvtool(...
    dsp.FilterCascade(cicFilt,cicGainCorr,compFilt), ...
    'Fs',FsIn,'Legend','off');
```



Halfband Decimator

The halfband filter provides efficient decimation by two. Halfband filters are efficient because approximately half of their coefficients are equal to zero, and those multipliers are excluded from the hardware implementation.

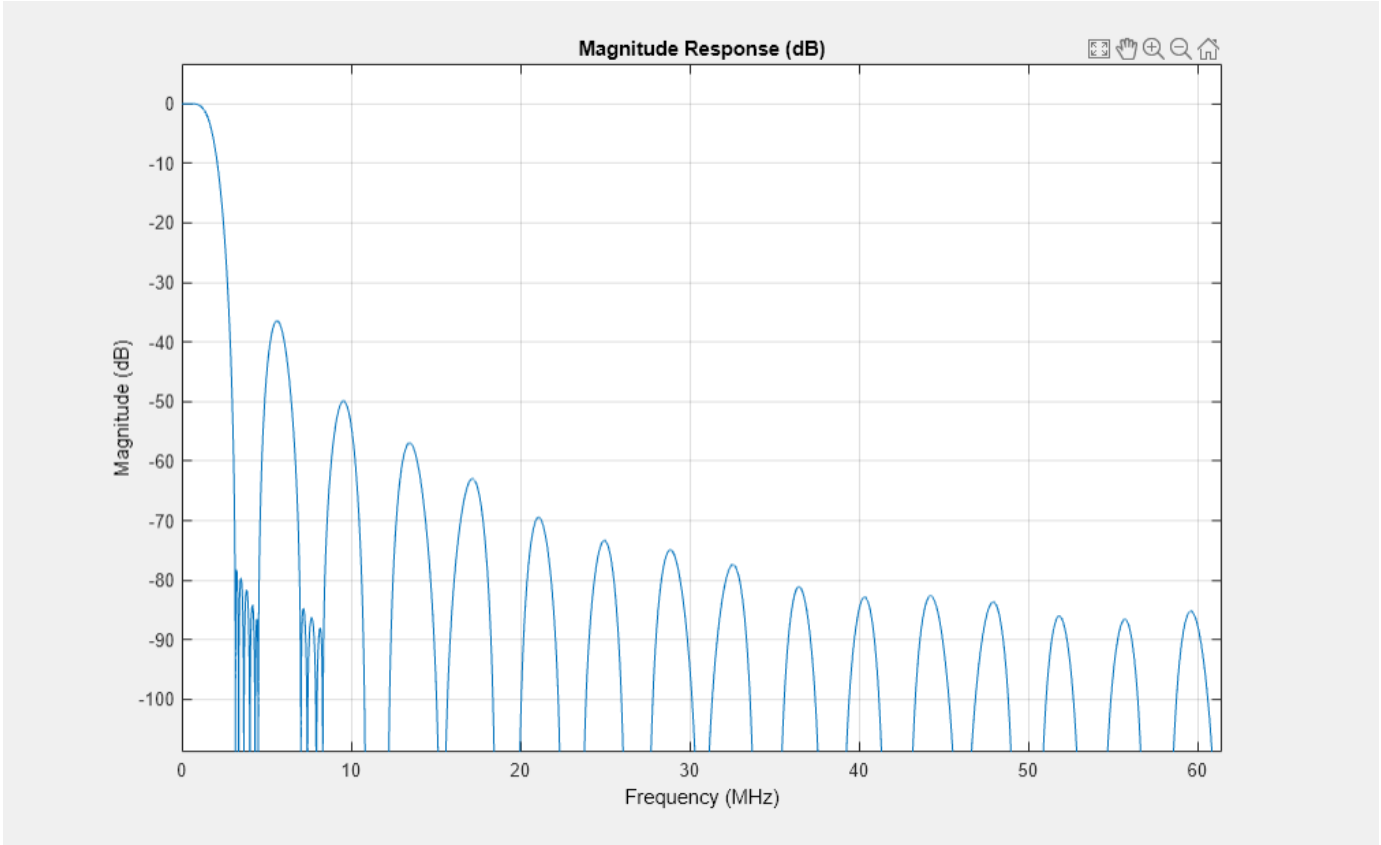
```
hbParams.FsOut = compParams.FsOut/2;
hbParams.TransitionWidth = hbParams.FsOut - 2*Fstop;
hbParams.StopbandAttenuation = Ast;

hbSpec = fdesign.decimator(2,'halfband',...
```

```
    'Tw,Ast', ...
    hbParams.TransitionWidth, ...
    hbParams.StopbandAttenuation, ...
    compParams.FsOut);
hbFilt = design(hbSpec,'SystemObject',true)
```

```
hbFilt =

  dsp.FIRDecimator with properties:

   Main
    DecimationFactor: 2
     NumeratorSource: 'Property'
           Numerator: [0.0089 0 -0.0565 0 0.2977 0.5000 0.2977 0 -0.0565 ... ]
           Structure: 'Direct form'

  Use get to show all properties
```

Plot the response of the DDC up to the halfband filter output.

```
ddcPlots.halfbandFIR = fvtool(...
    dsp.FilterCascade(cicFilt,cicGainCorr,compFilt,hbFilt), ...
    'Fs',FsIn,'Legend','off');
```



Final FIR Decimator

The final FIR implements the detailed passband and stopband characteristics of the DDC. This filter has more coefficients than the earlier FIR filters, but because it operates at a lower sampling rate it can use resource sharing for an efficient hardware implementation.

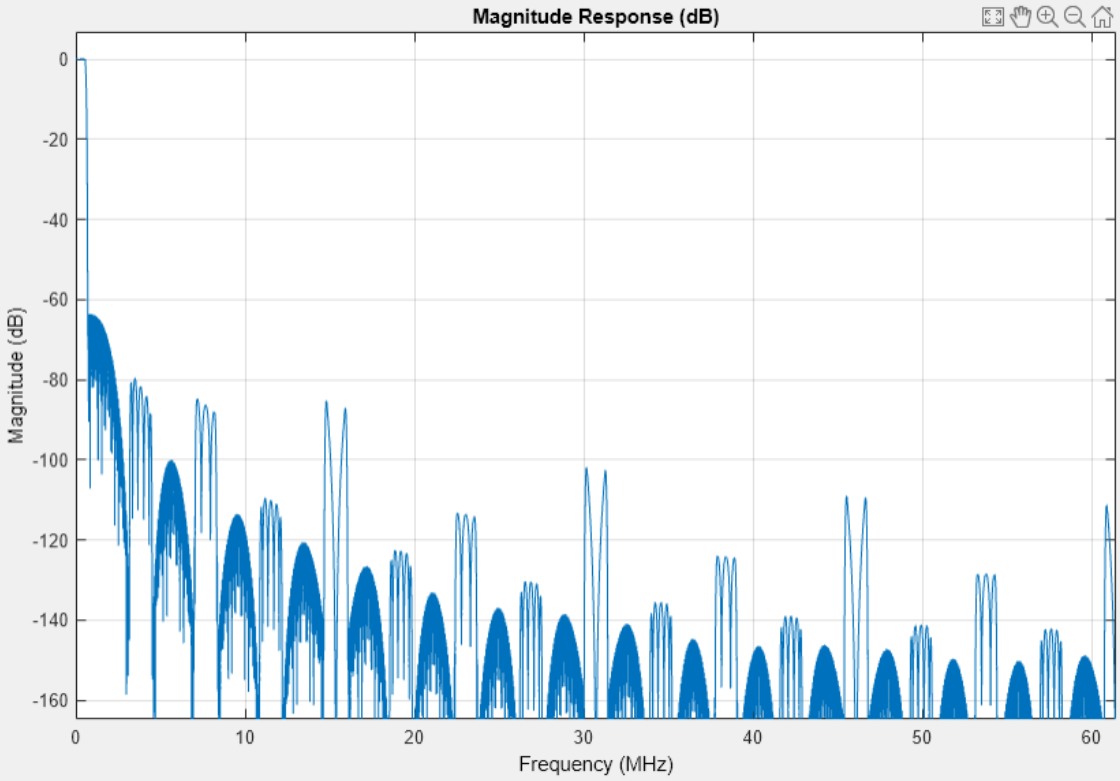
Add 3 dB of headroom to the stopband attenuation so that the DDC still meets the specification after fixed-point quantization. This value was found empirically by using `fvtool`.

```
finalSpec = fdesign.decimator(2,'lowpass', ...
    'Fp,Fst,Ap,Ast',Fpass,Fstop,Ap,Ast+3,hbParams.FsOut);
finalFilt = design(finalSpec,'equiripple','SystemObject',true)
```

```
finalFilt =

  dsp.FIRDecimator with properties:

   Main
    DecimationFactor: 2
     NumeratorSource: 'Property'
           Numerator: [9.3365e-04 0.0013 9.3466e-04 -5.3189e-04 -0.0022 ... ]
           Structure: 'Direct form'

  Use get to show all properties
```

Visualize the overall magnitude response of the DDC.

```
ddcFilterChain           = dsp.FilterCascade(cicFilt,cicGainCorr,compFilt,hbFilt,finalFilt);
ddcPlots.overallResponse = fvtool(ddcFilterChain,'Fs',FsIn,'Legend','off');
```



Fixed-Point Conversion

The frequency response of the floating-point DDC filter chain now meets the specification. Next, quantize each filter stage to use fixed-point types and analyze them to confirm that the filter chain still meets the specification.

Filter Quantization

This example uses 16-bit coefficients, which are sufficient to meet the specification. Using fewer than 18 bits for the coefficients minimizes the number of DSP blocks that are required for an FPGA implementation. The input to the DDC filter chain is 16-bit data with 15 fractional bits. The filter outputs are 18-bit values, which provide extra headroom and precision in the intermediate signals.

For the CIC decimator, choosing the `'Minimum section word lengths'` fixed-point data type option automatically optimizes the internal wordlengths based on the output wordlength and other CIC parameters.

```
cicFilt.FixedPointDataType = 'Minimum section word lengths';
cicFilt.OutputWordLength = 18;
```

Configure the fixed-point properties of the gain correction and FIR-based System objects. The object uses the default `RoundingMethod` and `OverflowAction` property values (`'Floor'` and `'Wrap'` respectively).

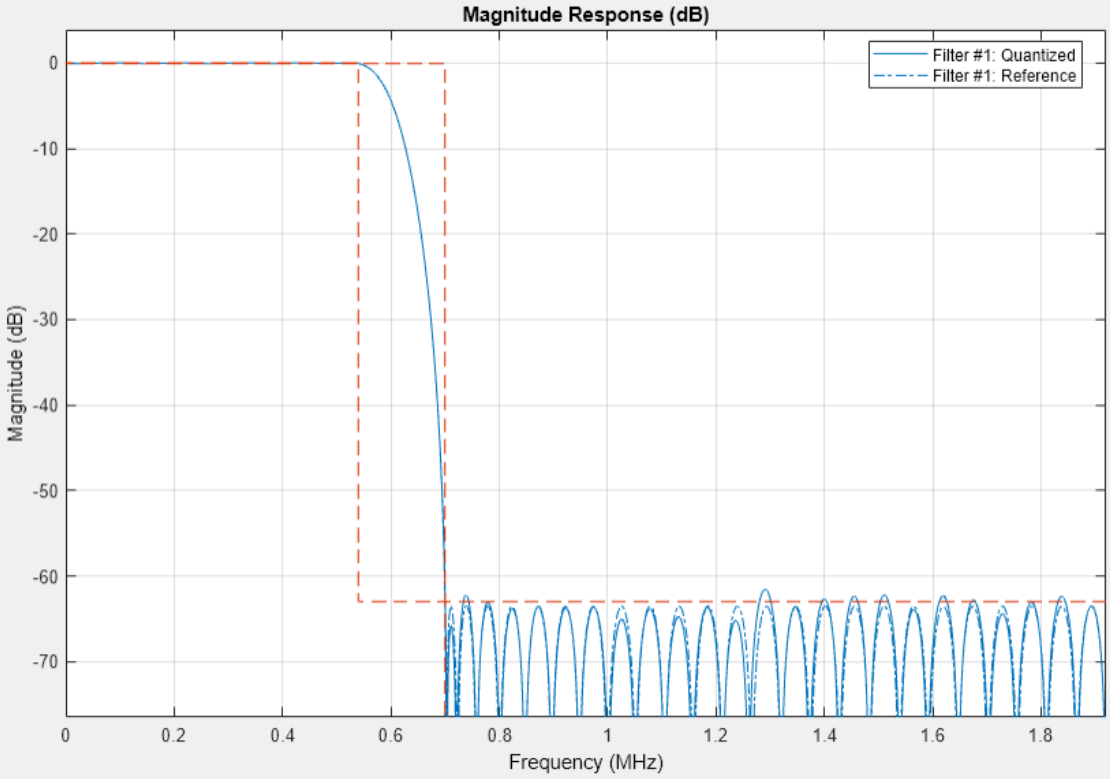
```
% CIC Gain Correction
cicGainCorr.FullPrecisionOverride = false;
cicGainCorr.CoefficientsDataType = 'Custom';
```

```
cicGainCorr.CustomCoefficientsDataType = numerictype(fi(cicGainCorr.Numerator,1,16));
cicGainCorr.OutputDataType = 'Custom';
cicGainCorr.CustomOutputDataType = numerictype(1,18,16);

% CIC Droop Compensation
compFilt.FullPrecisionOverride = false;
compFilt.CoefficientsDataType = 'Custom';
compFilt.CustomCoefficientsDataType = numerictype([],16,15);
compFilt.ProductDataType = 'Full precision';
compFilt.AccumulatorDataType = 'Full precision';
compFilt.OutputDataType = 'Custom';
compFilt.CustomOutputDataType = numerictype([],18,16);

% Halfband
hbFilt.FullPrecisionOverride = false;
hbFilt.CoefficientsDataType = 'Custom';
hbFilt.CustomCoefficientsDataType = numerictype([],16,15);
hbFilt.ProductDataType = 'Full precision';
hbFilt.AccumulatorDataType = 'Full precision';
hbFilt.OutputDataType = 'Custom';
hbFilt.CustomOutputDataType = numerictype([],18,16);

% FIR
finalFilt.FullPrecisionOverride = false;
finalFilt.CoefficientsDataType = 'Custom';
finalFilt.CustomCoefficientsDataType = numerictype([],16,15);
finalFilt.ProductDataType = 'Full precision';
finalFilt.AccumulatorDataType = 'Full precision';
finalFilt.OutputDataType = 'Custom';
finalFilt.CustomOutputDataType = numerictype([],18,16);
```

Fixed-Point Analysis

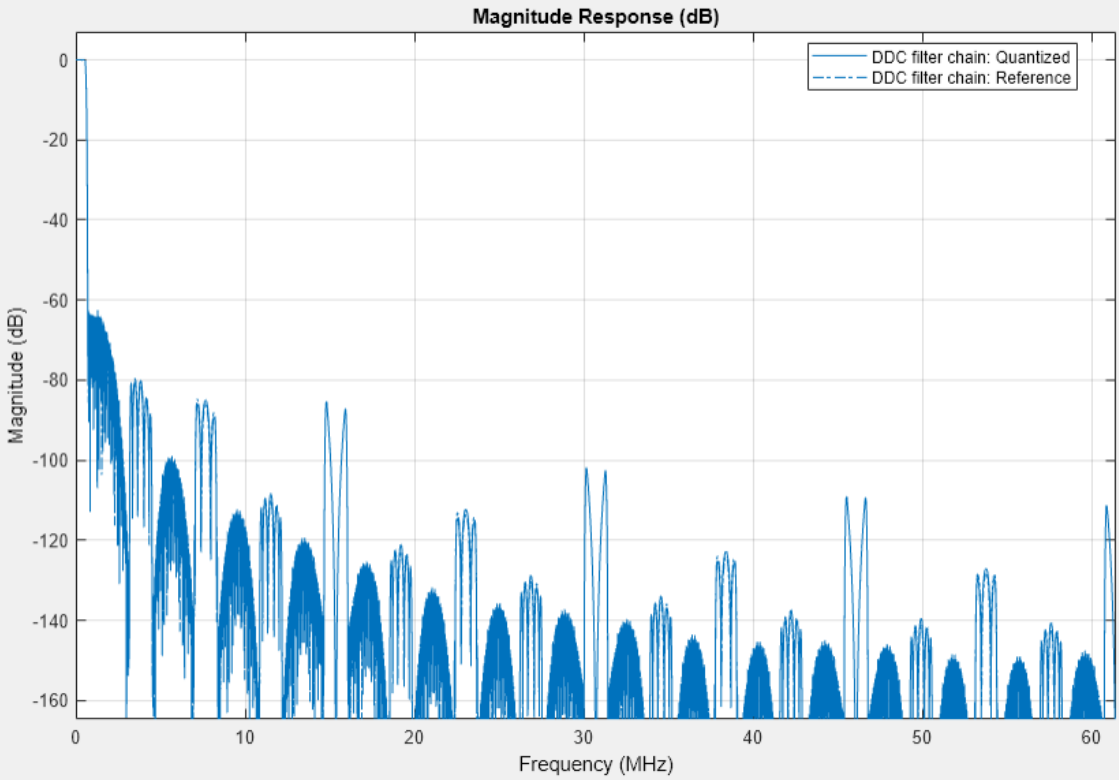
Inspect the quantization effects with `fvtool`. You can analyze the filters individually or in a cascade. `fvtool` shows the quantized and unquantized (reference) responses overlayed. For example, this figure shows the effect of quantizing the final FIR filter stage.

```
ddcPlots.quantizedFIR = fvtool(finalFilt, ...
    'Fs',hbParams.FsOut,'arithmetic','fixed');
```



Redefine the `ddcFilterChain` cascade object to include the fixed-point properties of the individual filters. Then, use `fvtool` to analyze the entire filter chain and confirm that the quantized DDC still meets the specification.

```
ddcFilterChain = dsp.FilterCascade(cicFilt, ...
    cicGainCorr,compFilt,hbFilt,finalFilt);
ddcPlots.quantizedDDCResponse = fvtool(ddcFilterChain, ...
    'Fs',FsIn,'Arithmetic','fixed');

legend(ddcPlots.quantizedDDCResponse, ...
    'DDC filter chain');
```



HDL-Optimized Simulink Model

The next step in the design flow is to implement the DDC in Simulink using blocks that support HDL code generation.

Model Configuration

The model relies on variables in the MATLAB workspace to configure the blocks and settings. It uses the same filter chain variables defined earlier in the example. Next, define the NCO characteristics and the input signal. The example uses these characteristics to configure the NCO block.

Specify the desired frequency resolution and calculate the number of accumulator bits that are required to achieve the desired resolution. Set the desired spurious free dynamic range, and then define the number of quantized accumulator bits. The NCO uses the quantized output of the accumulator to address the sine lookup table. Also compute the phase increment that the NCO uses to generate the specified carrier frequency. The NCO applies phase dither to those accumulator bits that are removed during quantization.

```
nco.Fd = 1;
nco.AccWL =  nextpow2(FsIn/nco.Fd)+1;
SFDR  = 84;
nco.QuantAccWL = ceil((SFDR-12)/6);
nco.PhaseInc = round((-Fc*2^nco.AccWL)/FsIn);
nco.NumDitherBits = nco.AccWL-nco.QuantAccWL;
```

The input to the DDC comes from the `ddcIn` variable. For now, assign a dummy value for `ddcIn` so that the model can compute its data types. During testing, `ddcIn` provides input data to the model.

```
ddcIn = 0; %#ok<NASGU>
```

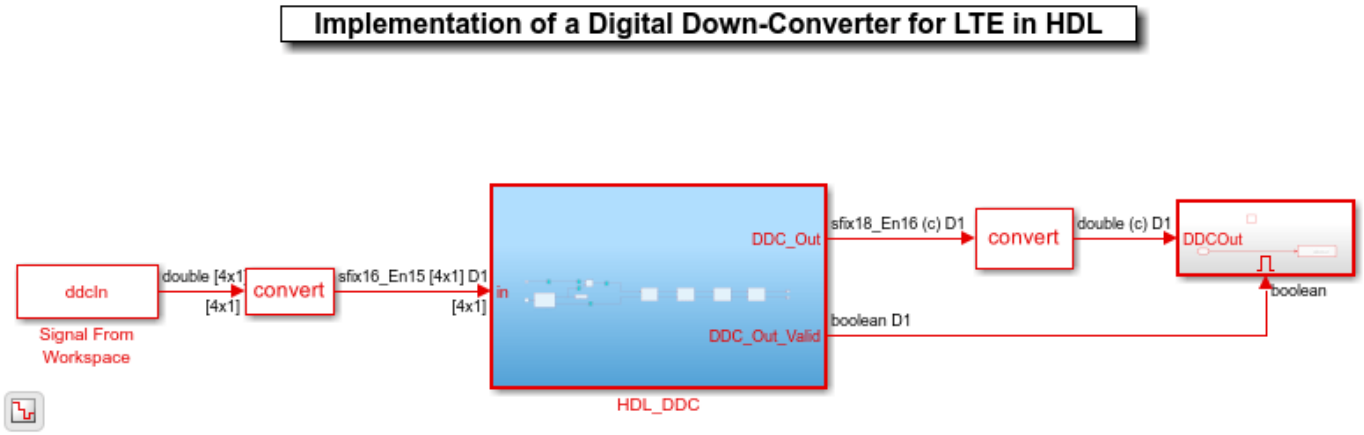
You can create a sample-based signal by setting up the `FrameSize` to 1, and output each individual sample as it is received. For a higher input sampling frequency or power reducing consideration, this design could also realize frame-based processing, and the `FrameSize` should be modified accordingly. In this case, we're showing a case for the `FrameSize` of 4.

```
FrameSize = 4;
```

Model Structure

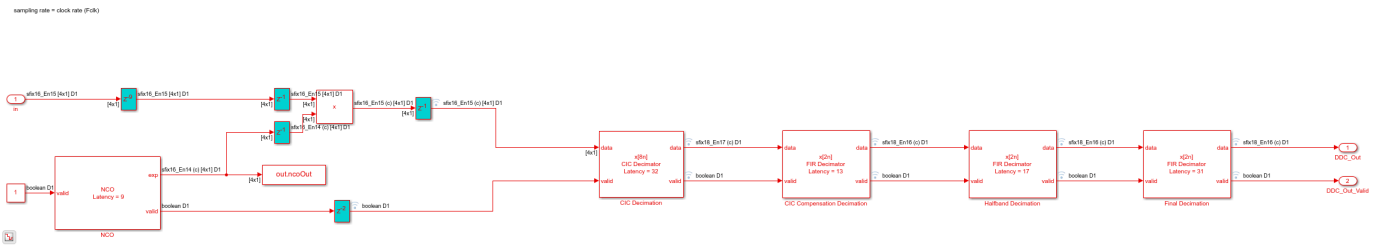
This figure shows the top level of the DDC Simulink model. The model imports the `ddcIn` variable from the MATLAB workspace by using a Signal From Workspace block, converts the input signal to 16-bit values, and applies the signal to the DDC. You can generate HDL code from the `HDL_DDC` subsystem.

```
modelName = 'DDCforLTEHDL';
open_system(modelName);
set_param(modelName,'SimulationCommand','Update');
set_param(modelName,'Open','on');
```



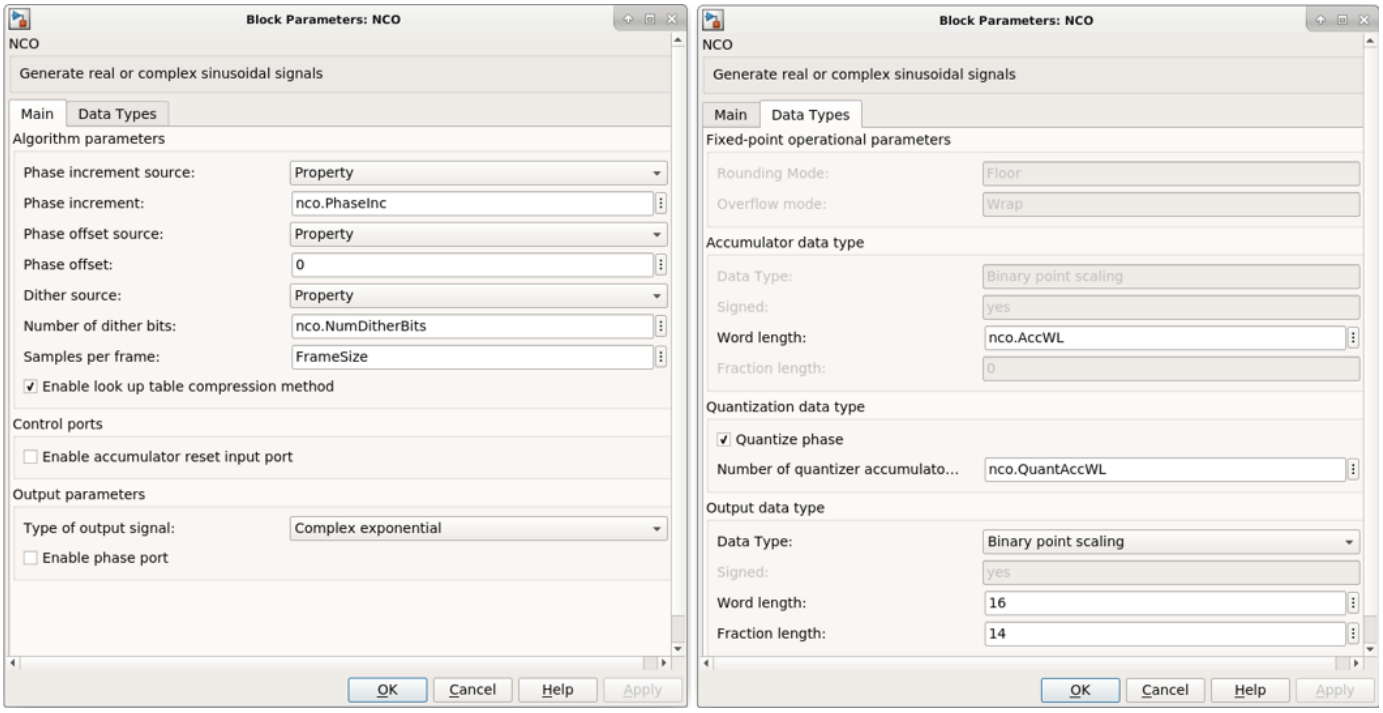
The `HDL_DDC` subsystem implements the DDC filter. First, the NCO block generates a complex phasor at the carrier frequency. This signal goes to a mixer that multiplies the phasor with the input signal. Then, the output of the mixer is passed to the filter chain and decimated to 1.92 Msps.

```
set_param([modelName '/HDL_DDC'],'Open','on');
```



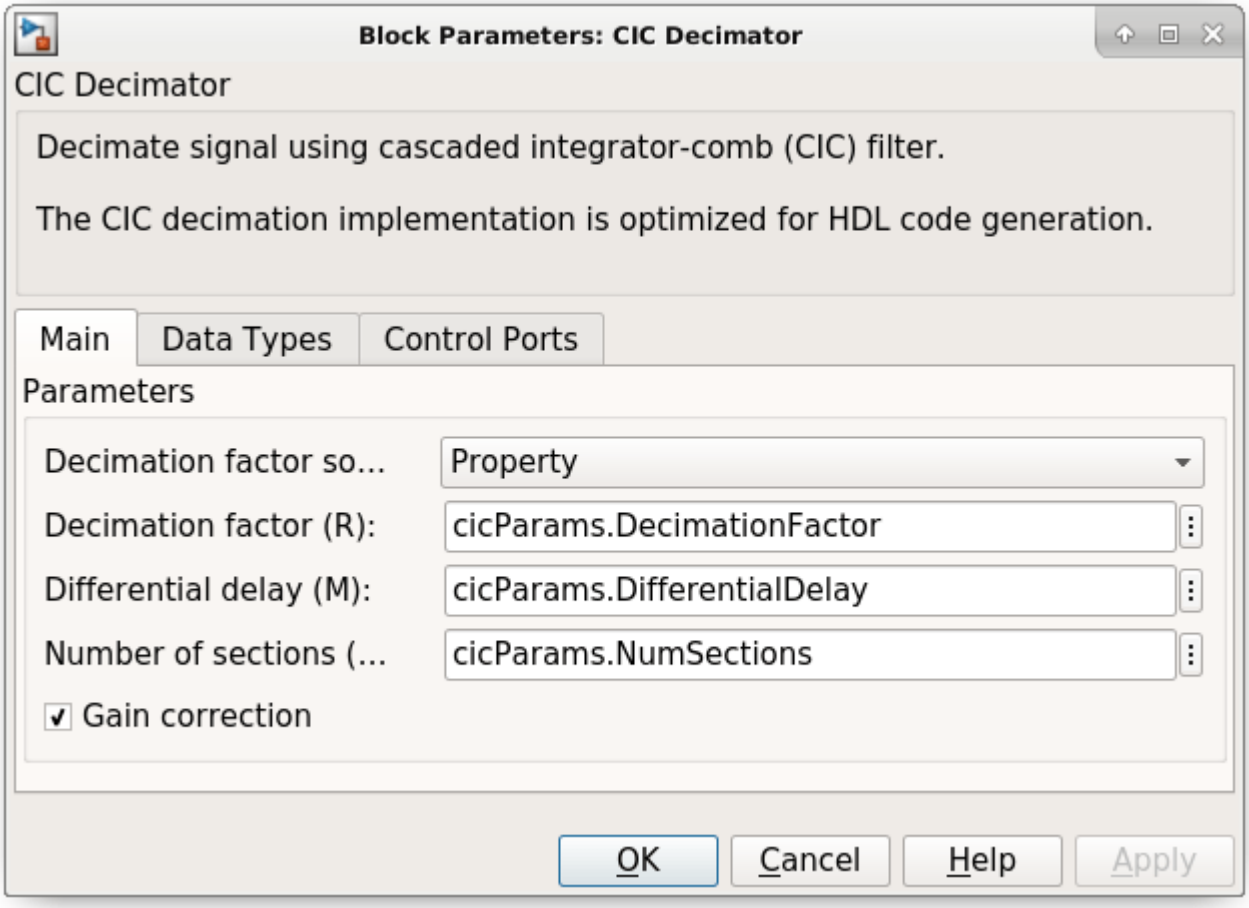
NCO Block Parameters

The NCO block in the model is configured with the parameters defined in the `nco` structure. This figure shows both tabs of the NCO block parameters dialog.



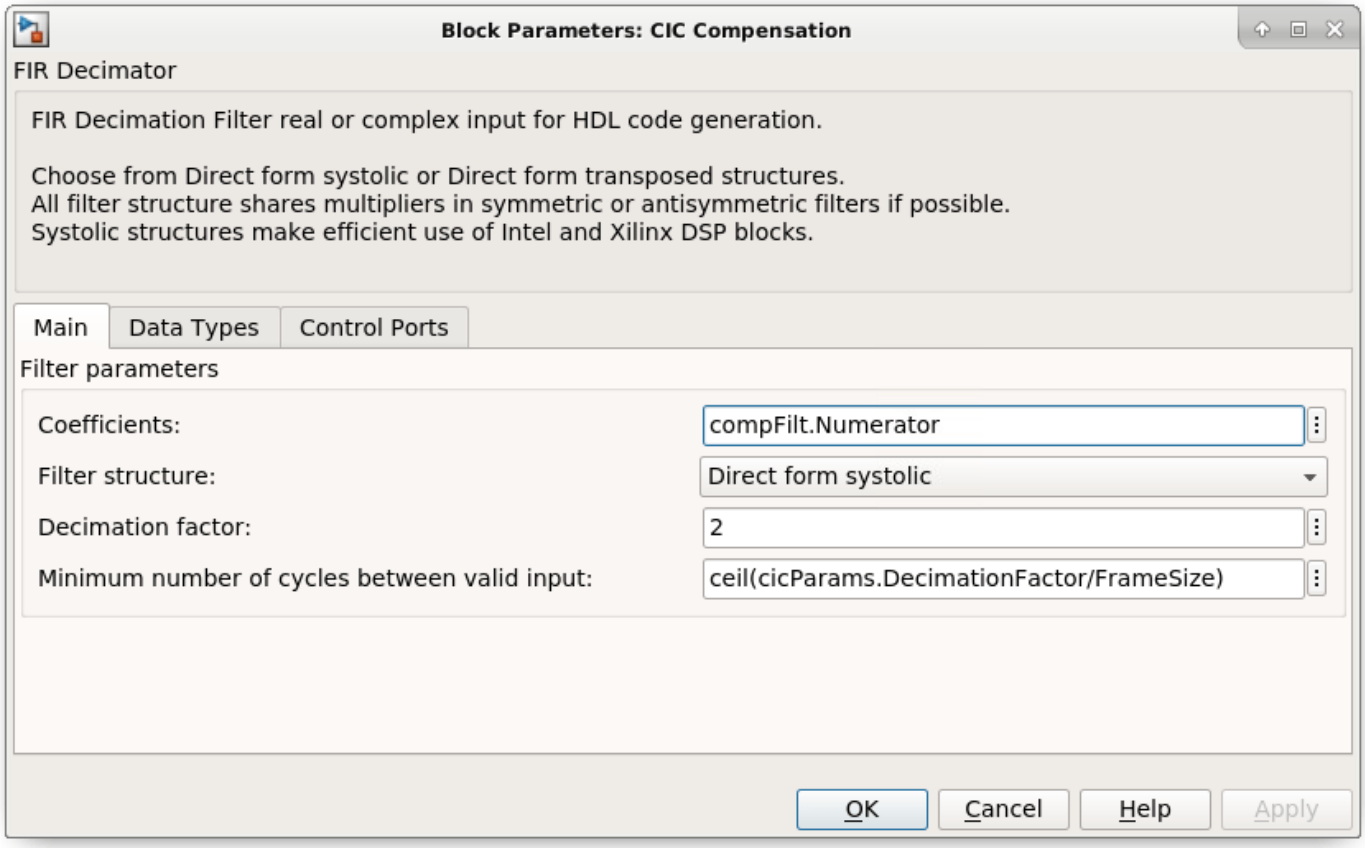
CIC Decimation and Gain Correction

The first filter stage is a CIC Decimator that is implemented with a CIC Decimator block. The block parameters are set to the `cicParams` structure values. To implement the gain correction, the model selects the **Gain correction** parameter. The image shows the block parameters for the CIC Decimator block.



The model configures the filters by using the properties of the corresponding System objects. The CIC compensation, halfband decimation, and final decimation filters operate at effective sample rates that are lower than the clock rate by factors of 8, 16, and 32, respectively. The model implements these sample rates by using the **valid** input signal to indicate which samples are valid at each rate. The signals in the filter chain all have the same Simulink sample time.

The CIC Compensation, Halfband Decimation, and Final Decimation filters are each implemented by an FIR Decimator. By setting the **Minimum number of cycles between valid input samples** parameter, we can use the invalid cycles between input samples. For example,the spacing between every input of CIC Compensation Decimator is 8, which equals the decimation factor. So the CIC Compensation Decimator has the **Minimum number of cycles between valid input samples** set to `ceil(cicParams.DecimationFactor/FrameSize)`, which equals 2 cycles. The image shows the block parameters for the CIC Compensation Decimation block.



The FIR Decimator block fully reuses the multipliers in time over the number of clock cycles you specify. For `FrameSize` is 4, the CIC Compensation Decimation filter with complex input data would use 4 multipliers. The Halfband Decimation uses 4 multipliers, and the Final Decimation uses 12 multipliers. For `FrameSize` is 1, since the inputs spacing of CIC Compensation Decimation and Halfband Decimation are larger than their filter length, those two decimators only require 2 multipliers. And the Final Decimation needs 4 multipliers at that time.

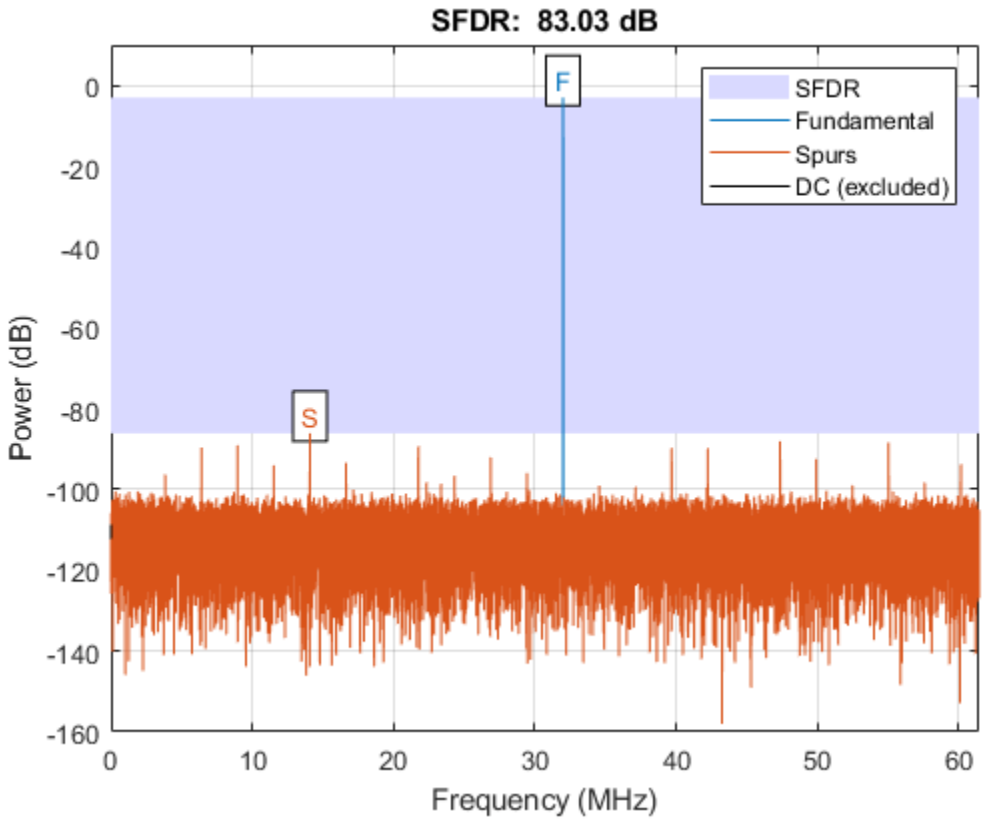
Sinusoid on Carrier Test and Verification

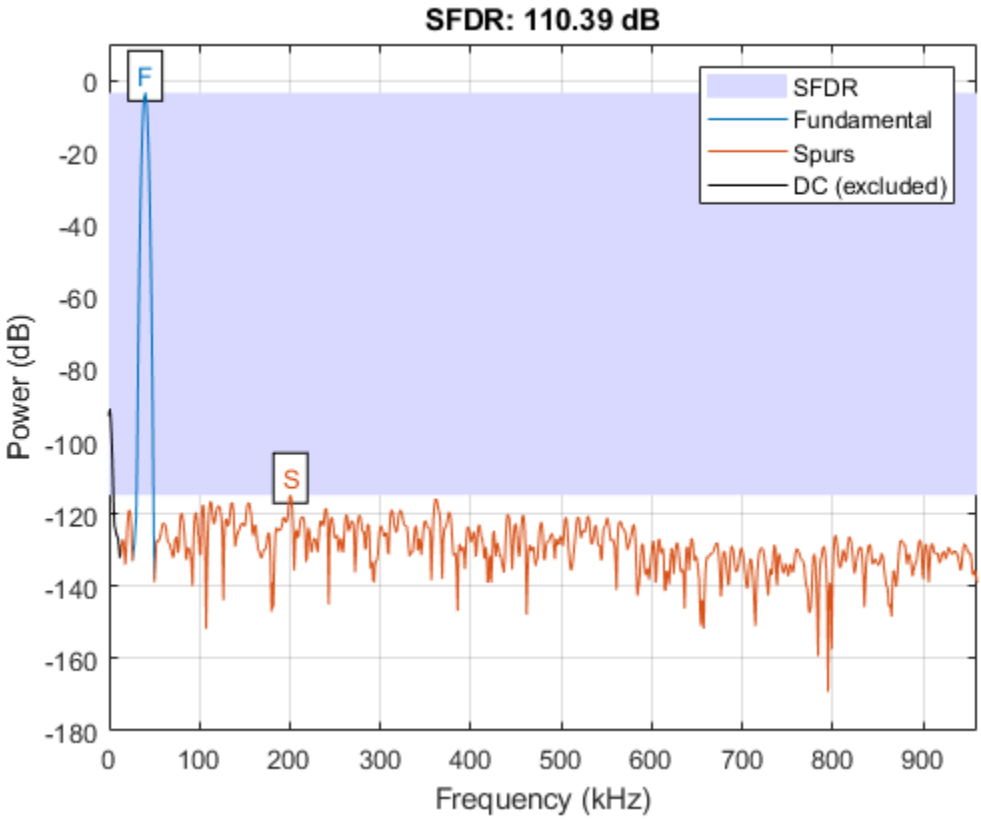
To test the DDC, modulate a 40 kHz sinusoid onto the carrier frequency and pass the modulated sine wave through the DDC. Then, measure the spurious- free dynamic range (SFDR) of the resulting tone and the SFDR of the NCO output. Plot the SFDR of the NCO and the fixed-point DDC output.

```
% Initialize random seed before executing any simulations.
rng(0);

% Generate a 40 kHz test tone, modulated onto the carrier.
ddcIn = DDCTestUtils.GenerateTestTone(40e3,Fc);

% Demodulate the test signal with the floating-point DDC.
ddcOut = DDCTestUtils.DownConvert(ddcIn,FsIn,Fc,ddcFilterChain);
release(ddcFilterChain);

% Demodulate the test signal by running the Simulink model.
out = sim(modelName);
```

```
% Measure the SFDR of the NCO, floating-point DDC outputs, and fixed-point
% DDC outputs.
results.sfdrNCO = sfdr(real(out.ncoOut),FsIn);
results.sfdrFloatDDC = sfdr(real(ddcOut),FsOut);
results.sfdrFixedDDC = sfdr(real(out.ddcFixedOut),FsOut);

disp('SFDR Measurements');
disp(['   Floating-point DDC SFDR: ',num2str(results.sfdrFloatDDC) ' dB']);
disp(['   Fixed-point NCO SFDR: ',num2str(results.sfdrNCO) ' dB']);
disp(['   Optimized fixed-point DDC SFDR: ',num2str(results.sfdrFixedDDC) ' dB']);
fprintf(newline);

% Plot the SFDR of the NCO and fixed-point DDC outputs.
ddcPlots.ncoOutSDFR = figure;
sfdr(real(out.ncoOut),FsIn);

ddcPlots.OptddcOutSFDR = figure;
sfdr(real(out.ddcFixedOut),FsOut);
```

```
SFDR Measurements
   Floating-point DDC SFDR: 291.4188 dB
   Fixed-point NCO SFDR: 83.0306 dB
   Optimized fixed-point DDC SFDR: 110.386 dB
```





LTE Signal Test

You can use an LTE test signal to perform more rigorous testing of the DDC. Generate a standard-compliant LTE waveform by using LTE Toolbox™ functions. Then, downconvert the waveform with the DDC model. Use LTE Toolbox functions to measure the error vector magnitude (EVM) of the resulting signals.

```
rng(0);
% Execute this test only if you have the LTE Toolbox product.
if license('test','LTE_Toolbox')

    % Generate a modulated LTE test signal by using the LTE Toolbox functions.
    [ddcIn,sigInfo] = DDCTestUtils.GenerateLTETestSignal(Fc);

    % Downconvert the signal with the floating-point DDC.
    ddcOut = DDCTestUtils.DownConvert(ddcIn,FsIn,Fc,ddcFilterChain);
    release(ddcFilterChain);

    % Downconvert the signal with the Simulink model, then measure and plot the
    % EVM of the floating-point and fixed-point results. Pad the input with zeros
    % to represent propagation latency and return the complete result.
    ddcIn = [ddcIn;zeros(2480*FrameSize,1)];
    out = sim(modelName);

    results.evmFloat = DDCTestUtils.MeasureEVM(sigInfo,ddcOut);
    results.evmFixed = DDCTestUtils.MeasureEVM(sigInfo,out.ddcFixedOut(1:length(ddcOut)));
```

```
    disp('LTE Error Vector Magnitude (EVM) Measurements');
    disp(['   Floating-point DDC RMS EVM: '  num2str(results.evmFloat.RMS*100,3) '%']);
    disp(['   Floating-point DDC Peak EVM: ' num2str(results.evmFloat.Peak*100,3) '%']);
    disp(['   Fixed-point DDC RMS EVM: '     num2str(results.evmFixed.RMS*100,3) '%']);
    disp(['   Fixed-point DDC Peak EVM: '    num2str(results.evmFixed.Peak*100,3) '%']);
    fprintf(newline);


end
```

```
LTE Error Vector Magnitude (EVM) Measurements
   Floating-point DDC RMS EVM: 0.633%
   Floating-point DDC Peak EVM: 2.44%
   Fixed-point DDC RMS EVM: 0.731%
   Fixed-point DDC Peak EVM: 2.69%
```

HDL Code Generation and FPGA Implementation

To generate the HDL code for this example you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `HDL_DDC` subsystem. The DDC was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place-and-route resource utilization results. The design met timing with a clock frequency of 331 MHz.

```
T = table(...
    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}),...
    categorical({'4341'; '383'; '8248'; '2.0'; '36'}),...
    'VariableNames',{'Resource','Usage'})
```

```
T =

  5x2 table

    Resource    Usage
    ________    _____

     LUT        4341
     LUTRAM     383
     FF         8248
     BRAM       2.0
     DSP        36
```

See Also

CIC Decimator | FIR Decimator | NCO

Related Examples

- “Implement Digital Upconverter for FPGA” on page 1-84

Implement Digital Upconverter for FPGA

This example shows how to design a digital upconverter (DUC) for radio communication applications such as LTE, and generate HDL code.

Introduction

DUCs are widely used in digital communication transmitters to convert baseband signals to radio frequency (RF) or intermediate frequency (IF) signals. The DUC operation increases the sample rate of the signal and shifts it to a higher frequency to facilitate subsequent processing stages. The DUC in this example performs sample rate conversion using a four-stage filter chain followed by complex frequency translation. The example starts by designing the DUC with DSP System Toolbox™ functions in floating point. Then, each stage is converted to fixed point, and used in a Simulink® model that generates synthesizable HDL code. The example uses these two test signals to demonstrate and verify the DUC operation:

- A sinusoid that is modulated onto a 32 MHz IF carrier.
- An LTE downlink signal with a bandwidth of 1.4 MHz, modulated onto a 32 MHz IF carrier.

The example downconverts the outputs of the floating-point and fixed-point DUCs, and compares the signal quality of the two outputs.

Finally, the example presents an implementation of the filter chain for FPGAs, and synthesis results.

This example uses `DUCTestUtils`, a helper class that contains functions for generating stimulus and analyzing the DUC output. For more information, see the `DUCTestUtils.m` file.

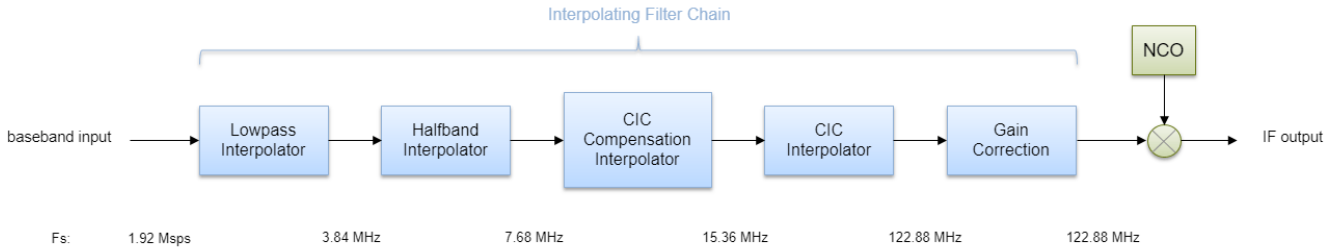
DUC Structure

The DUC consists of an interpolating filter chain, numerically controlled oscillator (NCO), and mixer. The filter chain consists of a lowpass interpolator, halfband interpolator, CIC compensation interpolator (FIR), CIC interpolator, and CIC gain correction.

The overall response of the filter chain is equivalent to that of a single interpolation filter with the same specification. However, splitting the filter into multiple interpolation stages results in a more efficient design that uses fewer hardware resources.

The first lowpass interpolator implements the precise `Fpass` and `Fstop` characteristics of the DUC. The halfband filter is an intermediate interpolator. The lower sampling rates at the beginning of the chain mean the earlier filters can optimize resource use by sharing multipliers. The CIC compensation interpolator improves the spectral response by compensating for the later CIC droop while interpolating by two. The CIC interpolator provides a large interpolation factor, which meets the filter chain upsampling requirements.

This figure shows a block diagram of the DUC.



The sample rate of the input to the DUC is 1.92 Msps, and the output sample rate is 122.88 Msps. These rates give an overall interpolation factor of 64. LTE receivers use 1.92 Msps as the typical sampling rate for cell search and master information block (MIB) recovery. The DUC filters are designed to suit this application. The DUC is optimized to run at a clock rate of 122.88 MHz.

DUC Design

This section explains how to design the DUC using floating-point operations and filter-design functions in MATLAB®. The DUC object enables you to specify several characteristics that define the response of the cascade for the four filters, including passband and stopband frequencies, passband ripple, and stopband attenuation.

DUC Parameters

This example designs the DUC filter characteristics to meet these desired response values for the given input sampling rate and carrier frequency.

```
FsIn = 1.92e6;      % Sampling rate at input to DUC
FsOut = 122.88;     % Sampling rate at the output
Fc = 32e6;          % Carrier frequency
Fpass = 540e3;      % Passband frequency, equivalent to 36*15kHz LTE subcarriers
Fstop = 700e3;      % Stopband frequency
Ap = 0.1;           % Passband ripple
Ast = 60;           % Stopband attenuation
```

First Lowpass Interpolator

This filter interpolates by two, and operates at the lowest sampling rate of the filter chain. The low sample rate means this filter can use resource sharing for an efficient hardware implementation.

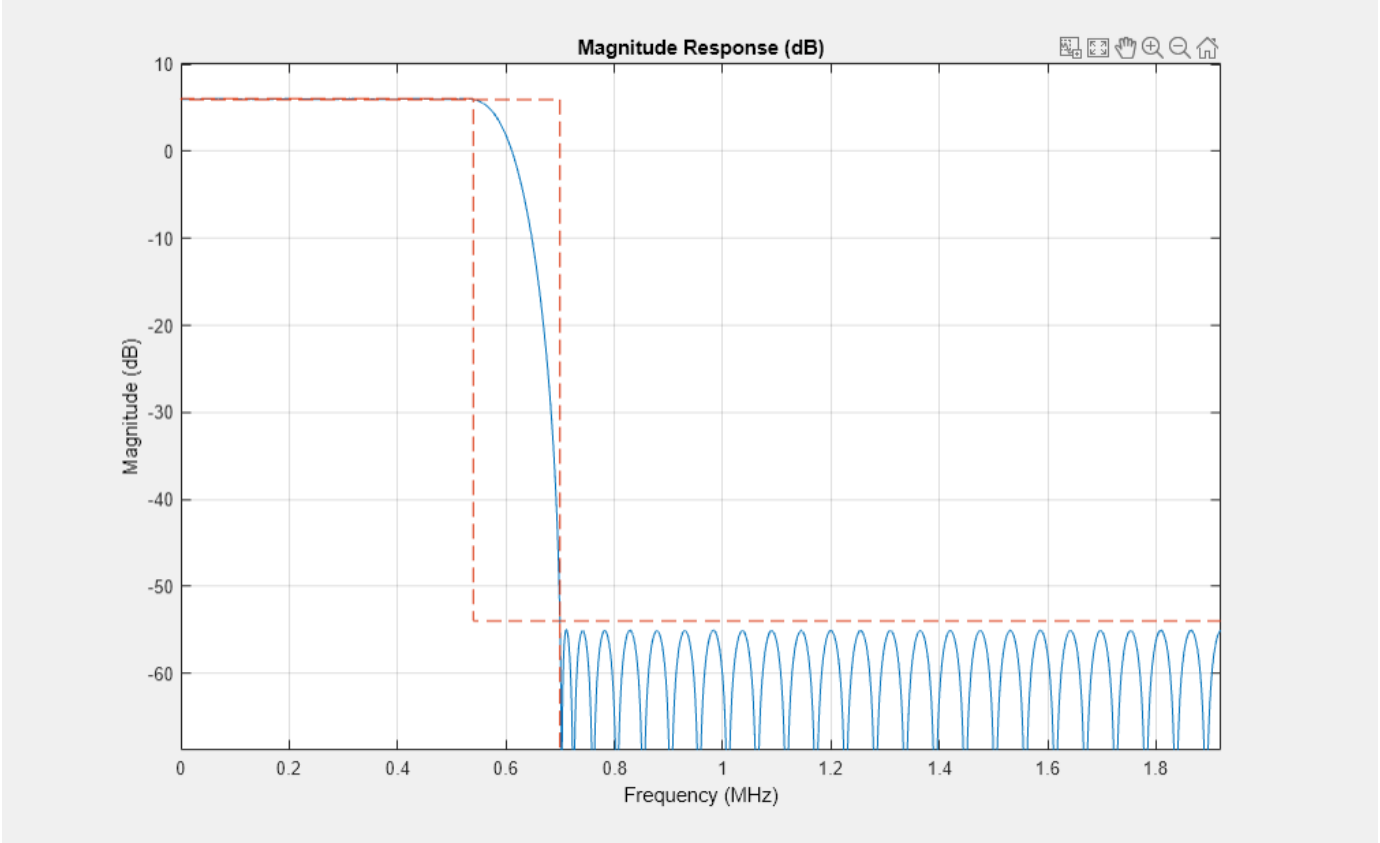
```
lowpassParams.FsIn = FsIn;
lowpassParams.InterpolationFactor = 2;
lowpassParams.FsOut = FsIn*lowpassParams.InterpolationFactor;

lowpassSpec = fdesign.interpolator(lowpassParams.InterpolationFactor, ...
    'lowpass','Fp,Fst,Ap,Ast',Fpass,Fstop,Ap,Ast,lowpassParams.FsOut);
lowpassFilt = design(lowpassSpec,'SystemObject',true)
```

```
lowpassFilt =

  dsp.FIRInterpolator with properties:

    InterpolationFactor: 2
        NumeratorSource: 'Property'
              Numerator: [0.0020 0.0021 4.9115e-04 -0.0027 -0.0050 ... ]
```

```
  Use get to show all properties
```

Display the magnitude response of the lowpass filter without gain correction.

```
ducPlots.lowpass = fvtool(lowpassFilt,'Fs',FsIn*2,'Legend','off');
```



Second Halfband Interpolator

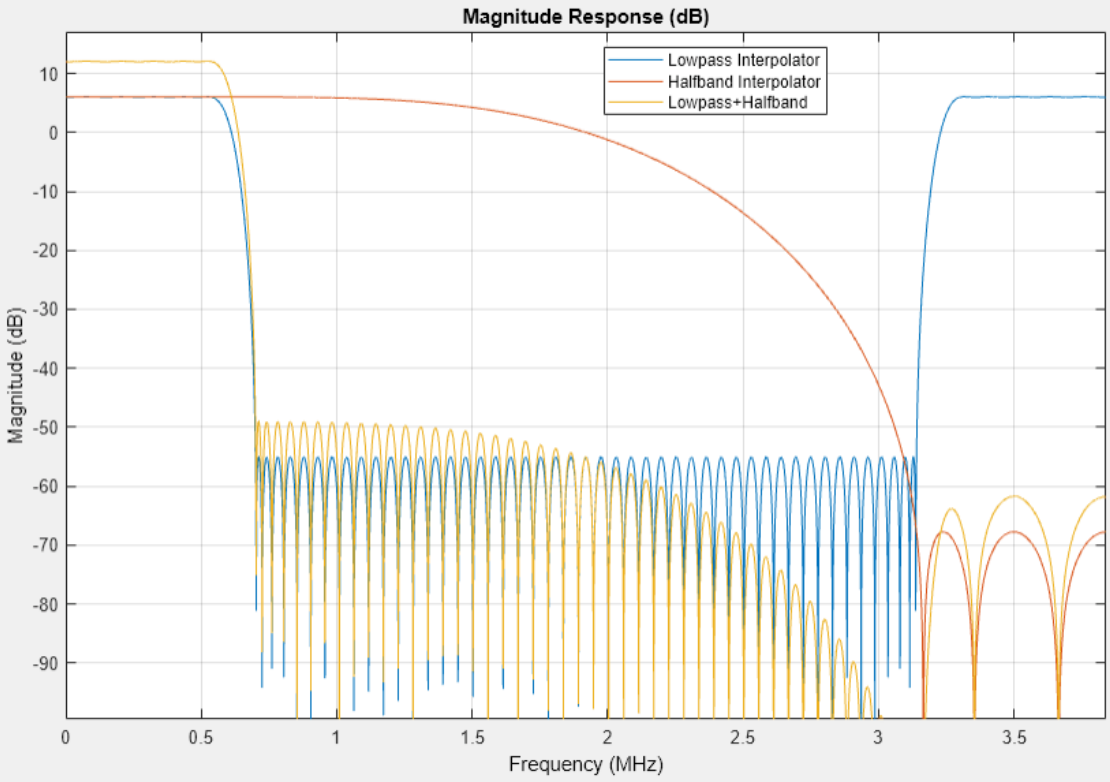
The halfband filter provides efficient interpolation by two. Halfband filters are efficient for hardware because approximately half of their coefficients are equal to zero, and those multipliers are excluded from the hardware implementation.

```
hbParams.FsIn = lowpassParams.FsOut;
hbParams.InterpolationFactor = 2;
hbParams.FsOut = lowpassParams.FsOut*hbParams.InterpolationFactor;
hbParams.TransitionWidth = hbParams.FsIn - 2*Fstop;
hbParams.StopbandAttenuation = Ast;

hbSpec = fdesign.interpolator(hbParams.InterpolationFactor,'halfband', ...
         'TW,Ast', ...
         hbParams.TransitionWidth, ...
         hbParams.StopbandAttenuation, ...
         hbParams.FsOut);

hbFilt = design(hbSpec,'SystemObject',true)
```

```
hbFilt =

  dsp.FIRInterpolator with properties:

    InterpolationFactor: 2
        NumeratorSource: 'Property'
              Numerator: [0.0178 0 -0.1129 0 0.5953 1 0.5953 0 -0.1129 0 ... ]

  Use get to show all properties
```

Visualize the magnitude response of the halfband interpolation.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt);
ducPlots.hbFilt = fvtool(lowpassFilt,hbFilt,ducFilterChain, ...
                         'Fs',[FsIn*2,FsIn*4,FsIn*4]);

legend(ducPlots.hbFilt, ...
       'Lowpass Interpolator', ...
       'Halfband Interpolator', ...
       'Lowpass+Halfband');
```



CIC Compensation Interpolator

Because the magnitude response of the last CIC filter has a significant *droop* within the passband region, the example uses an FIR-based droop compensation filter to flatten the passband response.

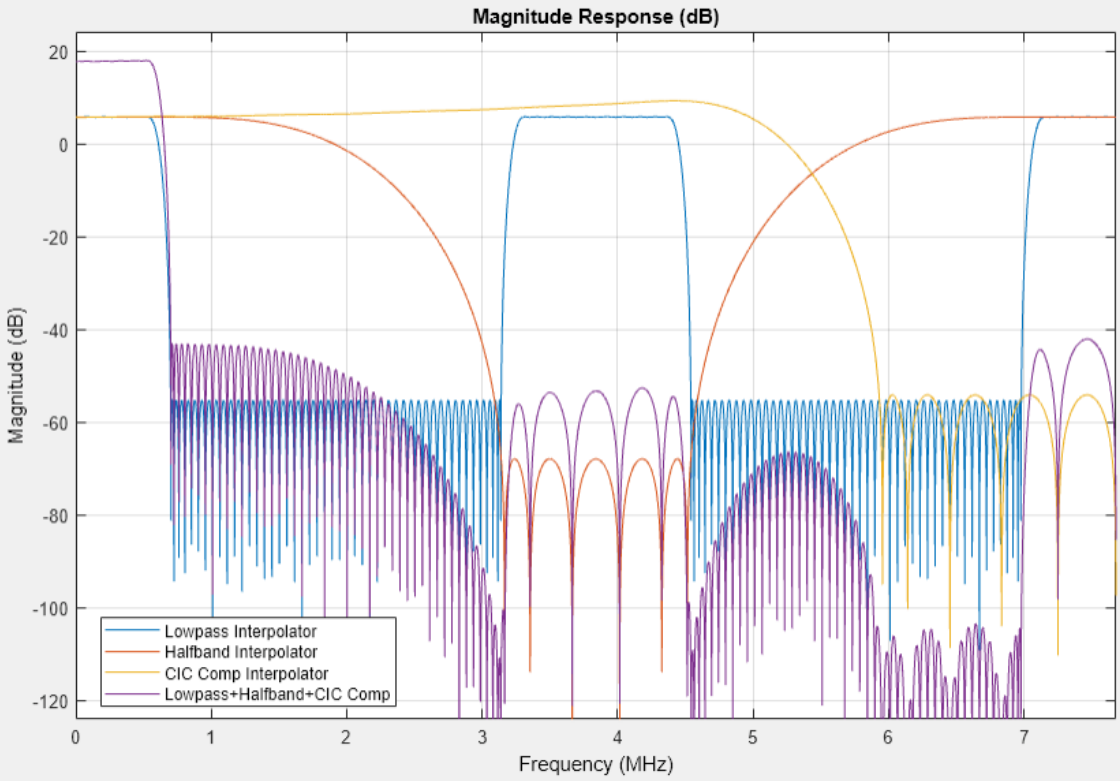
The droop compensator has the same properties as the CIC interpolator. This filter implements interpolation by a factor of two, so you must also specify bandlimiting characteristics for the filter. Also, specify the CIC interpolator properties that are used for this compensation filter as well as the later CIC interpolator.

Use the `design` function to return a filter System object with the specified characteristics.

```
compParams.FsIn = hbParams.FsOut;
compParams.InterpolationFactor = 2;                                    % CIC compensation interpolat
compParams.FsOut = compParams.FsIn*compParams.InterpolationFactor;  % New sampling rate
compParams.Fpass = 1/2*compParams.FsIn + Fpass;                        % CIC compensation passband f
compParams.Fstop = 1/2*compParams.FsIn + 1/4*compParams.FsIn;          % CIC compensation stopband f
compParams.Ap = Ap;                                                    % Same passband ripple as ove
compParams.Ast = Ast;                                                  % Same stopband attenuation a
N = 31;                                                                % 32 tap filter to take advan

cicParams.InterpolationFactor = 8;  % CIC interpolation factor
cicParams.DifferentialDelay = 1;    % CIC interpolator differential delay
cicParams.NumSections = 3;          % CIC interpolator number of integrator and comb sections

compSpec = fdesign.interpolator(compParams.InterpolationFactor,'ciccomp', ...
           cicParams.DifferentialDelay, ...
           cicParams.NumSections, ...
           cicParams.InterpolationFactor, ...
           'N,Fp,Ap,Ast', ...
           N,compParams.Fpass,compParams.Ap,compParams.Ast, ...
           compParams.FsOut);
compFilt = design(compSpec,'SystemObject',true)


compFilt =

  dsp.FIRInterpolator with properties:

    InterpolationFactor: 2
        NumeratorSource: 'Property'
              Numerator: [-6.9876e-04 -0.0099 0.0038 0.0134 -0.0255 ... ]

  Use get to show all properties
```

Plot the response of the CIC compensation interpolator.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt,compFilt);
ducPlots.cicComp = fvtool(lowpassFilt,hbFilt,compFilt,ducFilterChain, ...
                          'Fs',[FsIn*2,FsIn*4,FsIn*8,FsIn*8]);

legend(ducPlots.cicComp, ...
       'Lowpass Interpolator', ...
       'Halfband Interpolator', ...
       'CIC Comp Interpolator', ...
       'Lowpass+Halfband+CIC Comp');
```



CIC Interpolator

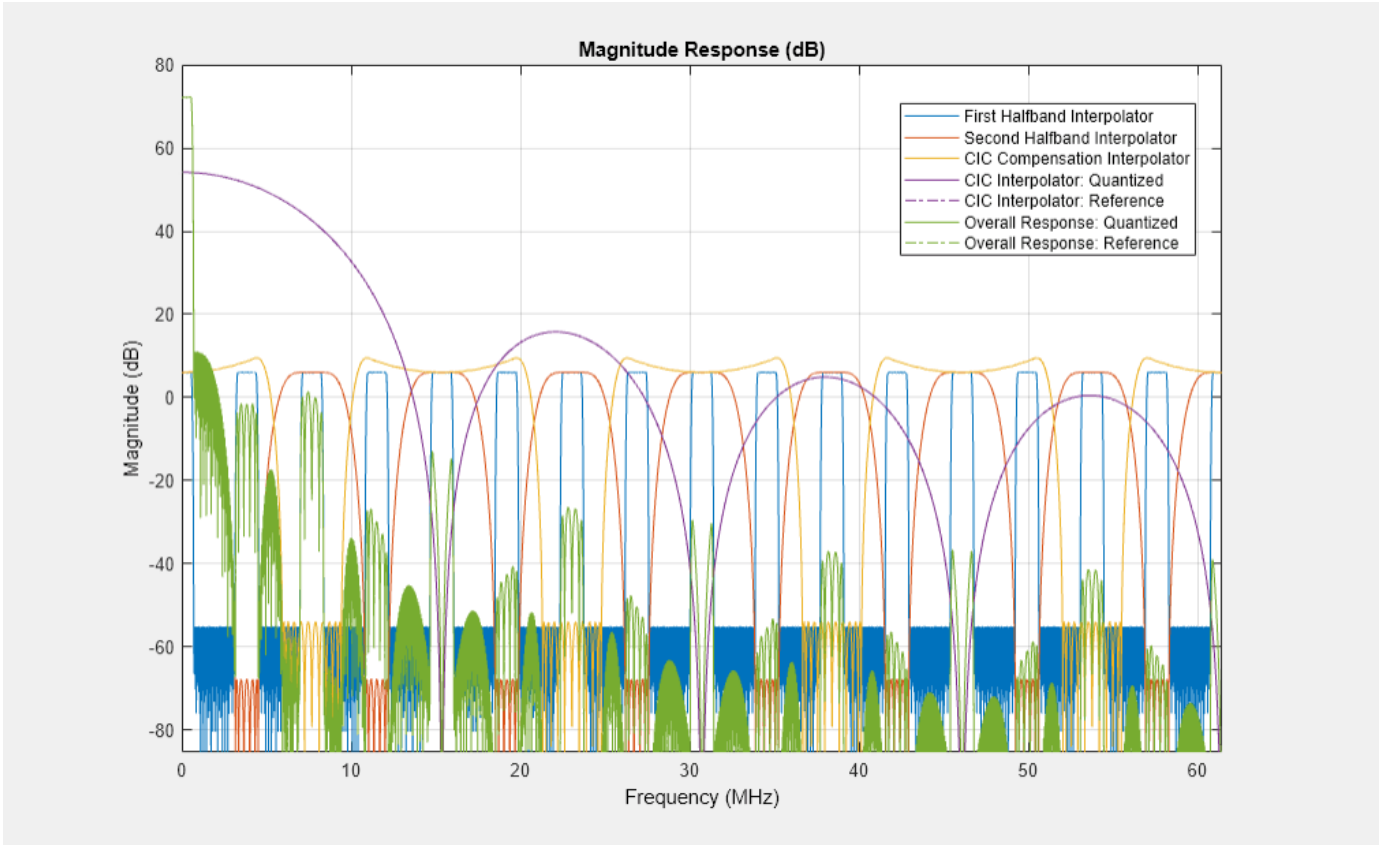
The last filter stage is implemented as a CIC interpolator because of this type of filter's ability to efficiently implement a large decimation factor. The response of a CIC filter is similar to a cascade of moving average filters, but a CIC filter uses no multiplication or division. As a result, the CIC filter has a large DC gain.

```
cicParams.FsIn = compParams.FsOut;
cicParams.FsOut = cicParams.FsIn*cicParams.InterpolationFactor;

cicFilt = dsp.CICInterpolator(cicParams.InterpolationFactor, ...
          cicParams.DifferentialDelay,cicParams.NumSections) %#ok<*NOPTS>
```

```
cicFilt =

  dsp.CICInterpolator with properties:

    InterpolationFactor: 8
      DifferentialDelay: 1
            NumSections: 3
     FixedPointDataType: 'Full precision'
```

Visualize the magnitude response of the CIC interpolation. CIC filters use fixed-point arithmetic internally, so `fvtool` plots both the quantized and unquantized responses.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt,compFilt,cicFilt);
ducPlots.cicInter = fvtool(lowpassFilt,hbFilt,compFilt,cicFilt,ducFilterChain, ...
                     'Fs',[FsIn*2,FsIn*4,FsIn*8,FsIn*64,FsIn*64]);

legend(ducPlots.cicInter, ...
       'First Halfband Interpolator', ...
       'Second Halfband Interpolator', ...
       'CIC Compensation Interpolator', ...
       'CIC Interpolator',...
       'Overall Response');
```



Every interpolator has a DC gain that is determined by its interpolation factor. The CIC interpolator has a larger gain than other filters. Call the `gain` function to get the gain factor of this filter.

Because the CIC gain is a power of two, a hardware implementation can easily correct for the gain factor by using a shift operation. For analysis purposes, the example represents the gain correction by using a one-tap `dsp.FIRFilter` System object. Combine the filter chain and the gain correction filter into a `dsp.FilterCascade` System object.

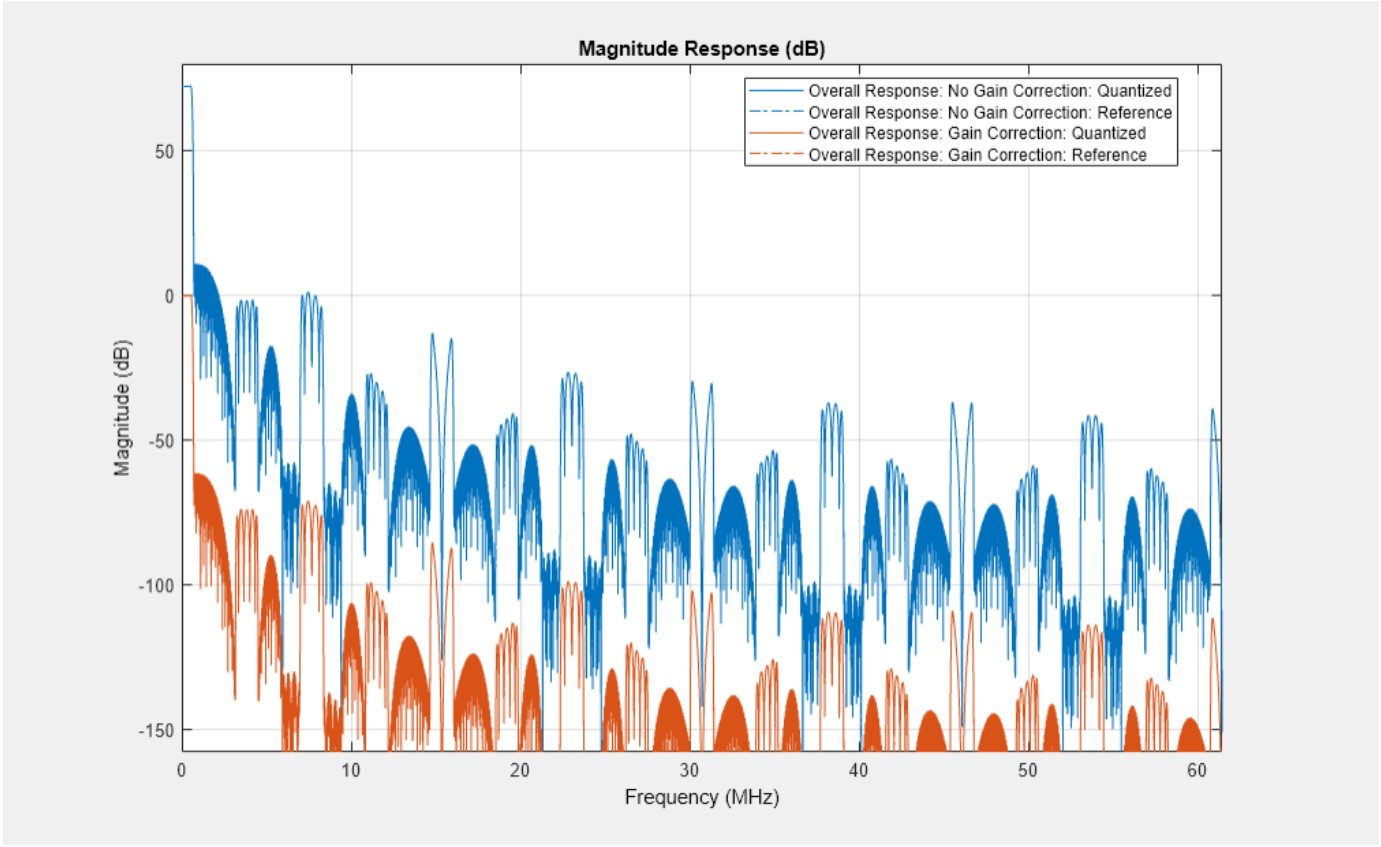
```
cicGain = gain(cicFilt)
Gain = lowpassParams.InterpolationFactor* ...
    hbParams.InterpolationFactor*compParams.InterpolationFactor* ...
    cicParams.InterpolationFactor*cicGain;
GainCorr = dsp.FIRFilter('Numerator',1/Gain)
```

```
cicGain =
```

```
    64


GainCorr =

  dsp.FIRFilter with properties:

           Structure: 'Direct form'
     NumeratorSource: 'Property'
           Numerator: 2.4414e-04
    InitialConditions: 0

  Use get to show all properties
```

Plot the overall chain response with and without gain correction.

```
ducPlots.overallResponse = fvtool(ducFilterChain,dsp.FilterCascade(ducFilterChain,GainCorr), ...
                           'Fs',[FsIn*64,FsIn*64]);
legend(ducPlots.overallResponse, ...
       'Overall Response: No Gain Correction',...
       'Overall Response: Gain Correction');
```



Fixed-Point Conversion

The frequency response of the floating-point DUC filter chain now meets the specification. Next, quantize each filter stage to use fixed-point types and analyze them to confirm that the filter chain still meets the specification.

Filter Quantization

This example uses 16-bit coefficients, which are sufficient to meet the specification. Using fewer than 18 bits for the coefficients minimizes the number of DSP blocks that are required for an FPGA implementation. The input to the DUC filter chain is 16-bit data with 15 fractional bits. The filter outputs are 18-bit values, which provide extra headroom and precision in the intermediate signals.

```
% First Lowpass Interpolator
lowpassFilt.FullPrecisionOverride = false;
lowpassFilt.CoefficientsDataType = 'Custom';
lowpassFilt.CustomCoefficientsDataType = numerictype([],16,15);
lowpassFilt.ProductDataType = 'Full precision';
lowpassFilt.AccumulatorDataType = 'Full precision';
lowpassFilt.OutputDataType = 'Custom';
lowpassFilt.CustomOutputDataType = numerictype([],18,14);

% Halfband
hbFilt.FullPrecisionOverride = false;
hbFilt.CoefficientsDataType = 'Custom';
hbFilt.CustomCoefficientsDataType = numerictype([],16,14);
hbFilt.ProductDataType = 'Full precision';
hbFilt.AccumulatorDataType = 'Full precision';
hbFilt.OutputDataType = 'Custom';
hbFilt.CustomOutputDataType = numerictype([],18,14);

% CIC Compensation Interpolator
compFilt.FullPrecisionOverride = false;
compFilt.CoefficientsDataType = 'Custom';
compFilt.CustomCoefficientsDataType = numerictype([],16,14);
compFilt.ProductDataType = 'Full precision';
compFilt.AccumulatorDataType = 'Full precision';
compFilt.OutputDataType = 'Custom';
compFilt.CustomOutputDataType = numerictype([],18,14);
```

For the CIC interpolator, choosing the `'Minimum section word lengths'` fixed-point data type option automatically optimizes the internal wordlengths based on the output wordlength and other CIC parameters.

```
cicFilt.FixedPointDataType = 'Minimum section word lengths';
cicFilt.OutputWordLength = 18;
```

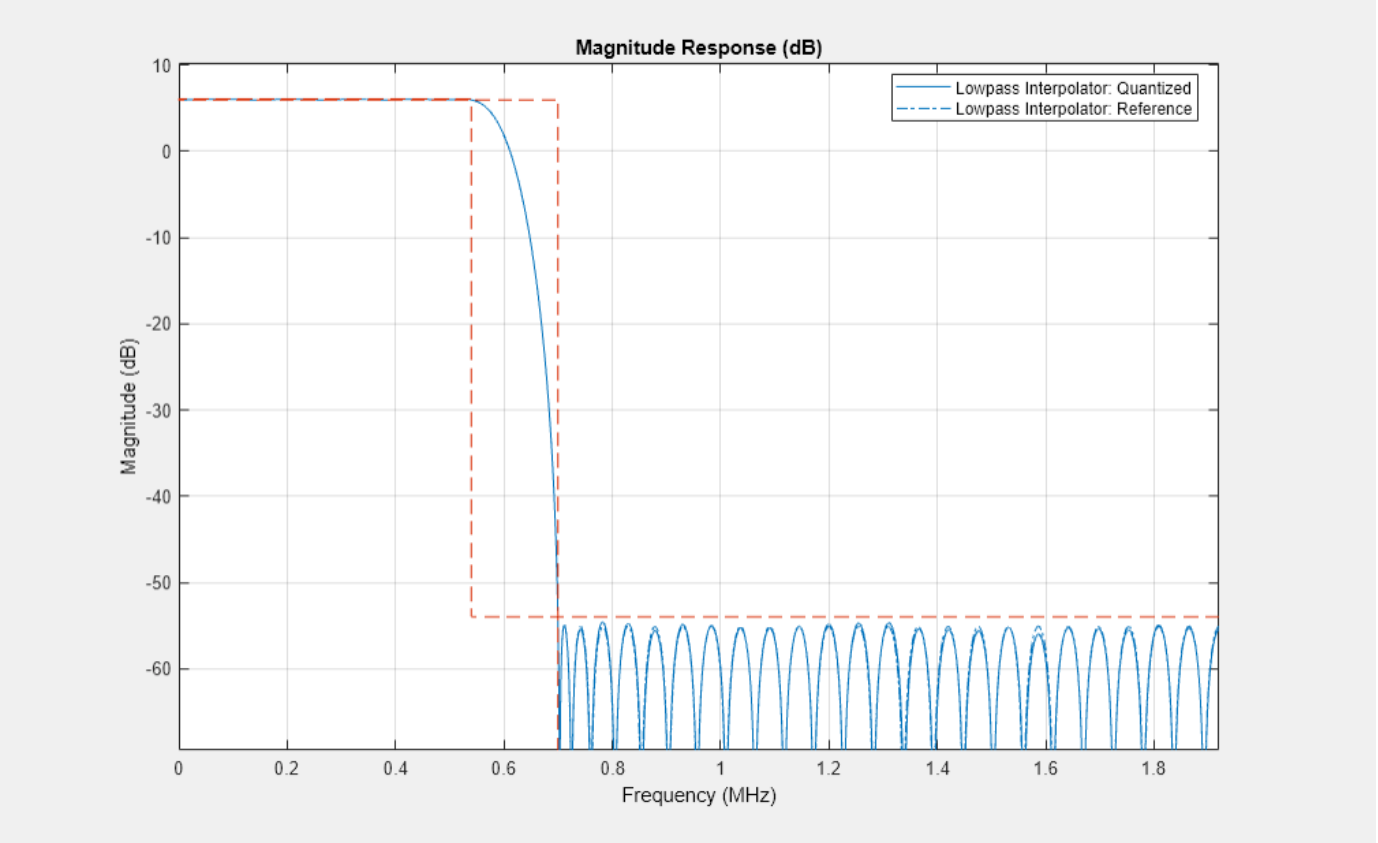
Configure the fixed-point properties of the gain correction and FIR-based System objects. The object uses the default `RoundingMethod` and `OverflowAction` property values (`'Floor'` and `'Wrap'` respectively).

```
% CIC Gain Correction
GainCorr.FullPrecisionOverride = false;
GainCorr.CoefficientsDataType = 'Custom';
GainCorr.CustomCoefficientsDataType = numerictype(fi(GainCorr.Numerator,1,16));
GainCorr.OutputDataType = 'Custom';
GainCorr.CustomOutputDataType = numerictype(1,18,14);
```

Fixed-Point Analysis

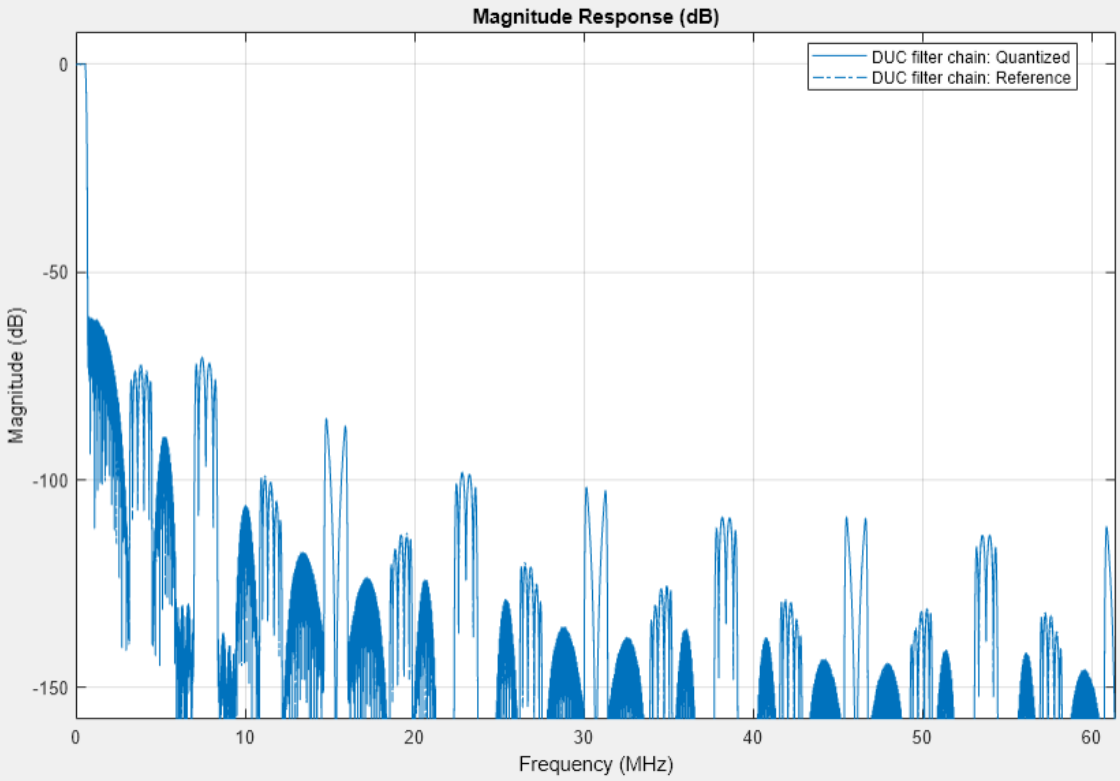
Inspect the quantization effects with `fvtool`. You can analyze the filters individually or in a cascade. `fvtool` shows the quantized and unquantized (reference) responses overlayed. For example, this figure shows the effect of quantizing the first FIR filter stage.

```
ducPlots.quantizedFIR = fvtool(lowpassFilt,'Fs',lowpassParams.FsIn*2,'arithmetic','fixed');

legend(ducPlots.quantizedFIR, ...
       'Lowpass Interpolator');
```



Redefine the `ducFilterChain` cascade object to include the fixed-point properties of the individual filters. Then use `fvtool` to analyze the entire filter chain and confirm that the quantized DUC still meets the specification.

```
ducFilterChain = dsp.FilterCascade(lowpassFilt,hbFilt,compFilt,cicFilt,GainCorr);
ducPlots.quantizedDUCResponse = fvtool(ducFilterChain, ...
    'Fs',FsIn*64,'Arithmetic','fixed');

legend(ducPlots.quantizedDUCResponse, ...
       'DUC filter chain');
```



HDL-Optimized Simulink Model

The next step in the design flow is to implement the DUC in Simulink using blocks that support HDL code generation.

Model Configuration

The model relies on variables in the MATLAB workspace to configure the blocks and settings. The filter blocks are configured by using the filter chain objects defined earlier in the example.

The input to the DUC comes from the `ducIn` variable. For now, assign a dummy value for `ducIn` so that the model can compute its data types. During testing, `ducIn` provides input data to the model.

```
ducIn = 0; %#ok<NASGU>
```

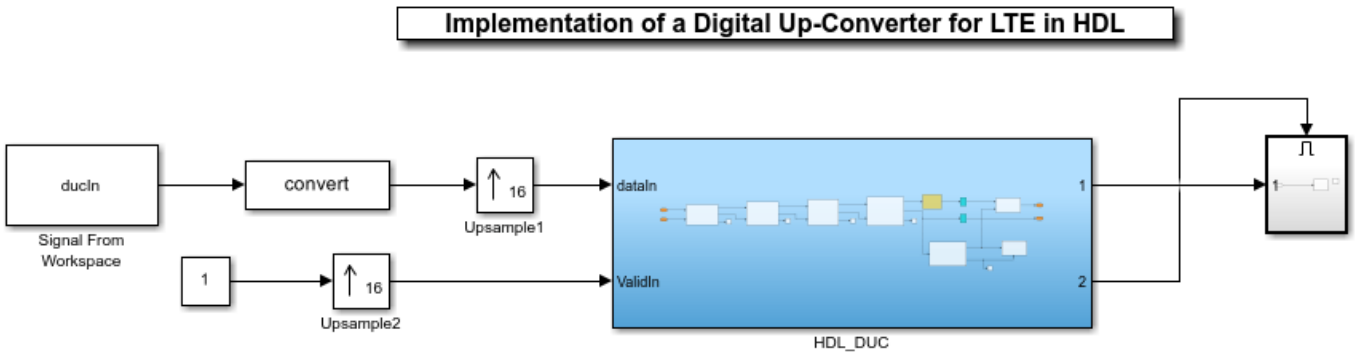
The `outputFrame` parameter sets the frame size of the output based on the DAC requirement. `outputFrame` affects the input vector size and valid sample spacing, and it should be a power of two.

```
outputFrame = 4;
```

Model Structure

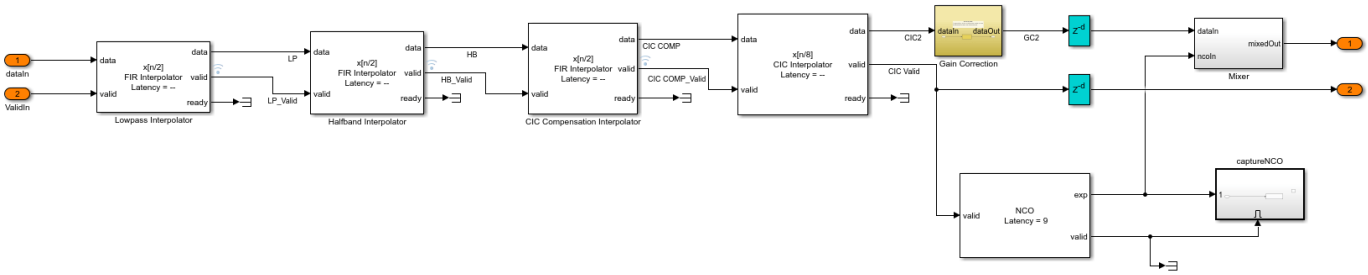
This figure shows the top level of the DUC Simulink model. The model imports the `ducIn` variable from the MATLAB workspace by using a **Signal From Workspace** block, converts the signal to 16-bit values, and applies the signal to the DUC. The design is single rate, and uses a *valid* signal to convey the rate change from block to block. To simulate the input running 64 times slower than the clock, it is upsampled by 64 with zero insertion. You can generate HDL code from the `HDL_DUC` subsystem.

```
modelName = 'DUCforLTEHDL';
open_system(modelName);
set_param(modelName,'Open','on');
```



The DUC implementation is inside the `HDL_DUC` subsystem.

```
set_param([modelName '/HDL_DUC'],'Open','on');
```

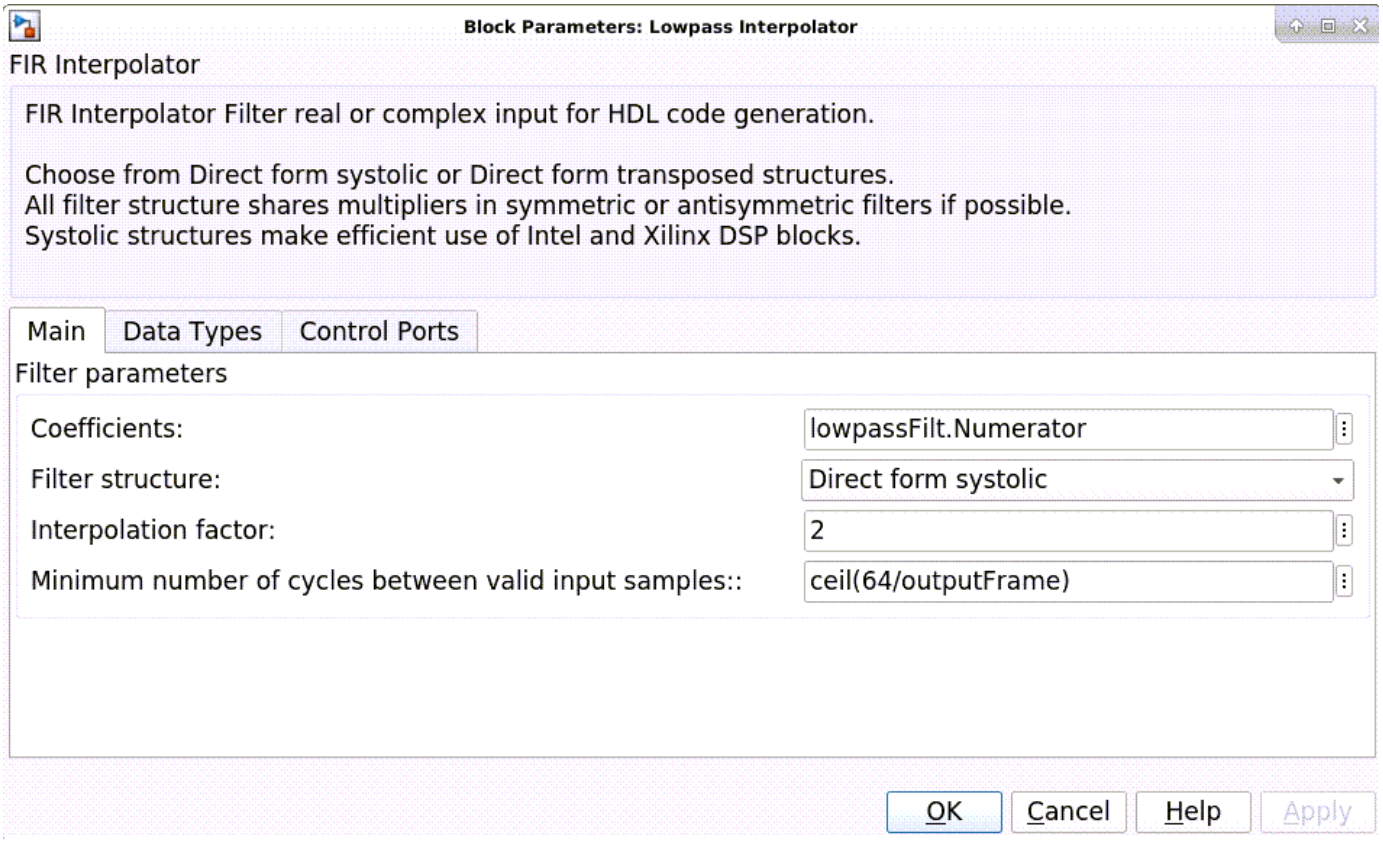


Filter Block Parameters

All of the filters are configured to inherit the coefficients of the corresponding System objects. Each block also has a "Minimum number of cycles between valid input" parameter that is used to optimize the resulting HDL code. The lowpass, halfband, and CIC compensation filters have cycles between valid inputs that can be used for resource sharing - 64, 32, and 16 cycles, respectively.

For example, because the sample rate of the input to the Lowpass Interpolation block is *Fclk*/64, 64 clock cycles are available to process each input sample.

The first filter interpolates by 2. Each polyphase branch is implemented by a separate FIR Filter. Because the number of cycles between valid input samples is greater than 1, each FIR is implemented using the partly serial systolic architecture. The filter has 69 coefficients in total, and after polyphase decomposition each branch has 35 coefficients. There are 64 cycles available for sharing, so each branch is implemented with a fully serial FIR. With complex input, each branch uses 2 multipliers, for a total of 4 multipliers in this filter.



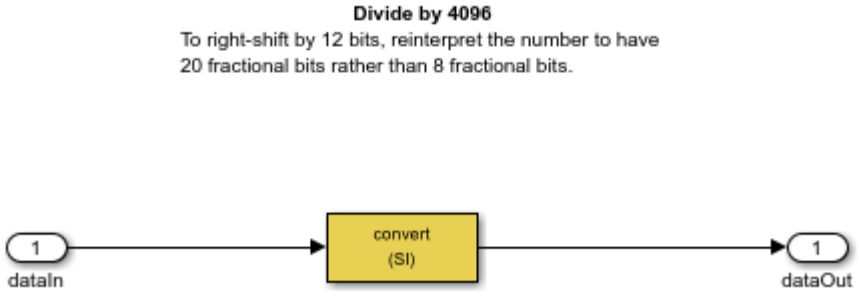
The second filter is a halfband interpolator. This filter can also take advantage of cycles between valid input to perform resource sharing. These cycles are available because each interpolator output has idle cycles between valid samples. The first filter has 64 cycles and interpolates by 2. Therefore, it will output data every 32 cycles. The second filter has 6 coefficients per polyphase branch and so it can be implemented as a fully serial FIR. In this filter, the second branch only contains one non-zero coefficient, and it is a power of 2. The FIR Interpolator block implements this branch as a shift rather than a multiplier. The second filter then has only 2 multipliers.

The third filter is a CIC compensation filter which interpolates by 2. It has 32 coefficients in total, which we specified when designing the filter. The filter is implemented using 2 fully serial complex FIRs, giving a total of 4 multipliers for this filter.

Gain Correction

The gain correction divides the output by 4096, which is equivalent to shifting right by 12 bits. Because the input and output signals of the gain correction each are expressed with 18 bits, the model implements this shift by reinterpreting the data type of the output signal. The Conversion block reinterprets the 12-bit number to have 20 fractional bits rather than 8 fractional bits.

```
set_param([modelName '/HDL_DUC/Gain Correction'],'Open','on');
```



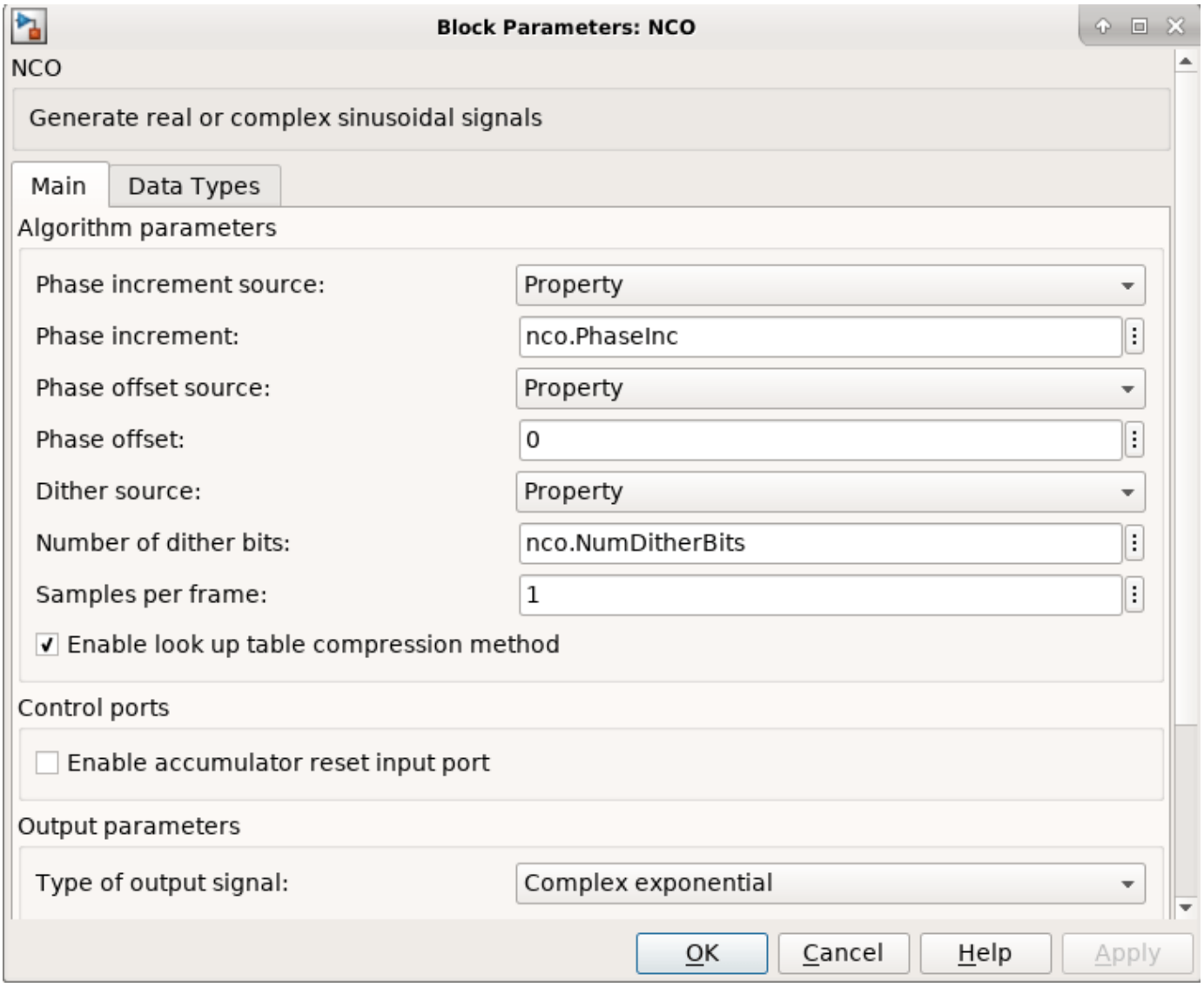
NCO Block Parameters

The NCO block generates a complex phasor at the carrier frequency. This signal goes to a mixer that multiplies the phasor with the output signal. The output of the mixer is sampled at 122.88 Msps.

Specify the desired frequency resolution, then calculate the number of accumulator bits required to achieve the desired resolution, and define the number of quantized accumulator bits. The NCO uses the quantized output of the accumulator to address the sine lookup table. Also compute the phase increment the NCO must use to generate the specified carrier frequency. The NCO applies phase dither to the accumulator bits that are removed during quantization.

```
nco.Fd = 1;
nco.AccWL = nextpow2(FsIn*64/nco.Fd) + 1;
nco.QuantAccWL = 12;
nco.PhaseInc = round((Fc*2^nco.AccWL)/(FsIn*64));
nco.NumDitherBits = nco.AccWL-nco.QuantAccWL;
```

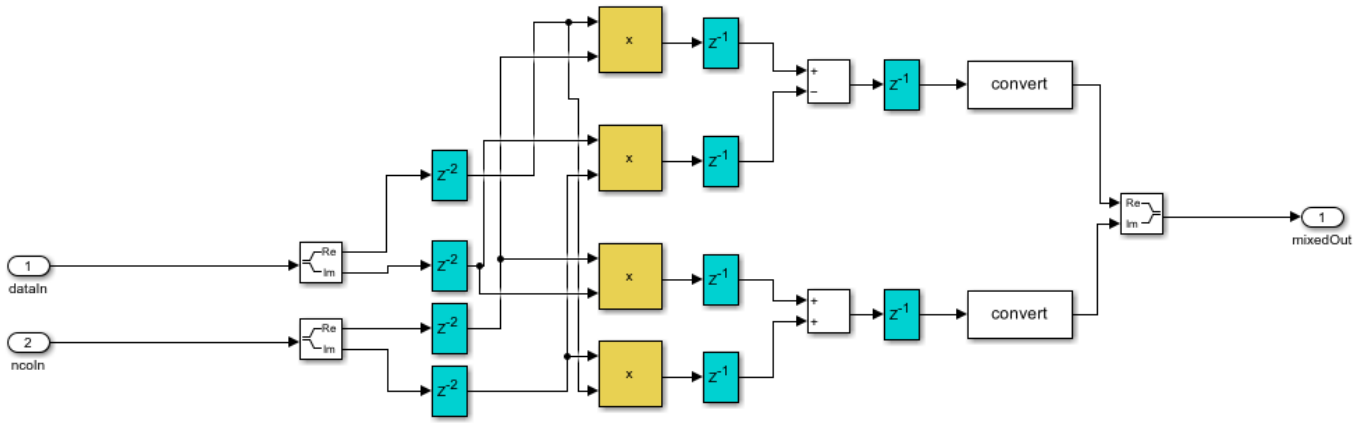
The NCO block in the model is configured with the parameters defined in the `nco` structure. This figure shows the NCO block parameters dialog.



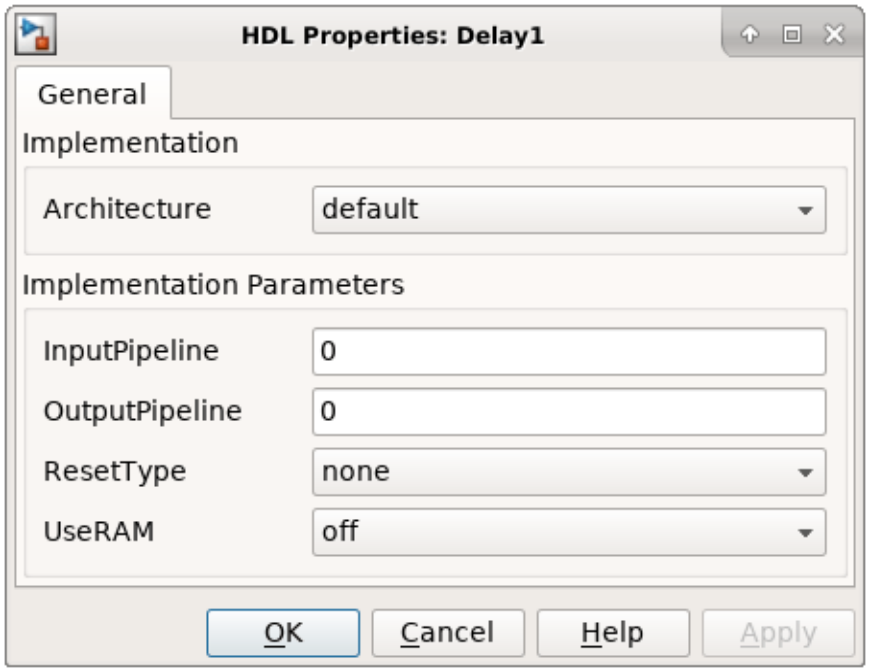
Mixer

The Mixer subsystem performs a complex multiply of the filter output and NCO.

```
set_param([modelName '/HDL_DUC/Mixer'],'Open','on');
```



To map the mixer to DSP slices on FPGA, the mixer is pipelined and the blocks have specific settings. The delay blocks in the mixer are configured to have reset turned off as shown in the image. There are 2 pipeline stages at the input, 1 between the multiplier and adder, and another post-adder. Also, the multipliers and adders are configured to use full-precision output. These blocks use `Floor` rounding method, and the saturate on overflow logic is disabled.



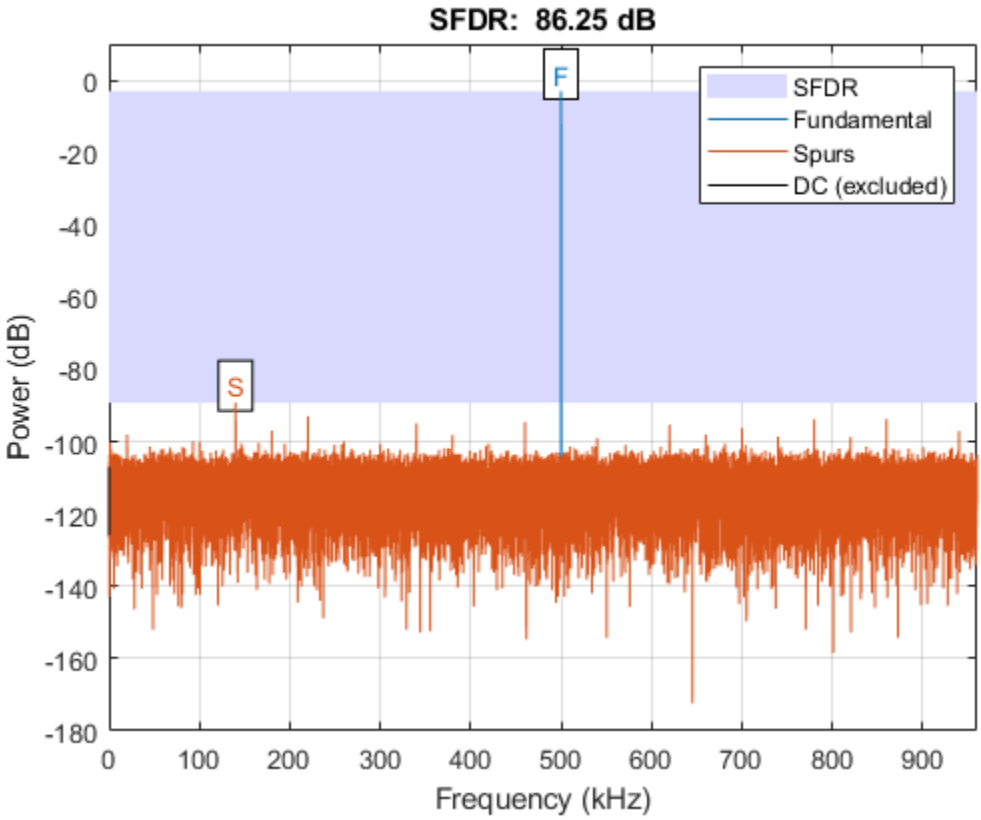
Sinusoid on Carrier Test

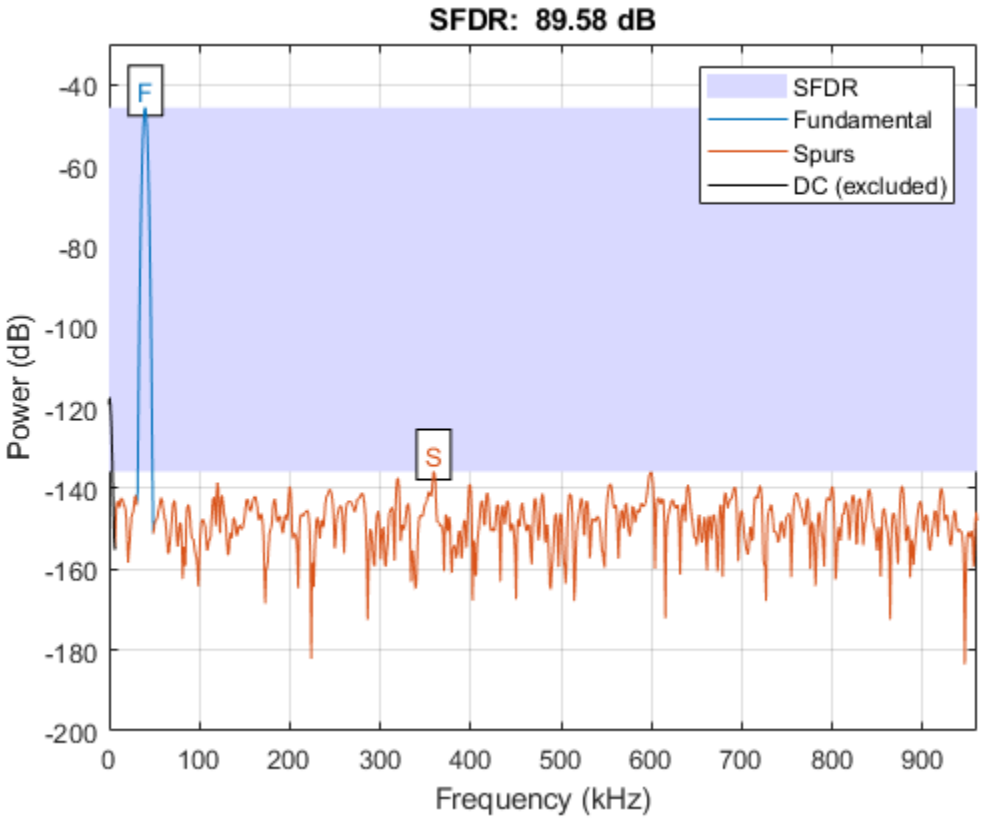
To test the DUC, pass a 40kHz sinusoid through the DUC and modulate the output signal onto the carrier frequency. Demodulate and resample the signal. Then measure the spurious free dynamic range (SFDR) of the resulting tone and the SFDR of the NCO output.

```
% Initialize random seed before executing any simulations.
rng(0);

% Generate a 40kHz test tone.
```

```
ducIn = DUCTestUtils.GenerateTestTone(40e3);

% Upconvert the test signal with the floating-point DUC.
ducTx = DUCTestUtils.UpConvert(ducIn,FsIn*64,Fc,ducFilterChain);
release(ducFilterChain);

% Down convert the output of DUC.
ducRx = DUCTestUtils.DownConvert(ducTx,FsIn*64,Fc);

% Upconvert the test signal by running the fixed-point Simulink model.
simOut = sim(modelName);

% Downconvert the output of DUC.
simTx = simOut.ducOut;
simRx = DUCTestUtils.DownConvert(simTx,FsIn*64,Fc);

% Measure the SFDR of the NCO, floating point DUC, and fixed-point DUC outputs.
results.sfdrNCO = sfdr(real(simOut.ncoOut),FsIn);
results.sfdrFloatDUC = sfdr(real(ducRx),FsIn);
results.sfdrFixedDUC = sfdr(real(simRx),FsIn);

disp('SFDR Measurements');
disp(['   Floating-point DUC SFDR: ',num2str(results.sfdrFloatDUC) ' dB']);
disp(['   Fixed-point NCO SFDR: ',num2str(results.sfdrNCO) ' dB']);
disp(['   Fixed-point DUC SFDR: ',num2str(results.sfdrFixedDUC) ' dB']);
fprintf(newline);

% Plot the SFDR of the NCO and fixed-point DUC outputs.
ducPlots.ncoOutSDFR = figure;
sfdr(real(simOut.ncoOut),FsIn);

ducPlots.ducOutSDFR = figure;
sfdr(real(simRx),FsIn);
```

```
SFDR Measurements
   Floating-point DUC SFDR: 287.9814 dB
   Fixed-point NCO SFDR: 86.2454 dB
   Fixed-point DUC SFDR: 89.5756 dB
```


SFDR: 86.25 dB
F
S
SFDR
Fundamental
Spurs
DC (excluded)
Power (dB)
0
-20
-40
-60
-80
-100
-120
-140
-160
-180
0
100
200
300
400
500
600
700
800
900
Frequency (kHz)



LTE Signal Test

You can use an LTE test signal to perform more rigorous testing of the DUC. Generate a standard-compliant LTE waveform by using LTE Toolbox™ functions. Then, upconvert the waveform with the DUC model. Use LTE Toolbox functions to measure the error vector magnitude (EVM) of the resulting signals.

```
rng(0);
% Execute this test only if you have the LTE Toolbox product.
if license('test','LTE_Toolbox')

    % Generate an LTE test signal by using LTE Toolbox functions.
    [ducIn, sigInfo] = DUCTestUtils.GenerateLTETestSignal();

    % Upconvert the signal with the floating-point DUC and modulate onto carrier.
    ducTx = DUCTestUtils.UpConvert(ducIn,FsIn*64,Fc,ducFilterChain);
    release(ducFilterChain);

    % Add noise to the transmit signal.
    ducTxAddNoise = DUCTestUtils.AddNoise(ducTx);

    % Downconvert the received signal.
    ducRx = DUCTestUtils.DownConvert(ducTxAddNoise,FsIn*64,Fc);

    % Upconvert the signal by using the Simulink model.
    simOut = sim(modelName);
```

```
    % Add noise to the transmit signal.
    simTx = simOut.ducOut;
    simTxAddNoise = DUCTestUtils.AddNoise(simTx);

    % Downconvert the received signal.
    simRx = DUCTestUtils.DownConvert(simTxAddNoise,FsIn*64,Fc);

    results.evmFloat = DUCTestUtils.MeasureEVM(sigInfo,ducRx);
    results.evmFixed = DUCTestUtils.MeasureEVM(sigInfo,simRx);

    disp('LTE EVM Measurements');
    disp(['   Floating-point DUC RMS EVM: '  num2str(results.evmFloat.RMS*100,3) '%']);
    disp(['   Floating-point DUC Peak EVM: ' num2str(results.evmFloat.Peak*100,3) '%']);
    disp(['   Fixed-point DUC RMS EVM: '     num2str(results.evmFixed.RMS*100,3) '%']);
    disp(['   Fixed-point DUC Peak EVM: '    num2str(results.evmFixed.Peak*100,3) '%']);
    fprintf(newline);

end
```

```
LTE EVM Measurements
   Floating-point DUC RMS EVM: 0.813%
   Floating-point DUC Peak EVM: 2.53%
   Fixed-point DUC RMS EVM: 0.816%
   Fixed-point DUC Peak EVM: 2.82%
```

HDL Code Generation and FPGA Implementation

To generate the HDL code for this example you must have the HDL Coder™ product. Use the `makehdl` and `makehdltb` commands to generate HDL code and an HDL test bench for the `HDL_DUC` subsystem. The DUC was synthesized on a Xilinx® Zynq®-7000 ZC706 evaluation board. The table shows the post place-and-route resource utilization results for outputFrame of size 4. The design met timing with a clock frequency of 335 MHz.

```
T = table(...
    categorical({'LUT'; 'LUTRAM'; 'FF'; 'BRAM'; 'DSP'}),...
    categorical({'4708'; '654'; '6849'; '2'; '32'}),...
    'VariableNames',{'Resource','Usage'})
```

```
T =

  5x2 table

    Resource    Usage
    ________    _____

     LUT        4708
     LUTRAM     654
     FF         6849
     BRAM       2
     DSP        32
```

See Also

FIR Interpolator | CIC Interpolator | NCO

Related Examples

- “Implement Digital Downconverter for FPGA” on page 1-66

HDL Optimized System Design

FIR Filter Architectures for FPGAs and ASICs

The Discrete FIR Filter, FIR Decimator, FIR Interpolator, Farrow Rate Converter, Channelizer, and Channel Synthesizer blocks all use the same FIR filter architectures to implement their algorithms.

These blocks provide filter implementations that make trade-offs between resources and throughput. The filter implementations also consider vendor-specific hardware details of the DSP blocks when adding pipeline registers to the architecture. These differences in pipeline register locations help fit the filter design to the DSP blocks on the FPGA. For a filter implementation that matches multipliers, pipeline registers, and pre-adders to the DSP configuration of your FPGA vendor, specify your target device when you generate HDL code.

The filter implementations remove multipliers for zero-valued coefficients, such as in half-band filters and Hilbert transforms. When you use scalar input data, the filters share multipliers for symmetric and antisymmetric coefficients. Frame-based filters do not support symmetry optimization.

The FIR filter implementations implement efficient complex multiplier architectures and support frame based input by using polyphase filters that share hardware resources across subfilters .

The architecture diagrams on this page assume a transfer function that has L coefficients (before optimizations that share multipliers for symmetric or antisymmetric or remove multipliers for zero-valued coefficients). N represents the number of cycles between valid input samples.

Filter Structure	Blocks	Settings
"Fully Parallel Systolic Architecture" on page 2-6	• Discrete FIR Filter • Farrow Rate Converter • Channelizer • Channel Synthesizer • FIR Decimator • FIR Interpolator	• For Discrete FIR Filter, Farrow Rate Converter, Channelizer, and Channel Synthesizer — Set **Filter structure** to `Direct form systolic`. • For FIR Decimator and FIR Interpolator — Set **Filter structure** to `Direct form systolic` and **Number of Cycles** to 1.
"Fully Parallel Transposed Architecture" on page 2-6	• Discrete FIR Filter • Farrow Rate Converter • Channelizer • Channel Synthesizer • FIR Decimator • FIR Interpolator	Set **Filter structure** to `Direct form transposed`.

Filter Structure	Blocks	Settings
"Partly Serial Systolic Architecture (1 < N < L)" on page 2-7	• Discrete FIR Filter • Farrow Rate Converter • FIR Decimator • FIR Interpolator	• For Discrete FIR Filter — Set **Filter structure** to `Partly serial systolic` and specify a serialization factor by either cycles or multipliers that results in $1<N<L$. • For Farrow Rate Converter, FIR Decimator, and FIR Interpolator — Set **Filter structure** to `Direct form systolic` and **Minimum number of cycles between valid input samples** to a value greater than 1 but less than L.
"Fully Serial Systolic Architecture (N ≥ L)" on page 2-8	• Discrete FIR Filter • Farrow Rate Converter • FIR Decimator • FIR Interpolator	• For Discrete FIR Filter — Set **Filter structure** to `Partly serial systolic` and specify a serialization factor by cycles such that $N \geq L$, or set the **Number of multipliers** to 1. • For Farrow Rate Converter, FIR Decimator, and FIR Interpolator — Set **Filter structure** to `Direct form systolic` and **Minimum number of cycles between valid input samples** to a value greater than L.

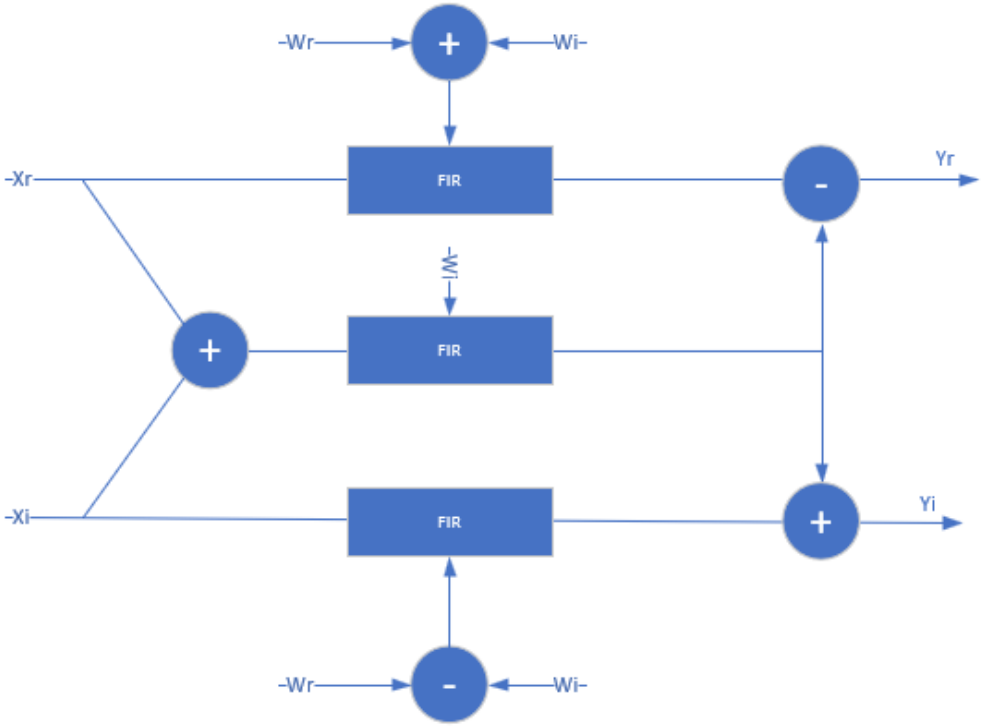
Complex Multipliers

If either data or coefficients are complex but not both, the filter blocks implement one filter to calculate the real output and a second filter to calculate the imaginary part. This implementation results in two multipliers for each filter tap.

When both the data and coefficients are complex, the block implements three filters in parallel. The diagram shows the filter implementation for complex input data $X = X_r + i \times X_i$ and complex coefficients $W = W_r + i \times W_i$.

When you specify coefficients from a parameter, $W_r + W_i$ and W_r-W_i are pre-calculated, so this implementation uses 3 DSP blocks for each filter tap, plus the input adder and two output adders. The input to each filter tap multiplier grows by one bit.

When you use programmable coefficients, the filter uses 2 more adders for each filter tap. These adders calculate the coefficients $W_r + W_i$ and W_r-W_i.

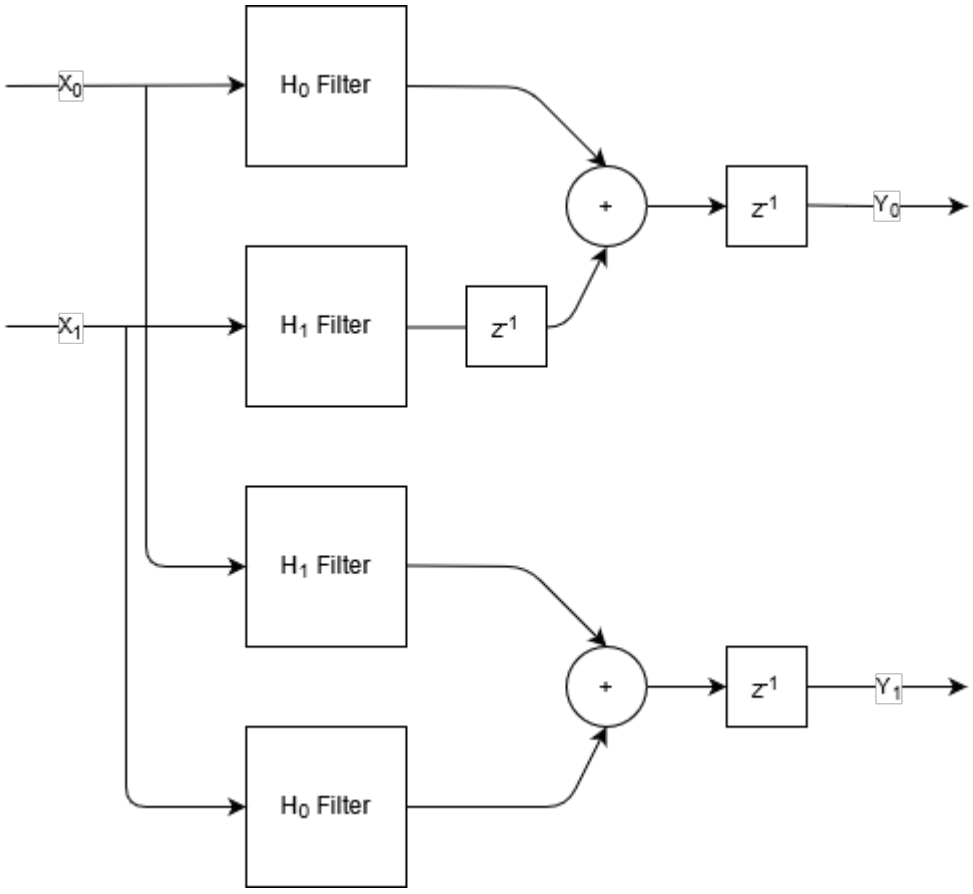


Frame-Based Input Data

The Discrete FIR Filter, FIR Decimator, FIR Interpolator, Channelizer, and Channel Synthesizer blocks accept frame-based input data to support gigasamples-per-second throughput. When you apply frame-based input data, the FIR filter implements a polyphase decomposition of your filter coefficients into V subfilters, where V is the size of the input vector. The frame-based filter increases throughput and uses more hardware resources than the scalar-input case. Frame-based filters do not implement symmetry optimization.

For a filter with a 1-by-2 input vector, $[Y_0 \ Y_1]$, the diagram shows the polyphase decomposition into two subfilters that implement this equation.

$$Y_0(z) = X_0(z)H_0(z) + z^{-1}X_1(z)H_1(z)$$
$$Y_1(z) = X_0(z)H_1(z) + X_1(z)H_0(z)$$



Each subfilter takes scalar input and is implemented with the architecture you selected, either `Direct form systolic` or `Direct form transposed`. If the subfilters have different latencies due to different numbers of coefficients, or zero-value coefficient optimization, then the implementation includes internal delays to align the output samples. You cannot use frame-based input with the serial systolic architecture.

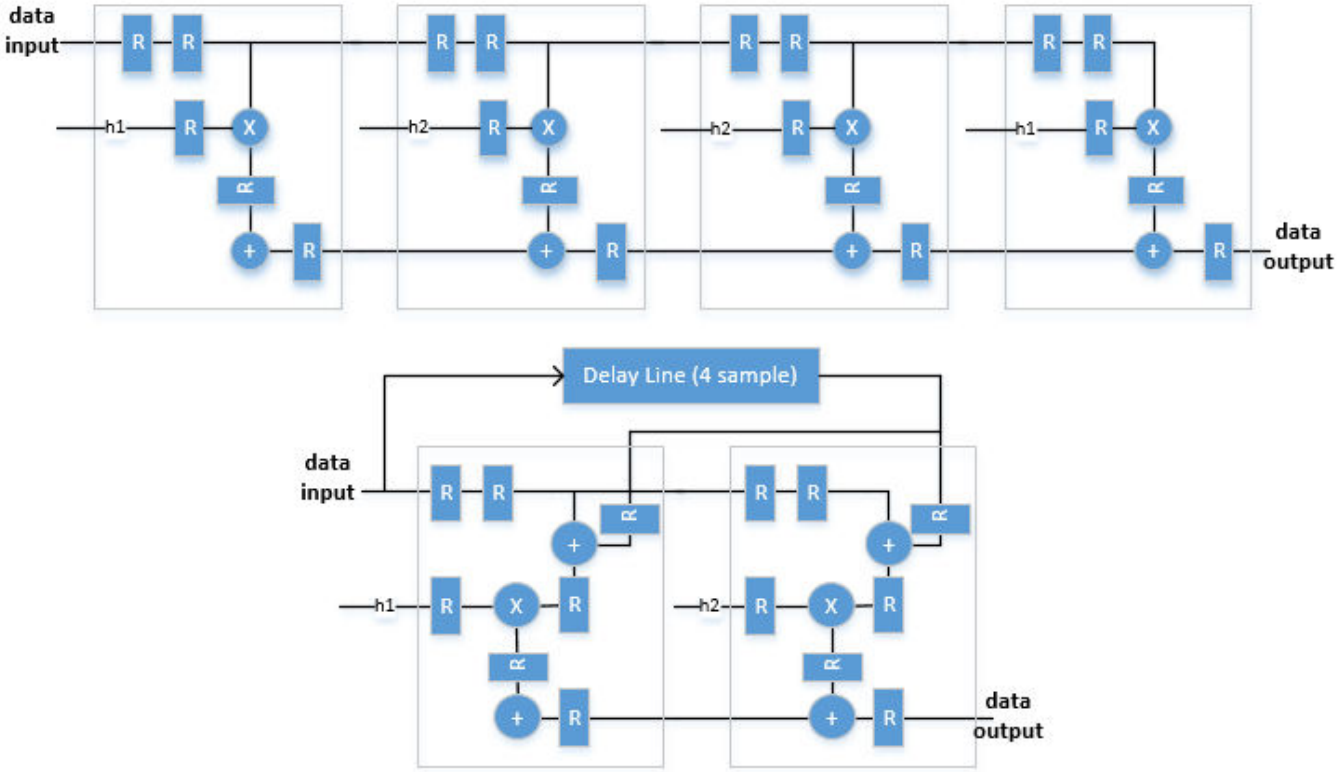
When you use frame-based input with programmable coefficients, the output may not match sample-for-sample with the output in scalar mode. This difference is because of the internal timing of

applying each sample in the input vector to the subfilters. Changes in the input coefficients effectively occur at different individual input samples than they do in scalar mode.

Fully Parallel Systolic Architecture

This filter architecture is a fully parallel systolic architecture with optimizations for symmetry or anti-symmetry and zero-valued coefficients. The latency depends on the coefficient symmetry and is displayed on the block icon.

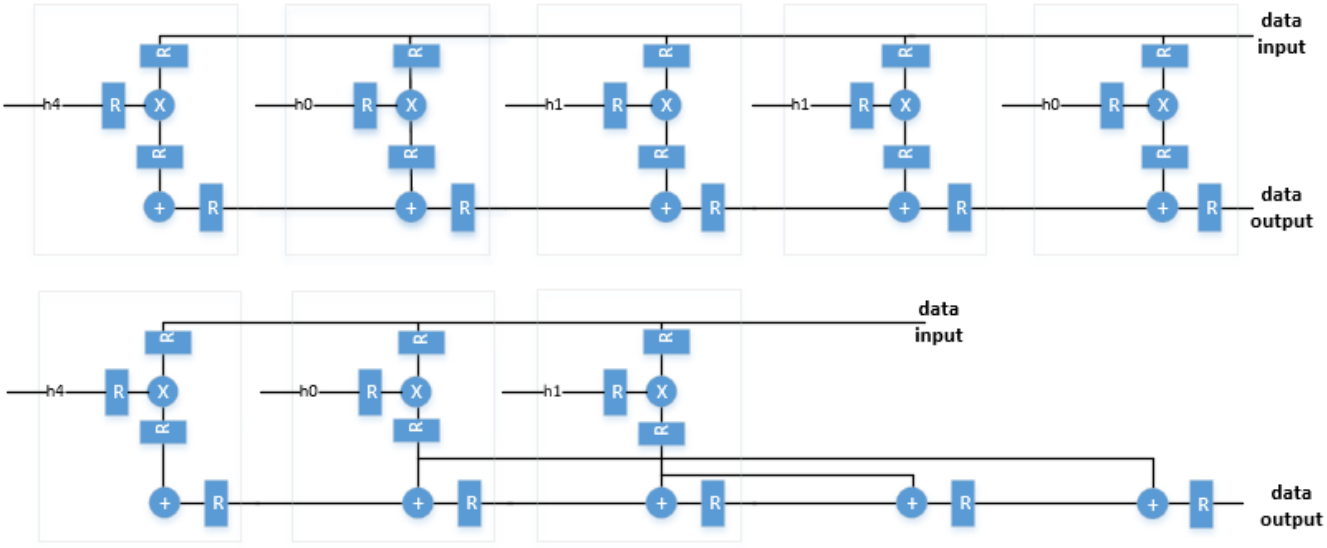
When symmetric pairs of coefficients have equal absolute values, they share one DSP block. This pair-sharing enables the implementation to use the pre-adder in Xilinx® and Intel® DSP blocks. The top half of the diagram shows a symmetric filter without the pair coefficient optimization. The bottom half of the diagram shows the architecture using the pair coefficient optimization.



Fully Parallel Transposed Architecture

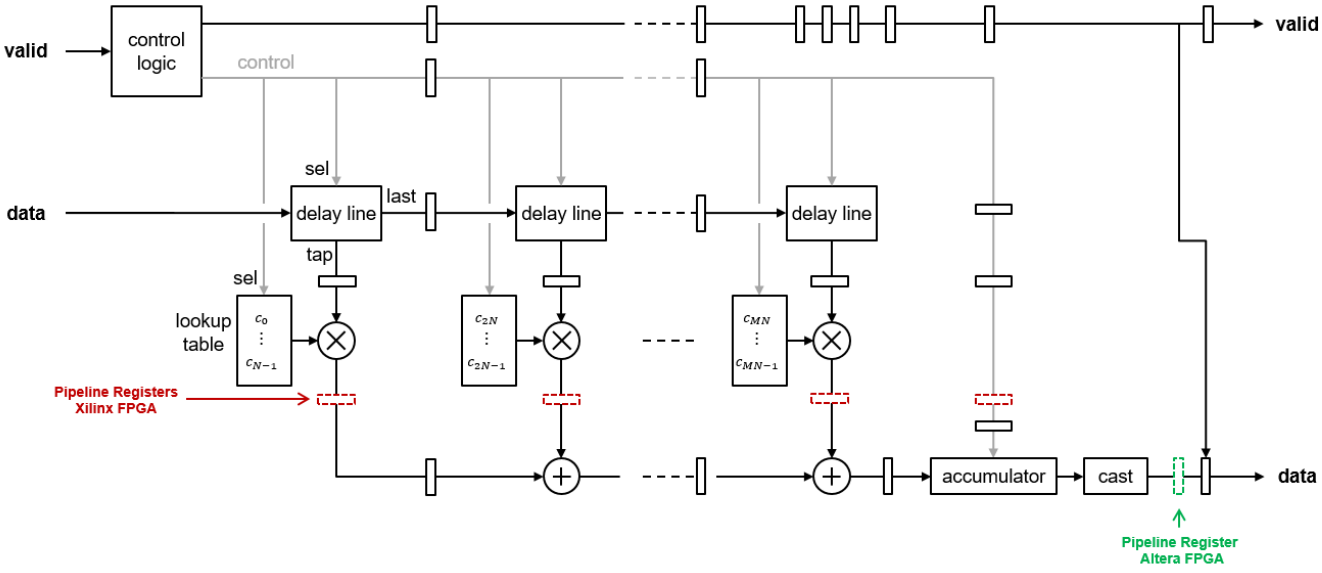
The fully parallel transposed architecture minimizes multipliers by sharing multipliers for any two or more coefficients that have equal absolute values. It also removes multipliers for zero-valued coefficients. The latency of the filter is six cycles when you use scalar input. This latency does not change with coefficient values.

The top half of the diagram shows the theoretical architecture for a partly-symmetric filter without the equal-absolute-value coefficient optimization. The bottom half of the diagram shows the transposed architecture as implemented using the equal-value coefficient optimization. If the coefficients are antisymmetric, the output adder becomes a subtraction.



Partly Serial Systolic Architecture (1 < N < L)

The partly serial implementation uses M = `ceil(`L/N`)` systolic cells. Each cell consists of a delay line, coefficient lookup table, and DSP (multiply-add) block. The coefficients are spread across the M lookup tables. The computation performed by each DSP block is serialized. Input samples to the block must be scalar and at least N cycles apart. The latency of the block is M + `ceil(`L/M`)` + 5.



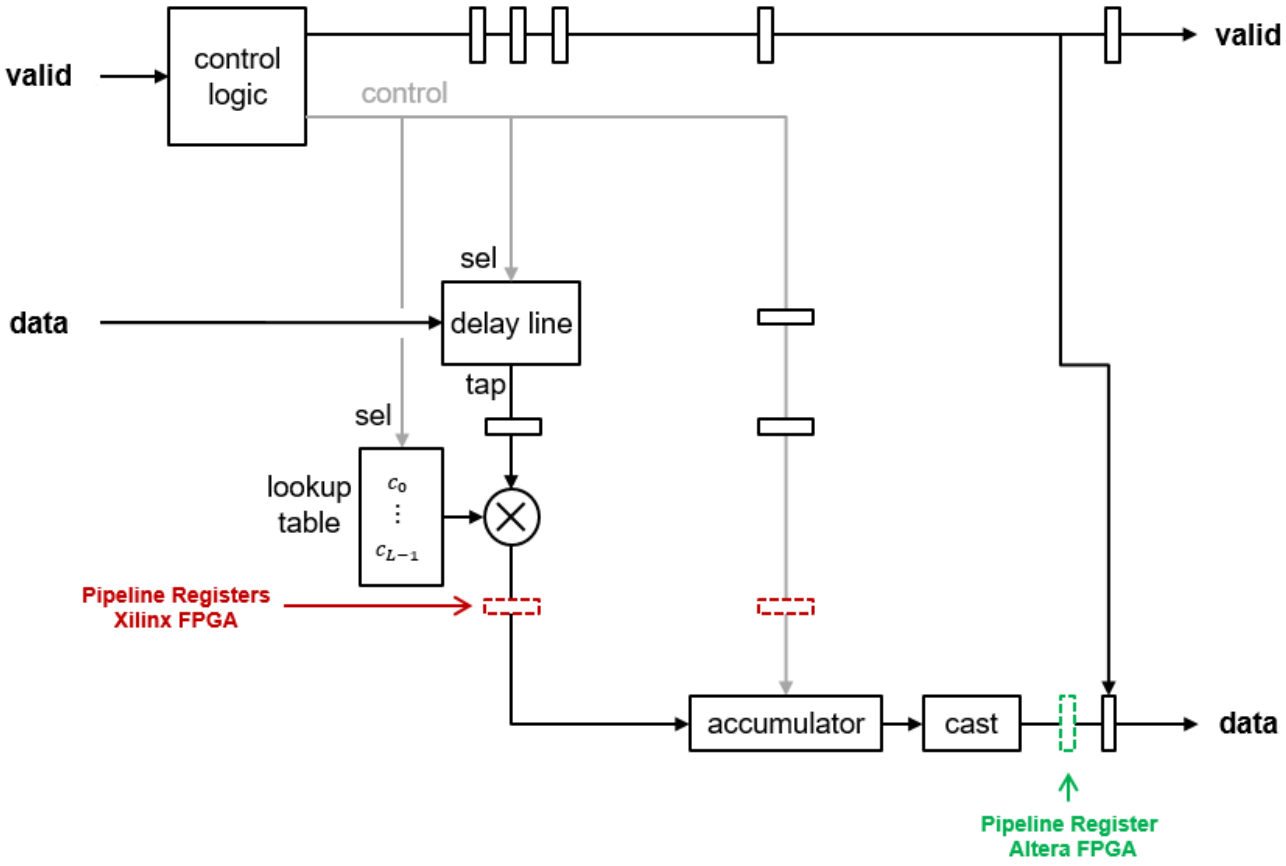
If all the coefficients in the lookup table for a multiplier are zeros or powers of two, the implementation does not include that multiplier. The powers of two multiplications are implemented as shifts.

The block implements a RAM-based delay line that uses fewer resources than a register-based implementation. Uninitialized RAM locations can result in X values at the start of your HDL

simulation. You can avoid X values by initializing the RAM from your test bench, or by enabling the **Initialize all RAM blocks** Configuration Parameter. This parameter sets the RAM locations to0 for simulation and is ignored by synthesis tools.

Fully Serial Systolic Architecture (N ≥ L)

When you choose a serialization factor such that $N \geq L$, the block implements a fully serial systolic architecture. For real coefficients and real input, the filter uses a single DSP (multiply-add) block with a delay line and a lookup table for all L coefficients. Input samples must be at least N cycles apart. The latency of the filter is $L + 5$.



See Also

Related Examples

- “Fully Parallel Systolic FIR Filter Implementation” on page 3-5
- “Partly Serial Systolic FIR Filter Implementation” on page 3-9

High-Throughput HDL Algorithms

DSP HDL Toolbox™ provides blocks to implement high-bandwidth applications such as radar, SIGINT, and 5G FR2 wireless. To meet their bandwidth requirements, these applications need to process incoming data as multiple samples in parallel, or frame-based processing. Frame-based processing increases throughput by implementing the algorithm in parallel on each sample in the input vector. These implementations increase data throughput but use more hardware resources. The ports of blocks that support frame-based processing accept column vector input and output signals. Each element of the vector represents a sample in time. Using frame-based input can achieve gigasamples-per-second (GSPS) throughputs. These high-throughput rates are also referred to as super-sample rates.

You can also find blocks that support frame-based input and hardware-optimized algorithms in Wireless HDL Toolbox™ libraries.

Blocks that Support Frame-Based Input

Supported Block	Parameters to Enable Frame Input	Limitations
FFT and IFFT	Connect a column vector to the input **data** port. The vector size must be a power of 2 between 1 and 64 and cannot be greater than the FFT length.	Frame-based input is supported only when **Architecture** is set to `Streaming Radix 2^2`.
Channelizer and Channel Synthesizer	Connect a column vector to the input **data** port. The vector size must be a power of 2 between 1 and 64 and cannot be greater than the FFT length.	
Discrete FIR Filter	Connect a column vector to the input **data** port. The vector size must be less than or equal to 64.	You cannot use frame-based input with the partly serial architecture.
Biquad Filter	Connect a column vector to the input **data** port. The input vector size can be up to 64 samples, but large vector sizes can make the calculation of internal datatypes challenging. Vector sizes of up to 16 samples are practical for hardware implementation.	Vector input is supported only when you set **Filter structure** to `Pipelined feedback form`.

Supported Block	Parameters to Enable Frame Input	Limitations
FIR Decimator	Connect a column vector to the input **data** port. The vector size must be less than or equal to 64 samples.	• You cannot use frame-based input with **Minimum number of cycles between valid input samples** greater than 1. • When the vector size is less than the decimation factor, the decimation factor must be an integer multiple of the vector size. • When the vector size is greater than the decimation factor, the vector size must be an integer multiple of the decimation factor.
FIR Interpolator	Connect a column vector to the input **data** port. The vector size must be less than or equal to 64 samples.	You cannot use frame-based input with **Minimum number of cycles between valid input samples** greater than 1.
CIC Decimator	Connect a column vector to the input **data** port. The input vector size can be up to 64 samples.	Vector input is not supported with programmable decimation rate.
CIC Interpolator	Connect a column vector to the input **data** port. The input vector size can be up to 64 samples.	Vector input is not supported with programmable interpolation rate.
NCO	Set the **Samples per frame** parameter to the desired output vector size.	
Complex to Magnitude-Angle	Connect a column vector to the input **data** port. The input vector size can be up to 64 samples.	
Delay	1 Connect a column vector to the input port. The input vector size can be up to 512 samples. 2 Set the **Input processing** parameter to `Columns as channels (frame-based)`.	

Supported Block	Parameters to Enable Frame Input	Limitations
NR LDPC Encoder	Connect a column vector of 64 samples to the input **data** port. The pattern of input bits in the vector depends on the **liftingSize**, see “Specifying Vector Input” (Wireless HDL Toolbox).	
NR LDPC Decoder	Connect a column vector of 64 samples to the input **data** port. The pattern of input bits in the vector depends on the **liftingSize**, see “Specifying Vector Input” (Wireless HDL Toolbox).	
WLAN LDPC Decoder	Connect a column vector of 8 samples to the input **data** port.	
Puncturer	Connect a column vector of 2 to 7 samples to the input **data** port. If input is a vector, the size of the vector must match the **Encoder rate** parameter value.	
OFDM Modulator	Connect a column vector to the input **data** port. The vector size must be a power of 2 in the range from 1 to 64, and less than or equal to the FFT length. For more information on how to specify vector inputs, see “Specifying Vector Input” (Wireless HDL Toolbox).	
OFDM Demodulator	Connect a column vector to the input **data** port. The vector size must be a power of 2 in the range from 1 to 64, and less than or equal to the FFT length.	

See Also

Related Examples

- “High-Throughput Channelizer for FPGA”
- “Gigasamples-per-Second Correlator and Peak Detector” on page 1-43

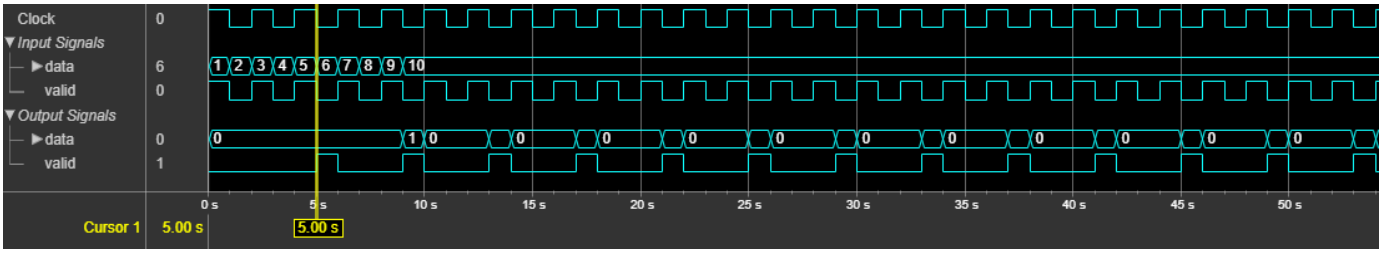
Hardware Control Signals

In this section...

Streaming Interface with Valid Signal

DSP System Toolbox™ blocks and functions process matrices of data. In hardware, processing a large matrix of data at one time has a high cost in memory and area. To save resources, serial processing is preferable in HDL designs. DSP HDL Toolbox blocks use a streaming sample interface that represents the characteristics of a physical signal. This interface means the blocks operate on one sample at a time, or on a small vector of samples at a time, rather than a large matrix. The blocks accept and return this streaming data accompanied by a control signal that indicates when the data is valid. You can model real-world data patterns such as sample rates or bursty data by using this valid control signal. One valid signal applies to all values in an input or output vector.

This waveform shows the input and output signals of the CIC Decimator block. The block is configured with a decimation rate of 2. The input data is valid every second cycle. The block decimates this signal and returns valid output data every fourth cycle.



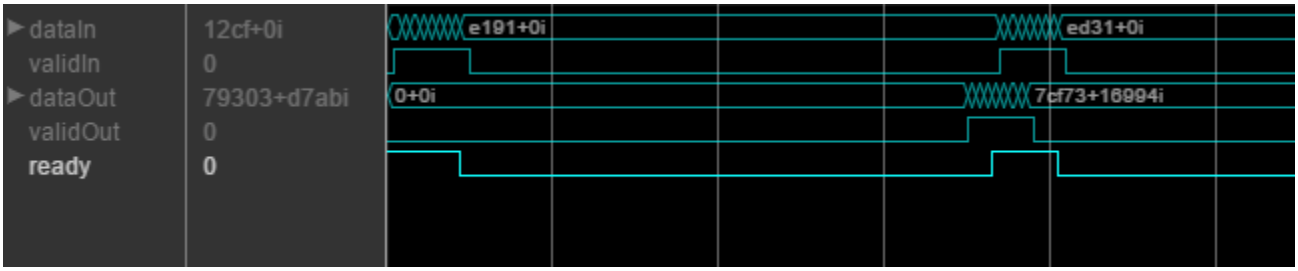
You can increase throughput by using frame-based data streams that pass vectors of signals through the blocks and System objects. Frame-based processing increases throughput by implementing the algorithm in parallel on each sample in the input vector. These implementations increase data throughput while using more hardware resources compared to scalar streaming algorithms. Most algorithms in DSP HDL Toolbox support input and output vectors of up to 64 samples, but your vector size might be further limited by the resources available for your design. For details of frame-based support in DSP HDL Toolbox, see “High-Throughput HDL Algorithms” on page 2-9.

Backpressure Signal

Some DSP HDL Toolbox blocks provide a backpressure output control signal, named `ready`. This signal indicates when the block has room for a new input sample. This signal is necessary because these blocks cannot accept new samples while performing certain internal computations or when internal storage is full, and the latency of those computations can vary with the configuration of the block.

For example, when you use the burst architecture of the FFT block, you cannot provide the next frame of input data until memory space is available. The `ready` signal indicates when the algorithm

can accept new input data. When `ready` is `1` (`true`), you can apply input `data` and `valid` signals. The algorithm ignores any input `data` and `valid` signals when `ready` is `0` (`false`). The waveform shows the `ready` signal is `1` at the start of simulation, then goes low when the block receives the first burst of input data. Then, while the block is processing that burst of data, the `ready` signal stays low. When the block has room for new input samples, it sets the `ready` signal to `1` again. Once the `ready` signal is high, the model can apply the next burst of input `data`.



For an example of how to use the ready signal with interpolators, see “Control Data Rate Using Ready Signal” on page 3-26.

Reset Signal

Most DSP HDL Toolbox blocks provide an optional input reset signal that clears internal states. You can also connect the global reset signal from HDL Coder™ tools to your designs. These are some considerations for using the reset signal options.

- By default, the filter blocks connect the generated HDL global reset to only the control path registers. The two reset parameters, **Enable reset input port** and **Use HDL global reset**, connect a reset signal to the data path registers. Because of the additional routing and loading on the reset signal, resetting data path registers can reduce synthesis performance.
- The **Enable reset input port** parameter enables the **reset** port on the block. The reset signal implements a local synchronous reset of the data path registers. For optimal use of FPGA resources, this option does not connect the reset signal to registers targeted to the DSP blocks of the FPGA.
- The **Use HDL global reset** parameter connects the generated HDL global reset signal to the data path registers. This parameter does not change the appearance of the block or modify simulation behavior in Simulink®. The generated HDL global reset can be synchronous or asynchronous depending on the **HDL Code Generation > Global Settings > Reset type** parameter in the model Configuration Parameters. Depending on your device, using the global reset might move registers out of the DSP blocks and increase resource use.
- When you select the **Enable reset input port** and **Use HDL global reset** parameters together, the global and local reset signals clear the control and data path registers.

Reset Considerations for Generated Test Benches

- FPGA-in-the-loop initialization provides a global reset but does not automatically provide a local reset. With the default reset parameters, the data path registers that are not reset can result in FPGA-in-the-loop (FIL) mismatches if you run the FIL model more than once without resetting the board. Select **Use HDL global reset** to reset the data path registers automatically, or select **Enable reset input port** and assert the local reset in your model so the reset signal becomes part of the Simulink FIL test bench.

- The generated HDL test bench provides a global reset but does not automatically provide a local reset. With the default reset parameters and the default register reset Configuration Parameters, the generated HDL code includes an initial simulation value for the data path registers. However, if you are concerned about X-propagation in your design, you can set the **HDL Code Generation > Global Settings > Coding style > No-reset register initialization** parameter in Configuration Parameters to `Do not initialize`. In this case, with the default block reset parameters, the data path registers that are not reset can cause X-propagation on the data path at the start of HDL simulation. Select **Use HDL global reset** to reset the data path registers automatically, or select **Enable reset input port** and assert the local reset in your model so the reset signal becomes part of the generated HDL test bench.

See Also

Related Examples

- “Implement FFT Algorithm for FPGA”
- “Control Data Rate Using Ready Signal” on page 3-26
- “High-Throughput Channelizer for FPGA”

More About

- “High-Throughput HDL Algorithms” on page 2-9
- “Digital Signal Processing Design for FPGAs and ASICs”

Block Reference Page Examples

Generate Sine Wave

This example shows how to use the NCO block to generate a sine wave.

The example generates a sine wave with these specifications.

```
F0 = 510;       % Desired output frequency (Hz)
Deltaf = 0.05;  % Frequency resolution (Hz)
SFDR = 90;      % Spurious free dynamic range (dB)
Ts = 1/8000;    % Sample period (s)
phOffd = pi/2;  % Desired phase offset (rad)
```

The frequency resolution of the sine wave depends on the size of the accumulator. Calculate the number of required accumulator bits to achieve the desired frequency resolution. `N` must be an integer value. Use this value to set the accumulator data type **Word length** parameter.

$$N = \text{ceil}(\text{log2}(\frac{1}{T_s \cdot \Delta f}))$$

```
N = ceil(log2(1/(Ts*Deltaf))) %#ok<*NOPTS>
```

```
N =

    18
```

Quantizing the output of the accumulator allows you to achieve better frequency resolution without increasing the lookup table size. Calculate the number of quantized accumulator bits from the spurious free dynamic range specification. Ensure that the **Quantize phase** parameter is selected, and then use this value to set the set the **Number of quantizer accumulator bits** parameter.

$$Q = \text{ceil}(\frac{SFDR - 12}{6})$$

```
Q = ceil((SFDR - 12)/6)
```

```
Q =

    13
```

Calculate the phase increment. Use this value to set the **Phase increment** parameter.

$$phIncr = \text{round}(F_0 \cdot 2^N T_s)$$

```
phIncr = round(F0*2^N*Ts)
```

```
phIncr =

       16712
```

Phase offset and dither are optionally added in the accumulator stage. Calculate the phase offset that the block adds from the desired phase offset of the output wave. Use this value to set the **Phase offset** parameter.

$$phOff = \frac{2^N \cdot PhOff_d}{2\pi}$$

```
phOff = (2^N*phOffd)/(2*pi)
```

```
phOff =

       65536
```

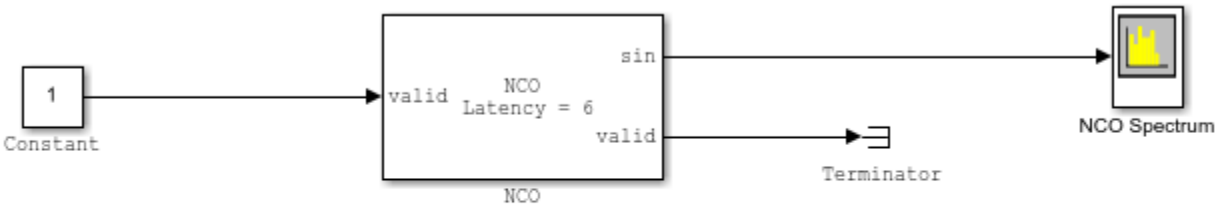
Select the number of dither bits. In general, a good choice for the number of dither bits is the accumulator word length minus the number of quantized accumulator bits. Use this value to set the **Number of dither bits** parameter.

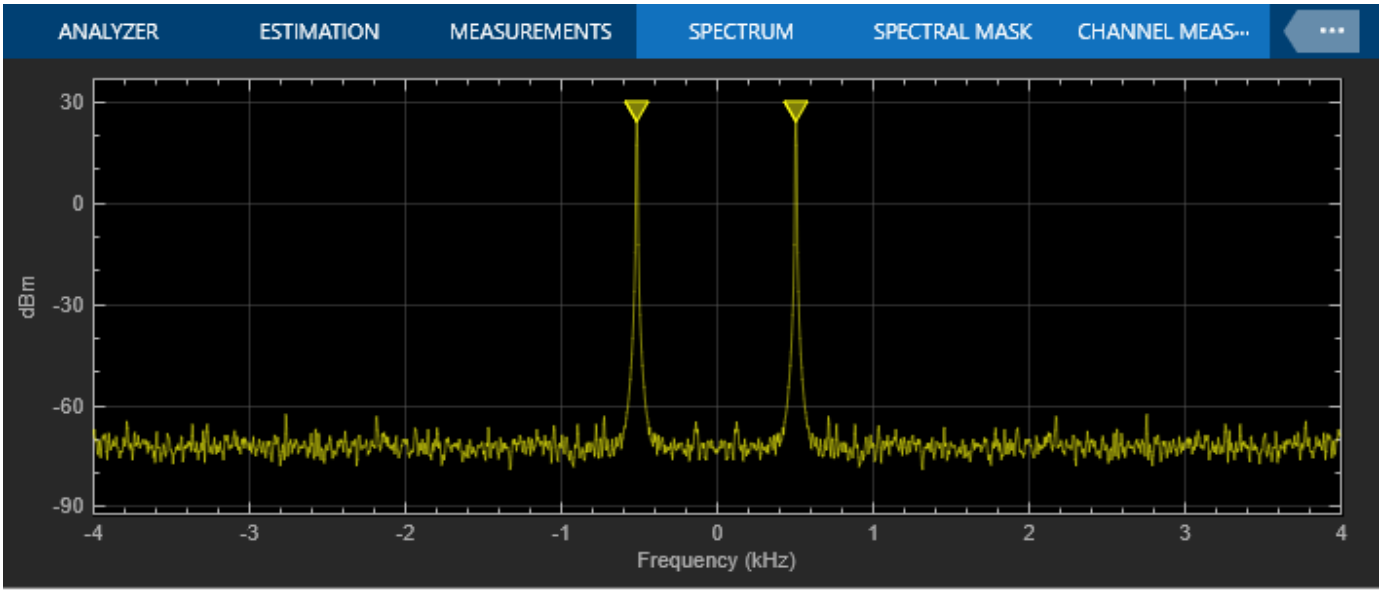
```
ditherBits = N - Q
```

```
ditherBits =

     5
```

Open and simulate the model. The NCO block in the model is configured with the variables calculated in this script. The output word length and fraction length depend on the constraints of your hardware; this example uses a word length of 16 and a fraction length of 14.

```
open_system('GenerateSineWaveHDL')
sim('GenerateSineWaveHDL')
```





Peaks

Channel 1	P1	P2
Frequency (Hz)	5.1042e+2	-5.1042e+2
Power (dBm)	2.3946e+1	2.3946e+1

Stopped VBW = 184.207 mHz RBW = 7.81250 Hz Sample Rate = 8.00000 kHz Frames = 182 T = 30.0000

The Peak Finder results in the Spectrum Analyzer show that the output waveform is generated at 510 Hz. Experiment with the model to observe the effects on the output shown on the Spectrum Analyzer. For example, try turning dithering on and off, and try changing the number of dither bits.

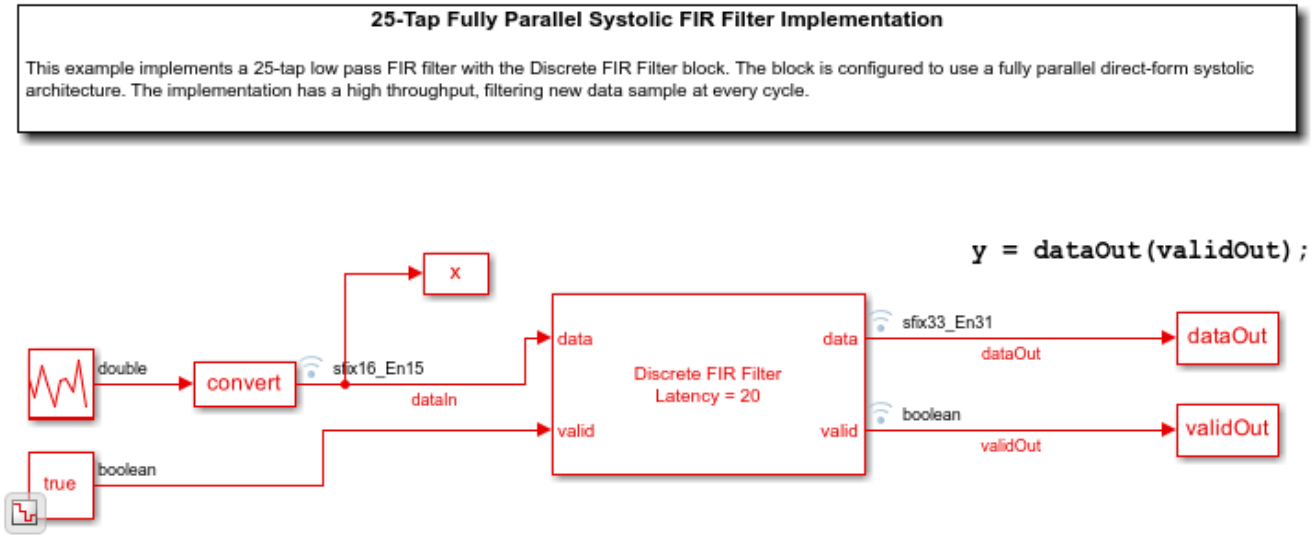
Fully Parallel Systolic FIR Filter Implementation

This example shows how to implement a fully parallel 25-tap lowpass FIR filter by using the Discrete FIR Filter block.

The model filters new data samples at every cycle.

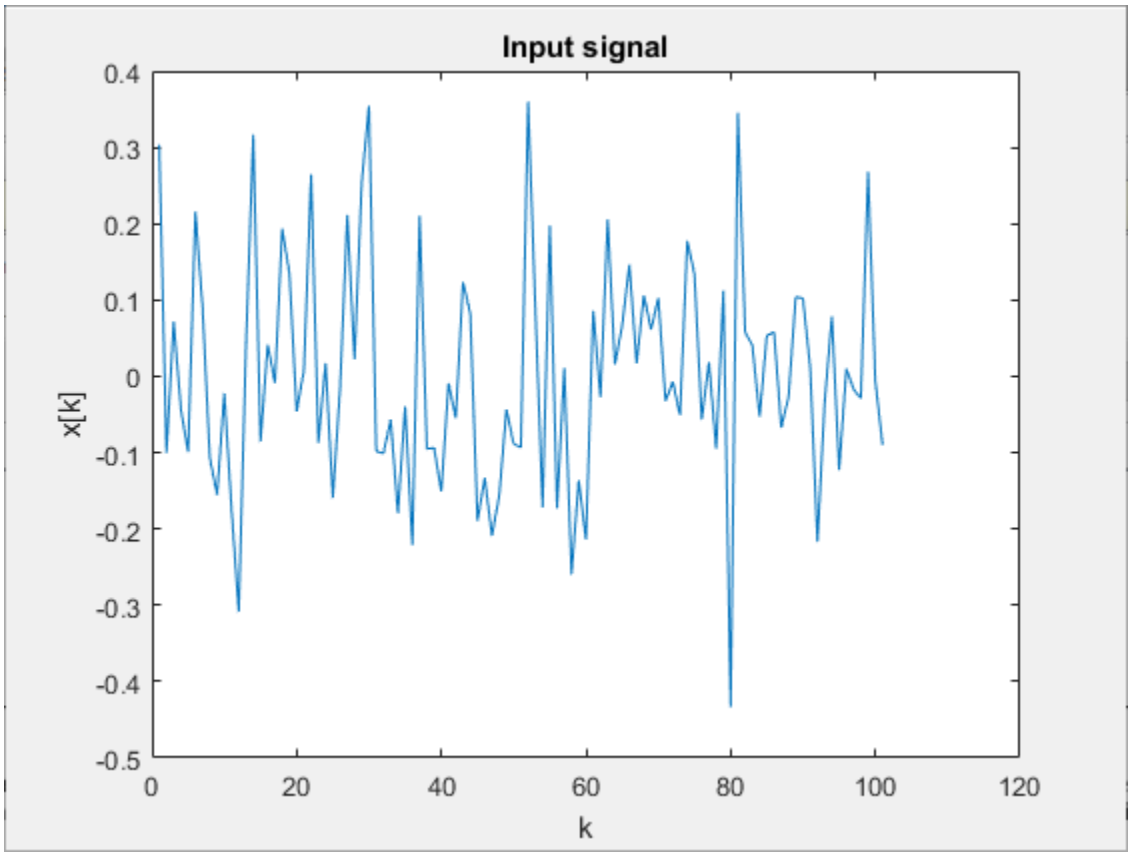
Open Model

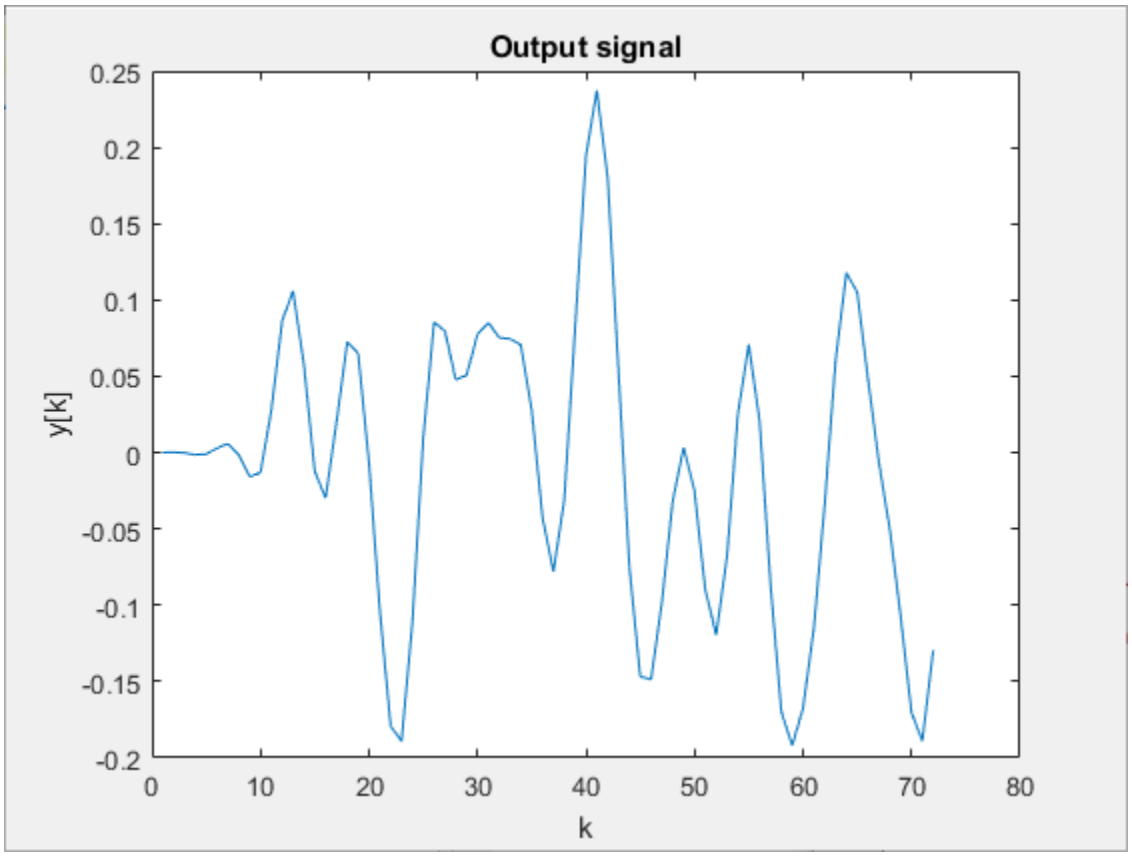
Open the model. Inspect the block parameters. **Filter structure** is set to `Direct form systolic`.



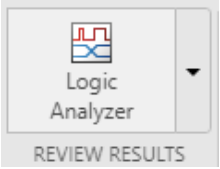
Run Model and Inspect Results

Run the model. Observe the input and output signals in the generated plots.

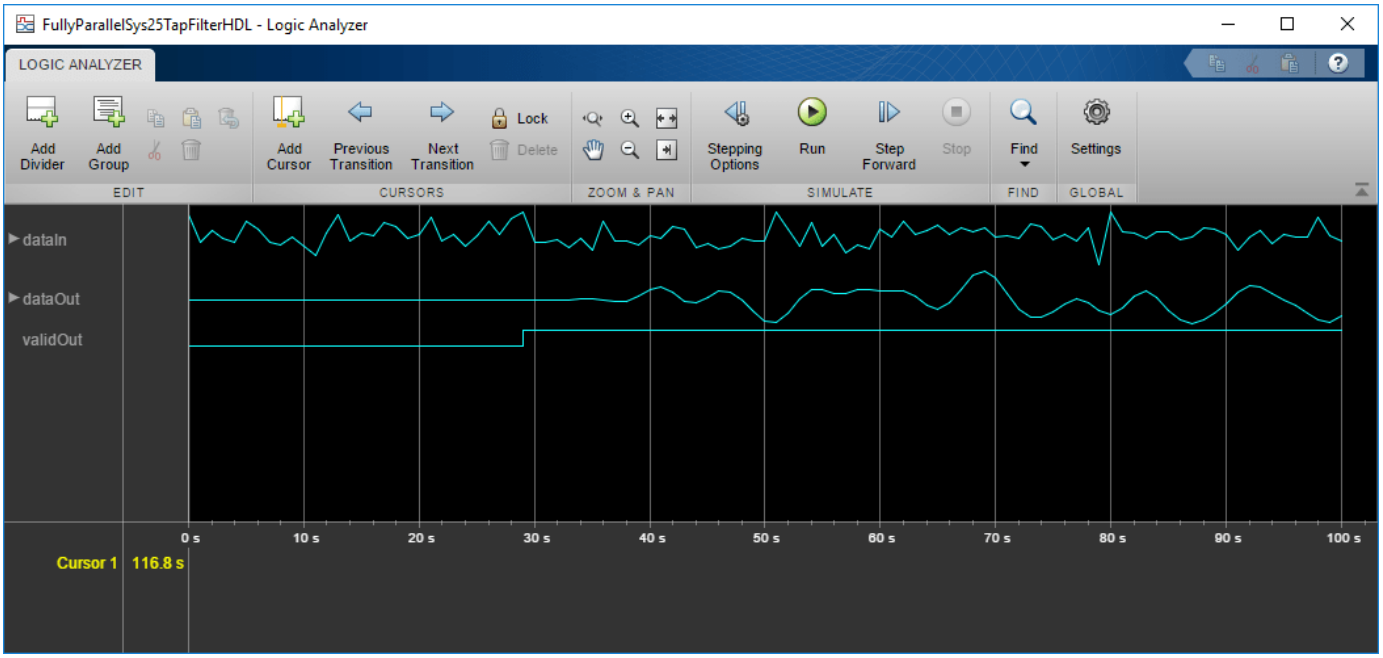

Input signal
x[k]
k
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
0
20
40
60
80
100
120



Use the model toolbar to open the **Logic Analyzer**. If the button is not displayed, expand the **Review Results** app gallery.



Note the pattern of the `validOut` signal.



Generate HDL Code

To generate HDL code from the Discrete FIR Filter block, right-click the block and select **Create Subsystem from Selection**. Then right-click the subsystem and select **HDL Code > Generate HDL Code for Subsystem**.

See Also

Blocks
Discrete FIR Filter

Partly Serial Systolic FIR Filter Implementation

This example shows how to implement a 32-tap lowpass FIR filter using the Discrete FIR Filter block.

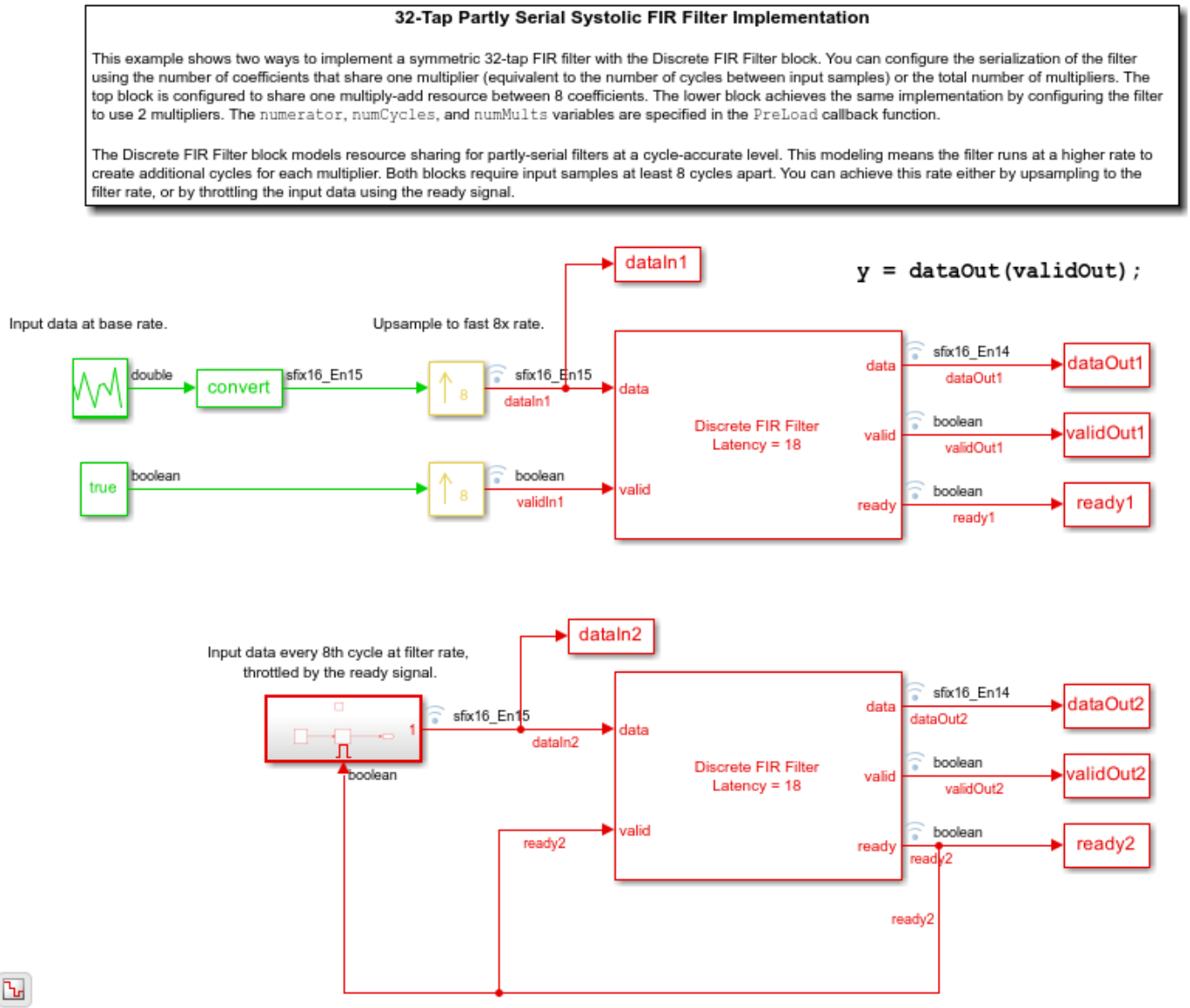
The two filter blocks in the example model implement an identical partly serial 32-tap filter. The top block configures the serial filter by specifying the number of cycles *N* between input samples. This spacing allows each multiplier to be shared by *N* coefficients. The second block is configured to use a certain number of multipliers, *M*. These two configurations result in the same filter implementation. For 32 symmetric coefficients, there are 16 unique coefficients. Therefore the filter shares each of 2 multipliers between 8 coefficients.

The model shows two ways of applying input samples, depending on the rate of the rest of your design.

Open Model

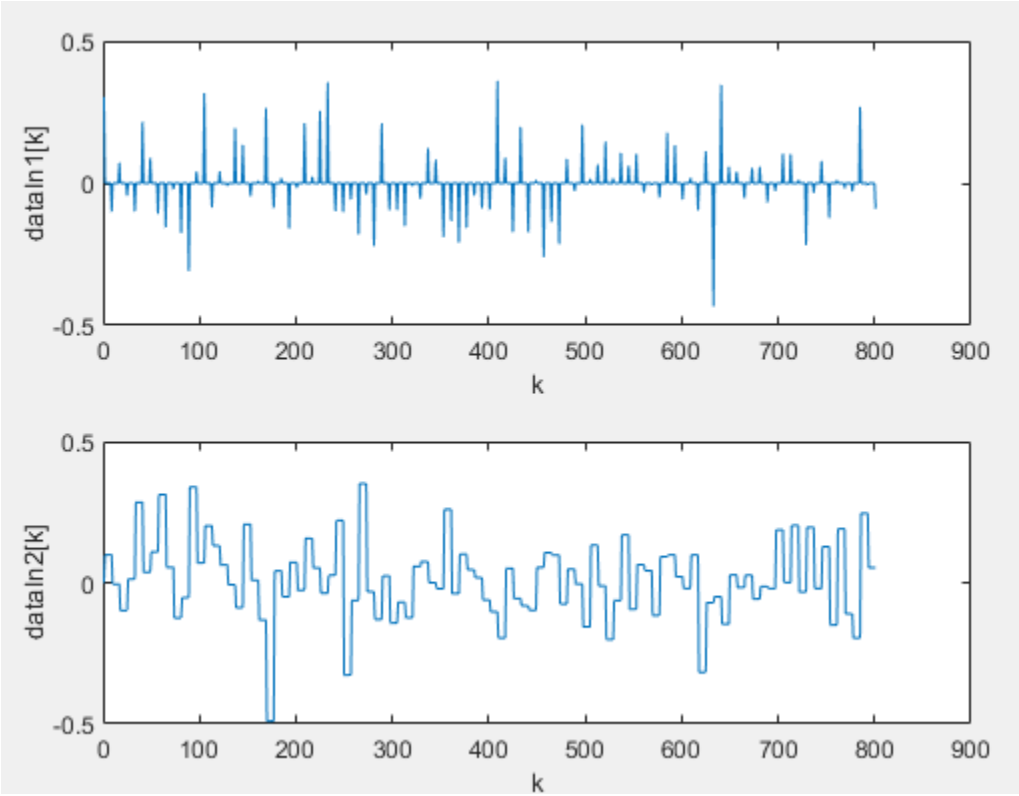
Open the model. Inspect the top block parameters. **Filter structure** is set to `Partly serial systolic` and **Specify serialization factor as** is set to `Minimum number of cycles between valid input samples`. **Number of cycles** is set (to 8) using a variable, `numCycles`. In the lower block, **Specify serialization factor as** is set to `Maximum number of multipliers`. **Number of multipliers** is set (to 2) using a variable. The variables are defined in the `PostLoadFcn` callback function.

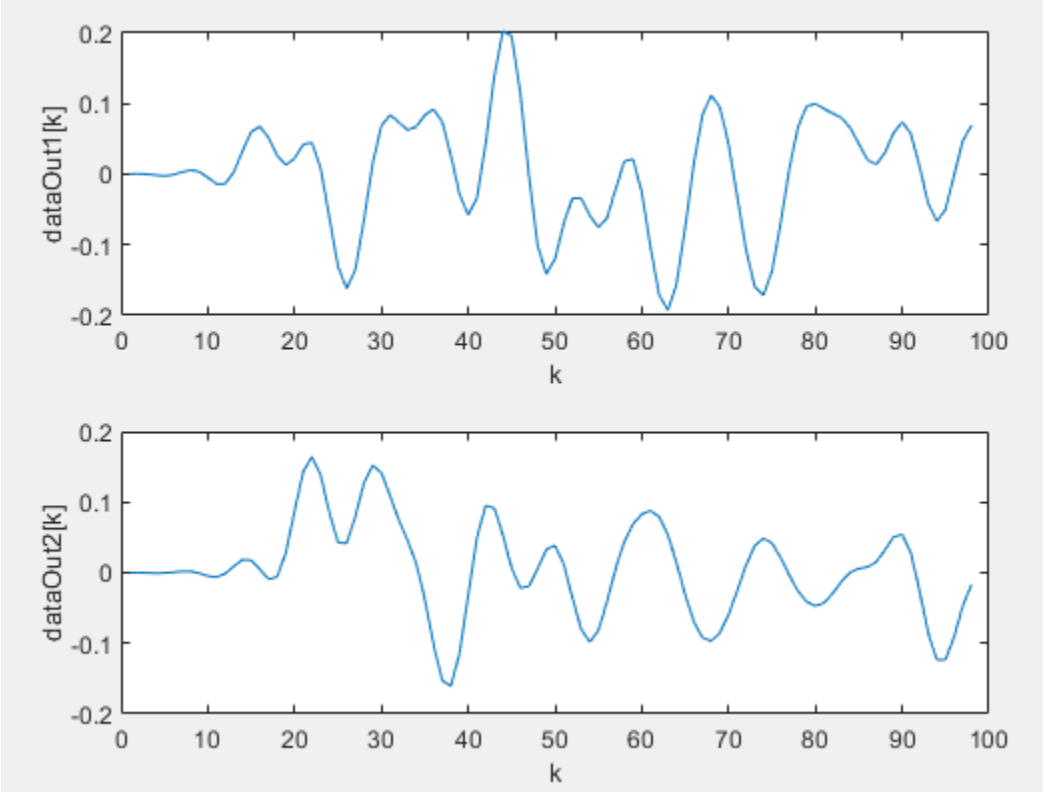
From the color coding, you can see the rate of both filter blocks is the same, while the rate of the generated input samples is different.



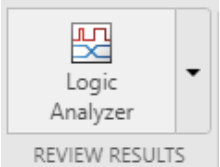
Run Model and Inspect Results

Run the model. Observe the input and output signals in the generated plots. The code to generate the plots is in the `PostSimFcn` callback function.

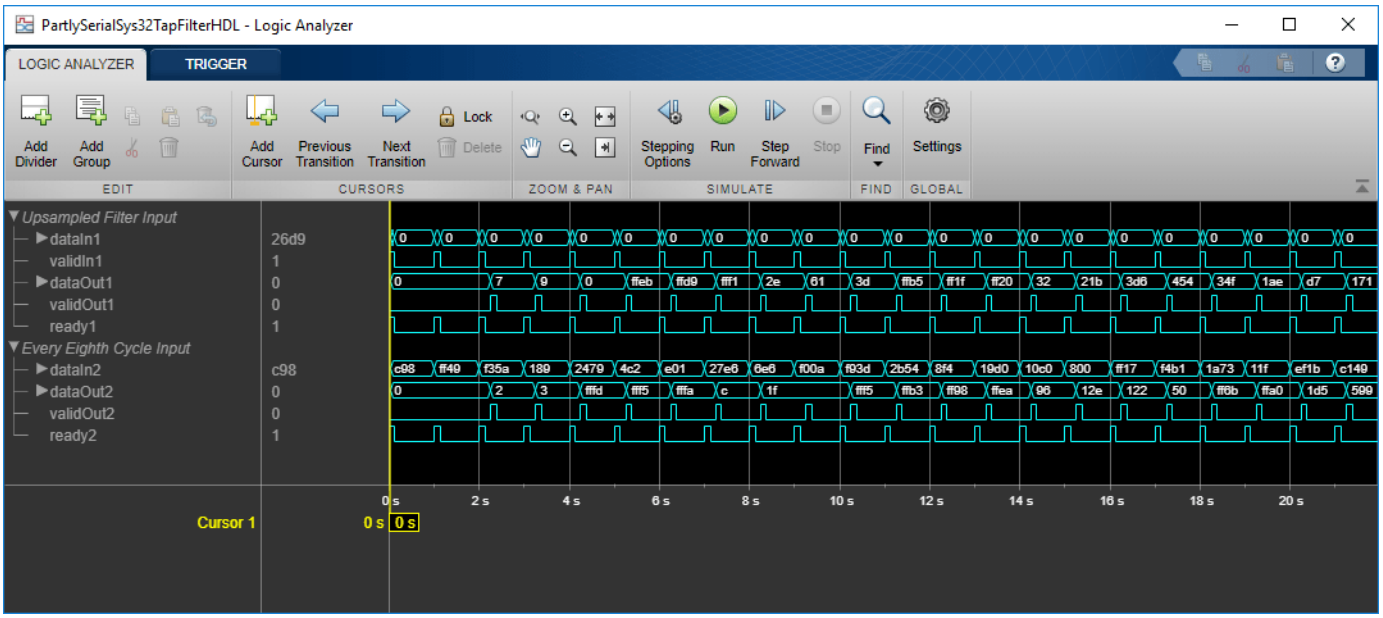

dataIn1[k]
0.5
0
-0.5
0
100
200
300
400
500
600
700
800
900
k
dataIn2[k]
0.5
0
-0.5
0
100
200
300
400
500
600
700
800
900
k


dataOut1[k]
0.2
0.1
0
-0.1
-0.2
0
10
20
30
40
50
60
70
80
90
100
k
dataOut2[k]
0.2
0.1
0
-0.1
-0.2
0
10
20
30
40
50
60
70
80
90
100
k

Use the model toolbar to open the **Logic Analyzer**. If the button is not displayed, expand the **Review Results** app gallery.



Inspect the rising edges of `ready`, `validIn`, and `validOut`.



Generate HDL Code

To generate HDL code from either Discrete FIR Filter block, right-click the block and select **Create Subsystem from Selection**. Then right-click the subsystem and select **HDL Code > Generate HDL Code for Subsystem**. The two blocks generate the same HDL code.

See Also

Blocks

Discrete FIR Filter

Optimize Programmable FIR Filter Resources

This example shows how to use programmable coefficients with the Discrete FIR Filter block and how to optimize hardware resources for programmable filters.

The Discrete FIR Filter block optimizes resource use when the filter coefficients are symmetric or antisymmetric or are zero-valued. To use these optimizations with programmable coefficients, all of the input coefficient vectors must have the same symmetry and zero-valued coefficient locations. Set the **Coefficients prototype** parameter to a representative coefficient vector. The block uses the prototype to optimize the filter by sharing multipliers for symmetric or antisymmetric coefficients, and removing multipliers for zero-valued coefficients.

If your coefficients are unknown or not expected to share symmetry or zero-valued locations, set **Coefficients prototype** to `[]`. When you do so, the block does not optimize multipliers.

This example shows how to set a prototype and specify programmable coefficients for a symmetric filter and a filter with zero-valued coefficients. The example also explains how the block reduces the number of multipliers in the filter in these cases.

This example uses a parallel interface to specify the filter coefficients. The Discrete FIR Filter block also supports using a memory-style interface for the coefficients. When you use the memory-style interface, your coefficient prototype cannot be empty. For an example that uses the memory-style interface, see “Programmable FIR Filter for FPGA”.

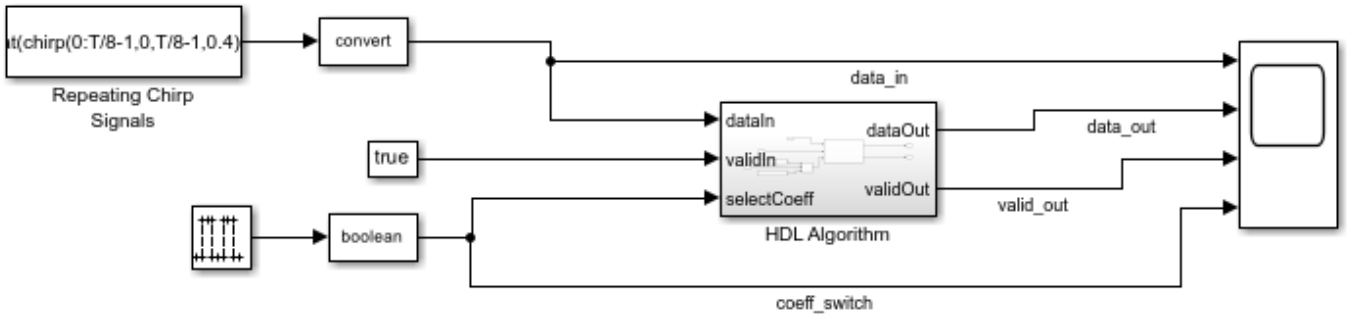
Symmetric Filter Coefficients

Design two FIR filters, one with a lowpass response and the other with the complementary highpass response. Both filters are odd-symmetric and have 43 taps.

```
Fpass = 0.45; % Passband frequency
Fstop = 0.55; % Stopband frequency
Apass = 1;    % Passband attenuation (dB)
Astop = 60;   % Stopband attenuation (dB)

f = fdesign.lowpass('Fp,Fst,Ap,Ast',Fpass,Fstop,Apass,Astop);
Hlp = design(f,'equiripple','FilterStructure','dffir'); % Lowpass
Hhp = firlp2hp(Hlp);                                    % Highpass

hpNumerator = Hlp.Numerator;  %#ok<NASGU>
lpNumerator = Hhp.Numerator;  %#ok<NASGU>
```

The example model shows a filter subsystem with a control signal to switch between two sets of coefficients. The HDL Algorithm subsystem includes a Discrete FIR Filter block and the two sets of coefficients defined by the created workspace variables.

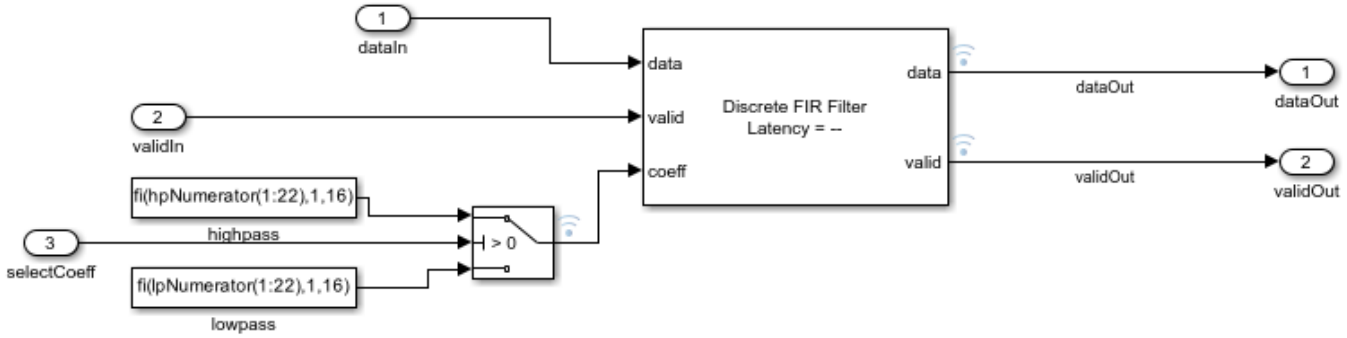
```
modelname = 'ProgFIRHDLOptim';
open_system(modelname);
```



Because two sets of coefficients are symmetric in the same way, you can reduce the number of multipliers in the filter implementation by setting the **Coefficient prototype** parameter of the Discrete FIR Filter block. Set **Coefficient prototype** to either of the coefficient vectors. The example model sets the prototype to `hpNumerator`.

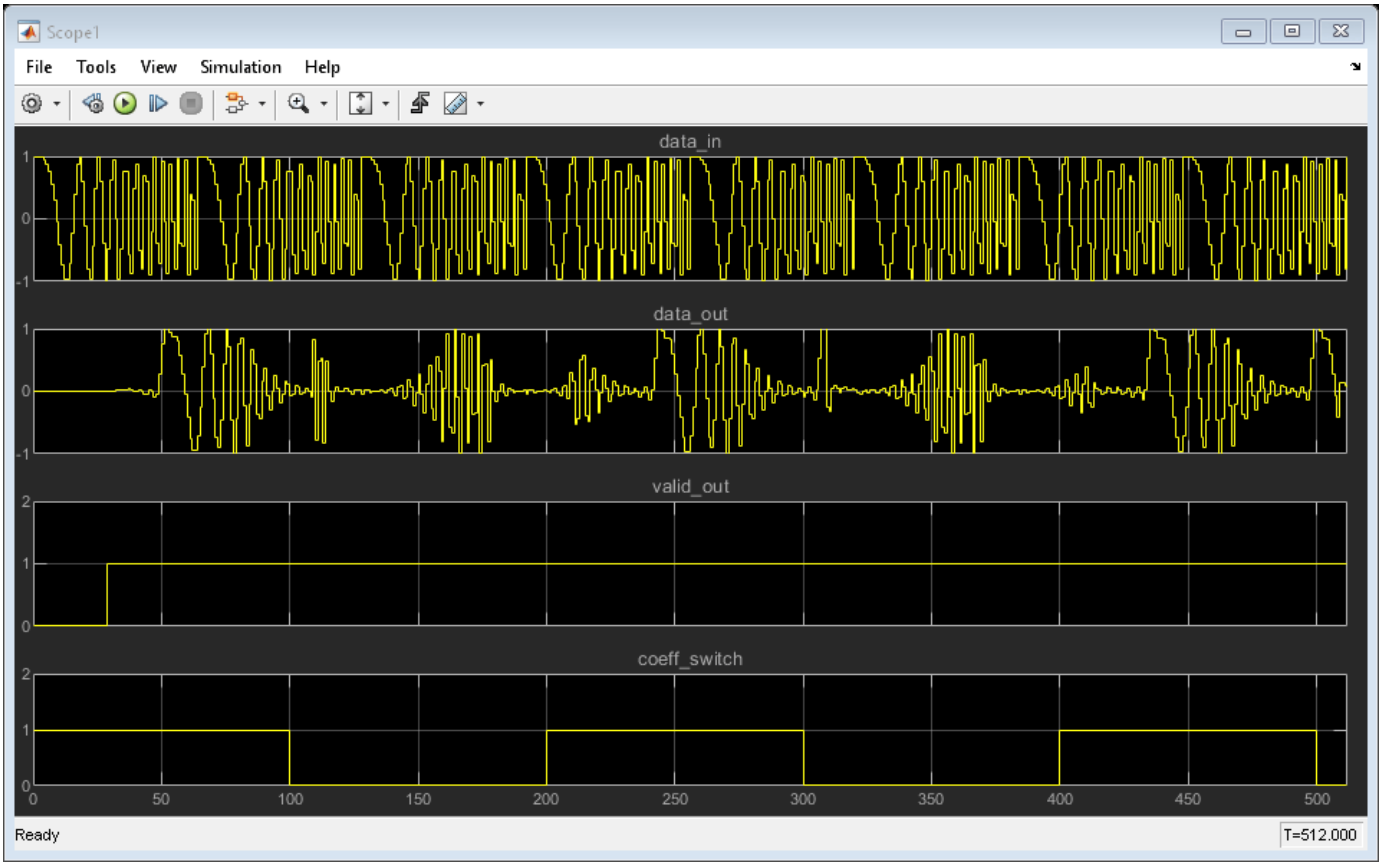
When you use the prototype for symmetric coefficients, provide only the unique coefficients to the **coeff** port. In this case, the filter has 43 odd-symmetric coefficients, so the input port expects the first half of the coefficients, that is, 22 values.

```
open_system('ProgFIRHDLOptim/HDL Algorithm');
```



The model switches coefficients every 100 cycles. The filtered output data shows the effect of the lowpass and highpass coefficients.

```
T = 512;
sim(modelname);
```



The model is configured to output the resource report from HDL code generation. This feature enables you to see the number of multipliers in the filter implementation. Because the block shares multipliers for symmetric coefficients, the filter implementation uses 22 multipliers rather than 43.

Multipliers	22
Adders/Subtractors	44
Registers	275
Total 1-Bit Registers	5117
RAMs	0
Multiplexers	266
I/O Bits	65
Static Shift operators	0
Dynamic Shift operators	0

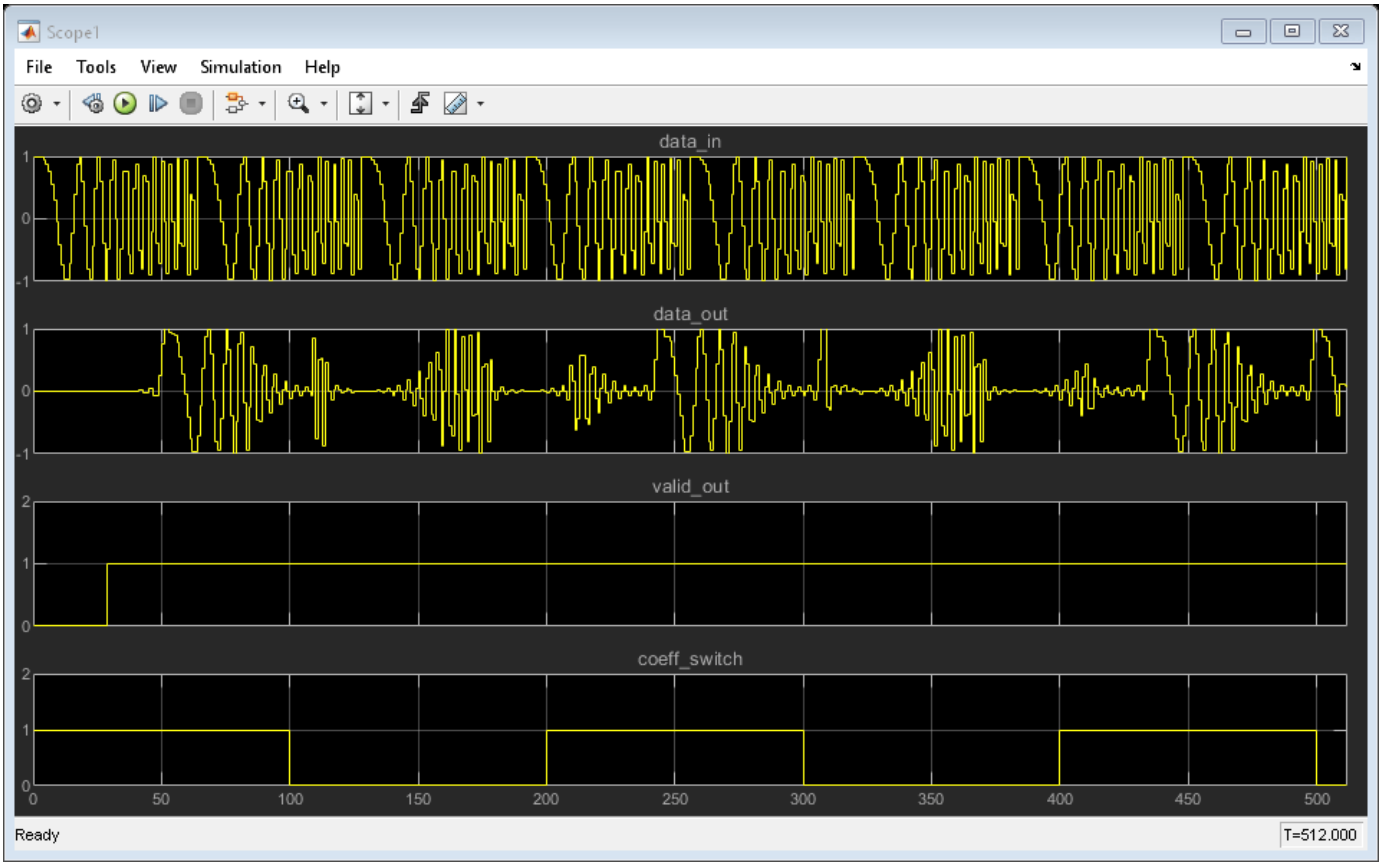
Zero-Valued Filter Coefficients

Design two halfband FIR filters, one with a lowpass response and the other with the complementary highpass response. Both filters have 43 symmetric taps, where every second tap is zero. Set the

Coefficient prototype parameter to either of the coefficient vectors. Changing the workspace value of `hpNumerator` updates the prototype in the block.

Similarly, specify 22 coefficients at the input port. Although no multipliers exist for the zero-valued coefficients, for the block to maintain the correct alignment of the coefficients, you must specify the zero-valued coefficients at the port.

```
N = 42;
f = fdesign.halfband('N,Ast',N,Astop);
Hlp = design(f,'equiripple','FilterStructure','dffir'); % Lowpass
Hhp = firlp2hp(Hlp);                                    % Highpass

hpNumerator = Hlp.Numerator;
lpNumerator = Hhp.Numerator;

sim(modelname);
```



The model is configured to output the resource report from HDL code generation. This feature enables you to see the number of multipliers in the filter implementation. In this case, because the filter optimizes for symmetry and zero-valued coefficients, the implementation uses 12 multipliers rather than 43.

Multipliers	12
Adders/Subtractors	24
Registers	265
Total 1-Bit Registers	4117
RAMs	0
Multiplexers	186
I/O Bits	65
Static Shift operators	0
Dynamic Shift operators	0

See Also

Blocks
Discrete FIR Filter

FIR Decimation for FPGA

This example shows how to decimate streaming samples using a hardware-friendly polyphase FIR filter. It also shows the differences between the `dsp.FIRDecimator` object and the hardware-friendly streaming interface of the FIR Decimator block.

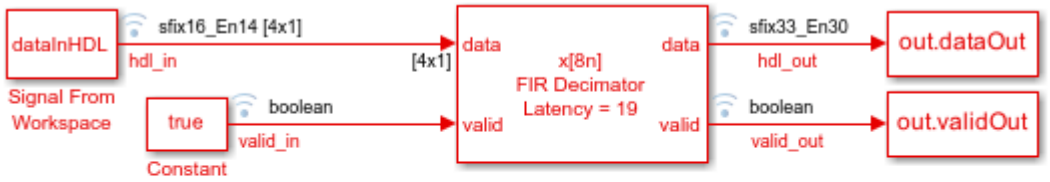
Generate an input sine wave in MATLAB®. The example model imports this signal from the MATLAB workspace. Choose a decimation factor, coefficients, and the input vector size for the HDL-optimized block.

```
T = 512;
waveGen = dsp.SineWave(0.9,100,0,'SamplesPerFrame',T);
dataIn = fi(waveGen()+0.05,1,16,14);

decimFactor=8;
coeffs = firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0]);
inputVecSize=4;
```

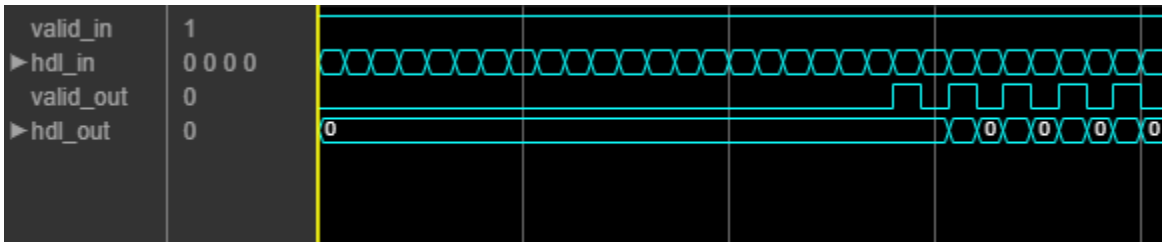
The example model imports the input signal from the MATLAB workspace and applies it as vectors to the FIR Decimator block. The model returns the output from the block to the MATLAB workspace for comparison.

The FIR Decimator block applies samples to the filter in a different phase than the `dsp.FIRDecimator object`. To match the numerical behavior, apply `decimFactor` - 1 zeros to the FIR Decimator block before the start of the data samples.

```
dataInHDL = [zeros(decimFactor-1,1);dataIn];

modelName = 'FIRDecimHDL';
open_system(modelName);
set_param(modelName,'SimulationCommand','Update');
```



The FIR Decimator block accepts a vector of `inputVecSize`-by-1 samples and returns scalar output. The block performs single-rate processing and indicates valid samples in the output signal by setting the output **valid** signal to 1. The Simulink sample time is `inputVecSize` for the whole model. The block sets the output **valid** signal to 1 every `decimFactor/inputVecSize` samples.




```
out = sim('FIRDecimHDL');
```

Create a `dsp.FIRDecimator` System object™ and generate reference data to compare against the HDL model output. The object can accept any input vector size that is a multiple of the decimation factor, so you can calculate the output with one call to the object. If you change the decimation factor to a value that is not a factor of `T`, you must also adjust the value of `T` (size of `dataIn`).

The block and the object are both configured to use full-precision internal data types. The object computes a different output data type in this mode. This difference means that the output might not match if the internal values saturate the data types.

```
decimRef = dsp.FIRDecimator(decimFactor,'Numerator',coeffs);
refDataOut = decimRef(dataIn);
```

For the output of the FIR Decimator block, select data samples where the output valid signal was `1`, and compare them with the samples returned from the `dsp.FIRDecimator` object.

```
hdlVec = out.dataOut(out.validOut);
refVec = refDataOut(1:size(hdlVec,1));
errVec = hdlVec - refVec;
maxErr = max(abs(errVec));
fprintf('\nFIR decimator versus reference: Maximum error out of %d values is %d\n',length(hdlVec
```

```
FIR decimator versus reference: Maximum error out of 55 values is 0
```

See Also

Blocks
FIR Decimator

Implement atan2 Function for HDL

This example shows how to use the Complex to Magnitude-Angle block to implement the `atan2` function in hardware.

This example model compares the output of Complex to Magnitude-Angle block with the `atan2` function implemented using Trigonometric Function block.

HDL Counter 1 and HDL Counter 2 blocks generate the real and imaginary parts, respectively, of the complex number.

Real-Imag to Complex block constructs the complex output from real and imaginary inputs.

Trigonometric Function block with function parameter set as `atan2` is used to generate the reference output angle. This block uses the CORDIC approximation to calculate the angle.

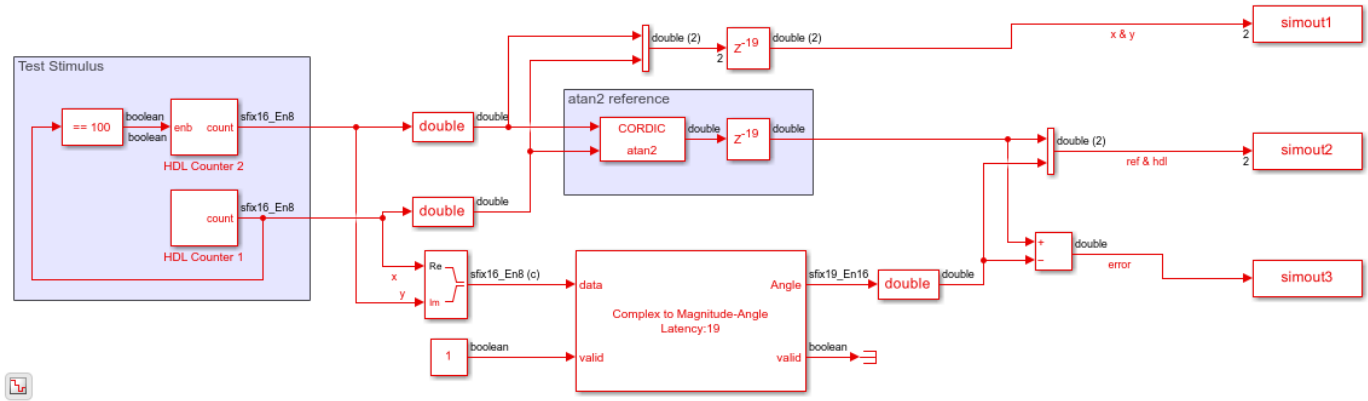
The Complex to Magnitude-Angle block is configured to return the angle in radians produces the angle of the complex input as an output. This block also models the latency of the hardware implementation.

To align the data for comparison, the reference data path includes a delay block with the same latency.

The Complex to Magnitude-Angle block supports HDL code generation, if you add it to a subsystem.

Run the Simulink™ model.

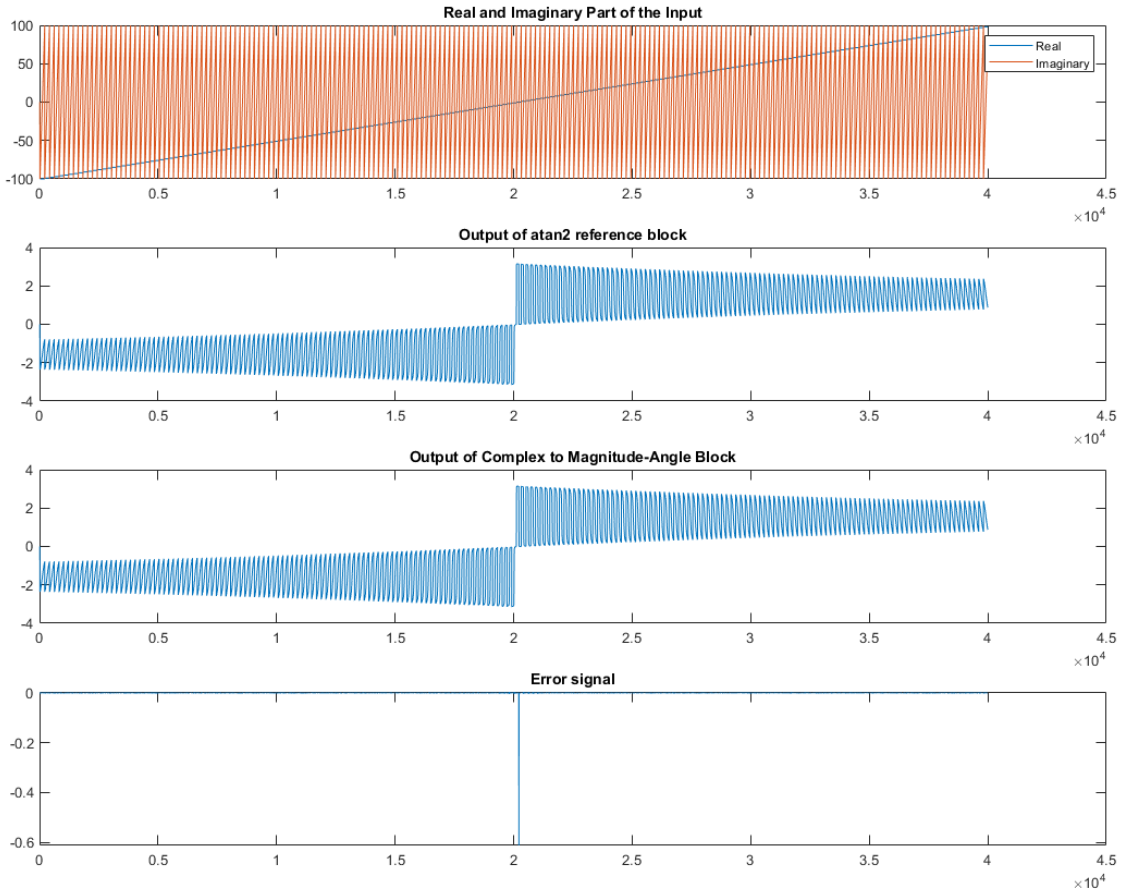
```
modelname = 'HDLatan2Example';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
out = sim(modelname);
```



Compare the outputs of Complex to Magnitude-Angle block against the atan2 function block.

```
figure('units','normalized','outerposition',[0 0 1 1])
subplot(4,1,1)
plot(simout1(:,1))
```

```
hold on;
plot(simout1(:,2))
hold off;
legend('Real','Imaginary')
title('Real and Imaginary Part of the Input')

subplot(4,1,2)
plot(simout2(:,1))
title('Output of atan2 reference block')

subplot(4,1,3)
plot(simout2(:,2))
title('Output of Complex to Magnitude-Angle Block')

subplot(4,1,4)
plot(simout3)
title('Error signal')
```



See Also

Blocks
Complex to Magnitude-Angle

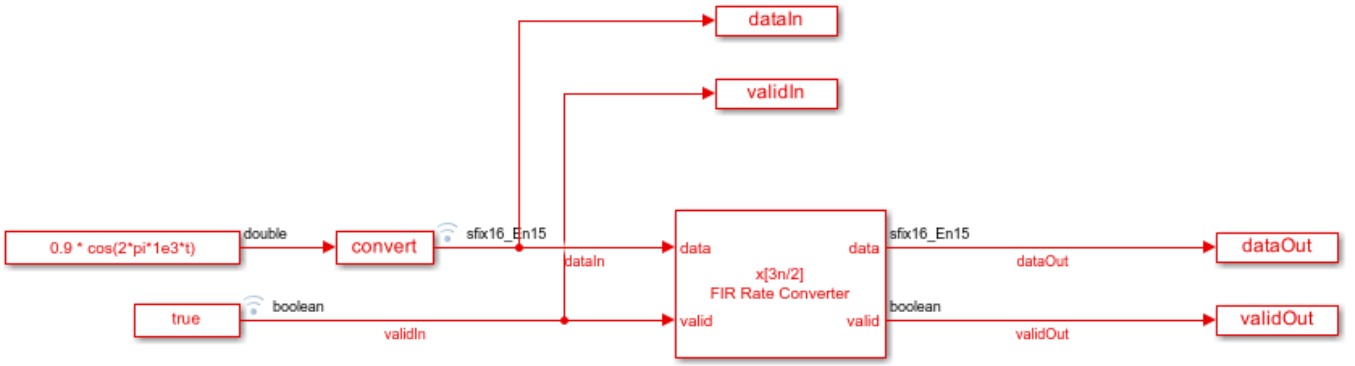
Functions
`atan2`

Downsample a Signal

Convert a signal from 48 kHz to 32 kHz using the FIR Rate Converter block.

The source is a cosine input signal, sampled at 48kHz. The model passes a new data sample into the block on every time step by holding the input `valid` port `true`. After resampling, the output `valid` signal is `true` on only 2/3 of the time steps.

Open the Model

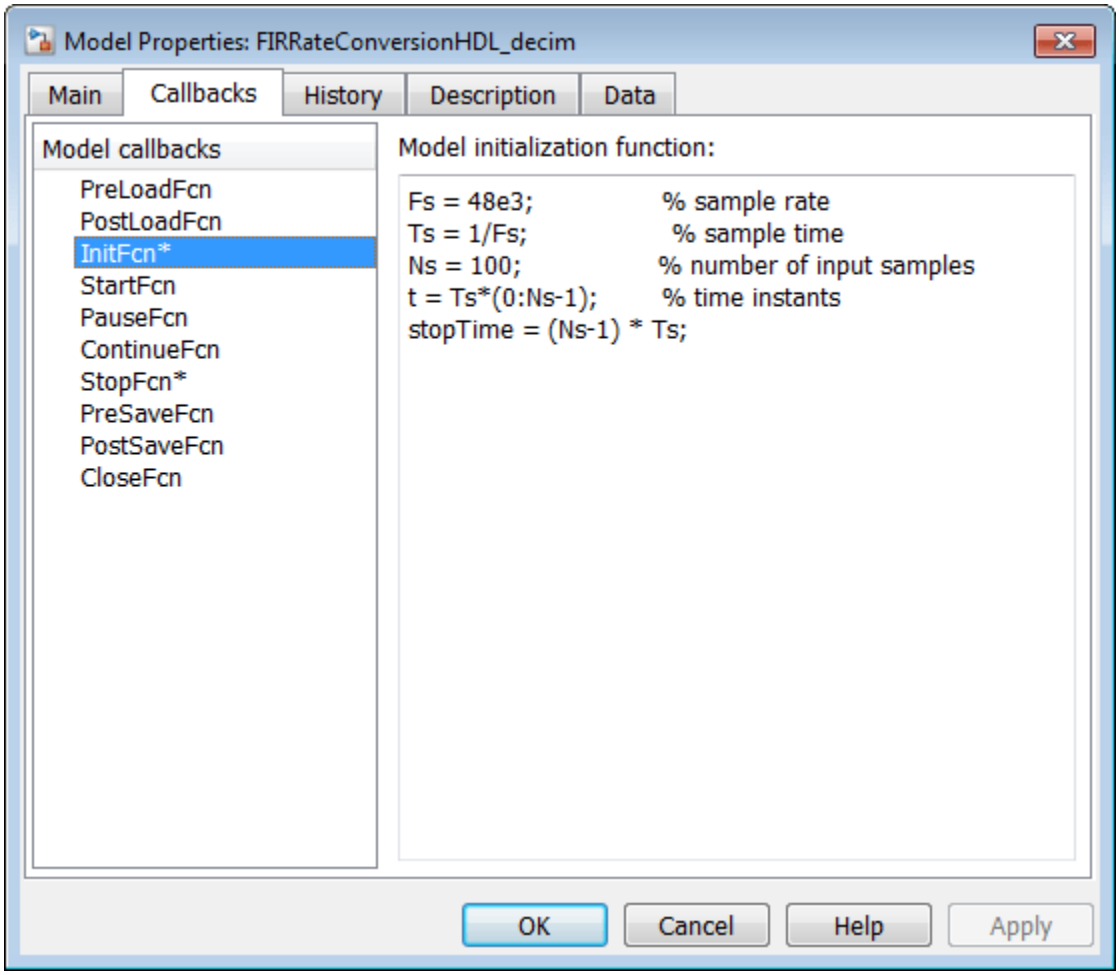


FIR Rate Converter - Decimation

This example changes the sample rate of a signal from 48kHz to 32kHz. The rate change factor is 2/3. New data is passed into the rate converter on every time step by setting the input valid=true, and 2 out of every 3 outputs are valid.

Configure the Model

Define the data rate parameters in the `InitFcn` callback.

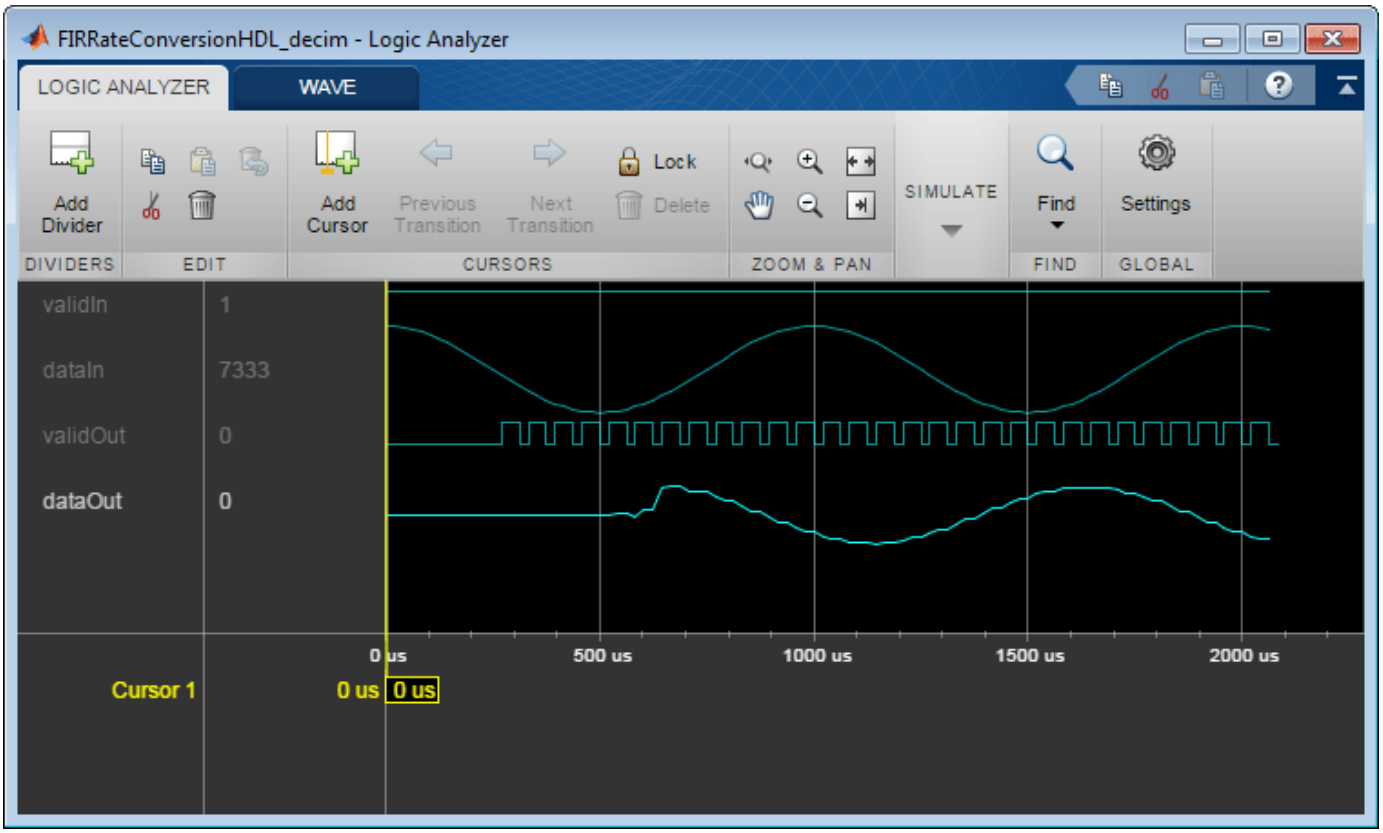


Configure the FIR Rate Converter block. Use the default interpolation factor of 2 and decimation factor of 3. Use the `firmpm` function to design an equiripple FIR filter. In the **Data Types** group, set the **Coefficients** data type to `fixdt(1,16,15)` to accommodate the filter you designed.

Run the Model and Display Results

Run the model. Use the **Logic Analyzer** to view the input and output signals of the block. The blue icon in the model indicates streamed signals. Launch the Logic Analyzer from the model's toolstrip.

In the Logic Analyzer, note the pattern of `validIn` and the resulting `validOut` signal.



Generate HDL Code

To generate HDL code from the FIR Rate Converter block, right-click the block and select **Create Subsystem from Selection**. Then right-click the subsystem and select **HDL Code > Generate HDL Code for Subsystem**.

Control Data Rate Using Ready Signal

This example shows how to use a backpressure signal to control an upstream data source for an interpolator design.

It also shows how to regulate the output data pattern by using a FIFO and a request signal.

Hardware algorithms operate on streaming data to make efficient use of hardware resources, such as memory for storing samples. The blocks have limited storage for incoming samples, and take time to process each sample. In an interpolator design, there must also be cycles available to accommodate the interpolated output samples.

To ensure that a block does not receive input samples when it cannot accept them, use one of these options.

- Space the input samples with invalid cycles in between to accommodate the algorithm. Regular input spacing can also enable internal resource sharing. For details of this type of input rate management, see the “Implement Digital Upconverter for FPGA” on page 1-84 example.
- Use a backpressure signal to stop upstream blocks from providing data when downstream blocks cannot accept it. DSP HDL Toolbox™ blocks provide the `ready` output signal to indicate when the algorithm can and cannot accept new input samples. This example shows how to use this signal to implement backpressure in your system.

The example model uses a FIR Rate Converter block to upsample a signal from 40 MHz to 100 MHz. To achieve this rate change, the block must return 2 or 3 output samples for each input sample. The generation of these extra samples means that the block cannot accept new input samples every cycle, and that the output rate can be nonuniform. This example shows how to use the `ready` output signal to control the upstream data source, and how to use a FIFO to regulate the output data rate.

Configure Model

Define the data rate parameters. The Simulink® model uses these values to configure the FIR Rate Converter block and the FIFO control signals. Use the `firpm` function to design an equiripple FIR filter.

```
FsIn = 40e6;                % input signal sample rate
TsIn = 1/FsIn;              % input signal sample time
NsIn = 100;                 % number of input samples
t = TsIn*(0:NsIn-1);        % input time instants

stopTime = (NsIn-1) * TsIn;

Ts = 1/200e6; % system clock rate; twice the output sample rate

coeffs = firpm(70, [0,.15,.25,1], [1,1,0,0]);
interpFactorL = 5;
decimFactorM = 2;
```

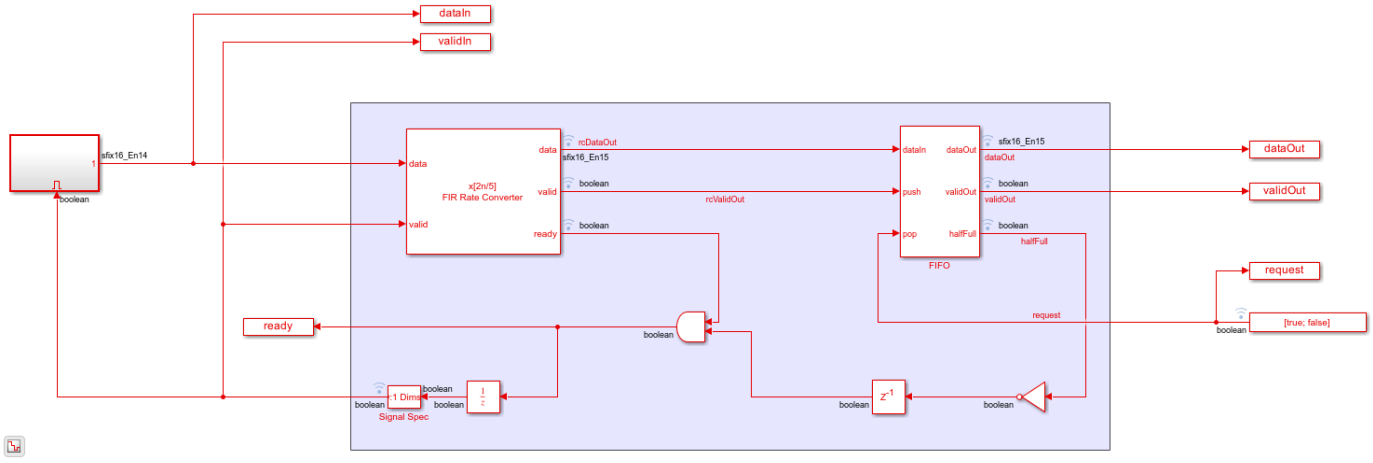
The output FIFO depth must be at least two times the delay around the feedback loop in cycles. This delay is equal to the number of cycles between the `request` signal going to 1 (`true`) to pop the FIFO (enabling a new input to the FIR Rate Converter block) and the resulting output `valid` from the FIR Rate Converter block.

The latency of the block is a pipelined adder tree for the filter coefficients, plus 7 cycles. When interpolating, each input sample can yield up to `ceil(L/M)` output samples. The feedback loop also includes one cycle for the FIFO to update, one cycle for `ready` computation, and one cycle for the register on the feedback path.

```
delayLineLength = ceil(length(coeffs)/interpFactorL);
rateConverterLatency = ceil(log2(delayLineLength)) + 7;
feedbackLoopLatency = rateConverterLatency + ceil(interpFactorL/decimFactorM) + 3;
fifoDepth = 2*feedbackLoopLatency;
```

In the model, the FIR Rate Converter block and the FIFO subsystem are configured by using the workspace variables already defined. The FIR Rate Converter block has the optional **ready** port enabled and the coefficient data type is set to `fixdt(1,16,15)`.

```
modelName = 'FIRRateConversionHDL_interpReadyRequest';
open_system(modelName);
set_param(modelName,'SimulationCommand','Update');
```



To control the input data rate, the block sets the `ready` output signal to 1 (`true`) when the block can accept a new input sample on the next time step. Upstream blocks must only apply valid input data when the `ready` signal is 1 (`true`). This example model combines the `ready` signal with a signal that indicates when the output FIFO is less than half full. The model connects the combined signal as an enable to a waveform source that generates one input sample at a time. This input data rate control ensures that data is only applied when the block is ready to accept it, and that the output FIFO will not overflow.

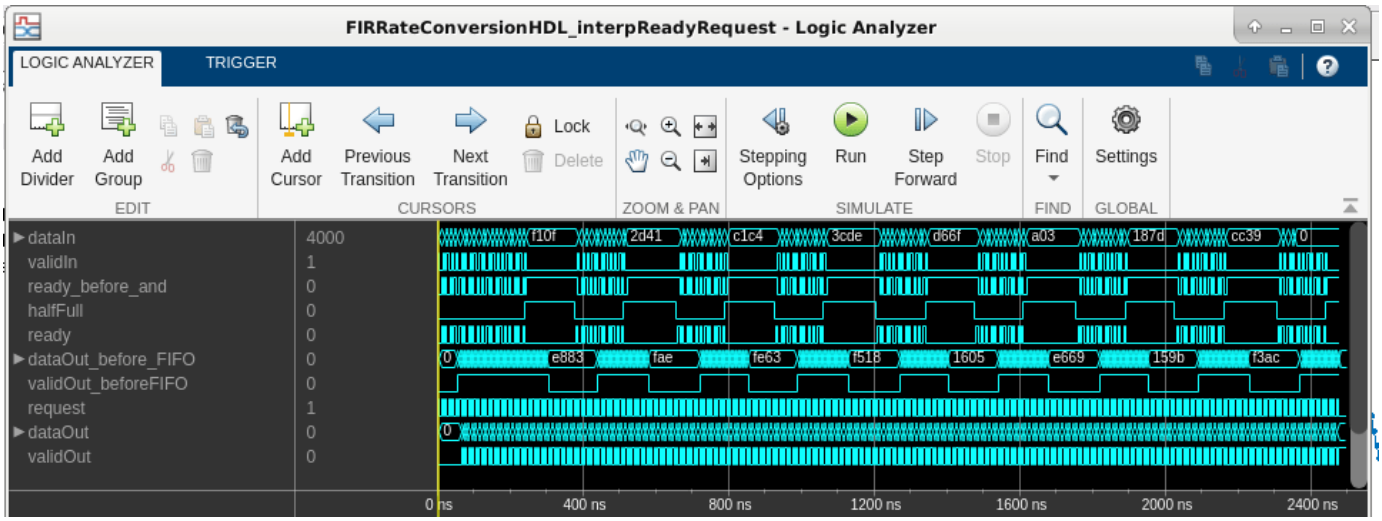
To control the output data rate, the model implements a FIFO for the output samples. The model connects a toggling signal to the FIFO pop port. When you use a system clock of 200MHz and return output samples from the FIFO every second time step, the output data rate is a uniform 100 MHz.

Run Model and Display Results

Open the **Logic Analyzer** to view the input and output signals of the block. The blue icon on signals in the model indicates those signals are logged and available to view in the **Logic Analyzer**.

The waveform shows how the `ready_before_and` signal toggles when the block is processing input samples and cannot accept new data. The FIFO `halffull` signal also turns off the combined `ready` signal to the input generator. The `validIn` signal responds to the combined `ready` signal to avoid sending new data when either the block or the FIFO does not have room. The waveform shows that

the `request` signal is high every second cycle, which generates the final `dataOut` and `validOut` signals from the FIFO.



Generate HDL Code

You must have the HDL Coder™ product to generate HDL code. To generate HDL code from the FIR Rate Converter block and the FIFO and ready logic, create a subsystem from the area shown in the model. Then right-click the subsystem and select **HDL Code > Generate HDL Code for Subsystem**.

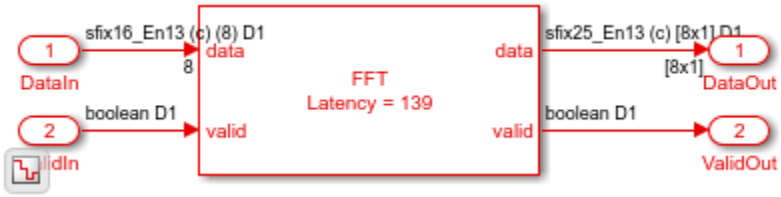
Automatic Delay Matching for the Latency of FFT Block

This example shows how to programmatically obtain the latency of an FFT block in a model. You can use the latency value for delay matching of parallel data paths.

Be cautious with delay matching for large signals or long latencies, since it adds memory to your hardware implementation. Alternatively, if the signal does not change within a frame, use the valid or frame control signals to align the signal with the output of the FFT block.

Open a model that contains an FFT or an IFFT block, such as the model from the “Implement FFT Algorithm for FPGA” example.

```
modelname = 'FFTHDLOptimizedExample_Streaming';
load_system(modelname);
set_param(modelname,'SimulationCommand','Update');
open_system([modelname '/FFT Streaming']);
```



This example includes a `ffthdlLatency` function that calculates the latency of the block for its current parameters. Call the function with the block pointer and input vector size. You can use the Simulink path to the block, or select the block in the model to obtain a pointer, `gcb`. In this model, the input signal is a vector of 8 samples.

```
latency = ffthdlLatency([modelname '/FFT Streaming/FFT'],8)
```

```
latency =

   139
```

The function copies the parameters from the block pointer and creates a System object™ with the same settings as the block. Then it calls the `getLatency` function on the object. See `getLatency`.

```
function lat = ffthdlLatency(block, vectorsize)

% default vector size = 1
if nargin == 1
    vectorsize = 1;
end

fftlen = evalin('base',get_param(block, 'FFTLength'));
arch = get_param(block, 'Architecture');
bri = strcmpi(get_param(block, 'BitReversedInput'), 'on');
bro = strcmpi(get_param(block, 'BitReversedOutput'), 'on');

fftobj = dsphdl.FFT('FFTLength',fftlen, ...
```

```
    'Architecture', arch, ...
    'BitReversedInput', bri, ...
    'BitReversedOutput', bro);

lat = getLatency(fftobj, fftlen, vectorsize);

end
```

See Also

Blocks
FFT | IFFT

Implement CIC Interpolator Filter for HDL

This example shows how to use the CIC Interpolator block to filter and upsample data. This block supports scalar and vector inputs. To work with scalar and vector inputs separately, this example uses two Simulink® models. You can generate HDL code from the subsystems in these Simulink models.

Set Up Input Data Parameters

Set up these workspace variables to configure the CIC Interpolator block. This block supports fixed and variable interpolation rates for scalar inputs and only a fixed interpolation rate for vector inputs. The example runs the `HDLCICInterpolatorModel.slx` model when you set the `scalar` value to `true` and runs the `HDLCICInterpolatorModelVectorSupport.slx` model when you set the `scalar` value to `false`. For scalar inputs, choose a range for the input `varRValue` values and set the interpolation factor value, `R`, to the maximum expected interpolation factor. For vector inputs, the input data must be a column vector of size 1 to 32.

```
R = 8;          % interpolation factor
M = 1;          % differential delay
N = 3;          % number of sections
scalar = true; % true for scalar; false for vector

if scalar
    varRValue = [2,4,R];
    vecSize = 1;
    modelname = "HDLCICInterpolatorModel";
else
    varRValue = R;       %#ok
    vecSize = 1;
    modelname = "HDLCICInterpolatorModelVectorSupport";
end

numFrames = length(varRValue);
dataSamples = cell(1,numFrames);
varRtemp = cell(1,numFrames);
cicFcnOutput = [];
WL = 0;                  % word length
FL = 0;                  % fraction length
```

Generate Reference Output from `dsp.CICInterpolator` System Object

To generate reference output data for comparison, apply the samples to the `dsp.CICInterpolator` System object™. This System object does not support variable interpolation rates, so you must create and release the object for each change in the interpolation factor.

```
for i = 1:numFrames
    framesize = 4;
    dataSamples{i} = fi(randn(framesize,1),1,16,8);
    varRtemp{i} = fi(varRValue(i)*ones(framesize,1),0,12,0);
    obj = dsp.CICInterpolator("DifferentialDelay",M,"NumSections",N, ...
                              "InterpolationFactor",varRValue(i));
    cicOut = step(obj,dataSamples{i}).';
    WL = max([WL,cicOut.WordLength]);
    FL = max([FL,cicOut.FractionLength]);
    cicFcnOutput = [fi(cicFcnOutput,1,WL,FL),cicOut];
    release(obj);
end
```

Convert Input to Stream of Samples and Import to Simulink Model

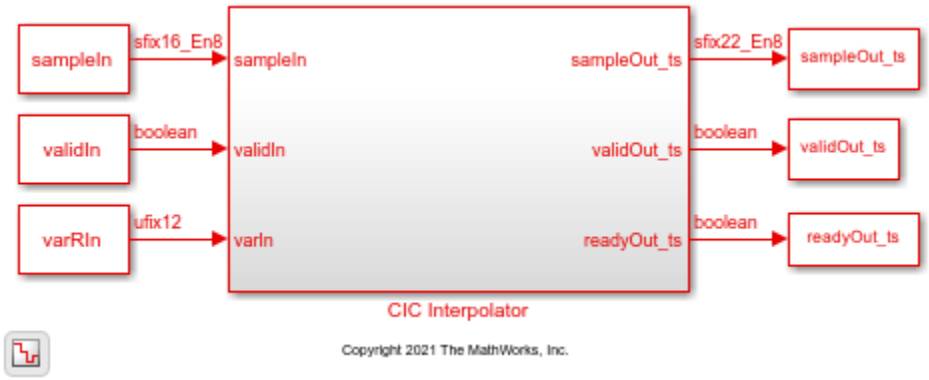
Generate a stream of samples by converting frames to samples. Provide those samples (`sampleIn`) as input to the Simulink model along with the corresponding valid signal (`validIn`) and interpolation rate (`varRIn`). The latency of the block for scalar and vector inputs is calculated based on the type of input and the number of sections, `N`. For more information about latency, see the **"Gain correction"** parameter. To flush remaining data before changing the interpolation rate, run the model by inserting the required number of idle cycles after each frame using the `idlecyclesbetweenframes` value.

```
idlecyclesbetweensamples = R; % upsampling the inputs by factor R
idlecyclesbetweenframes = floor((vecSize-1)*(N/vecSize)) + 1 + N ...
                          + (2+(vecSize+1)*N) + 9;

sampleIn = [];
validIn = [];
varRIn = [];
len = 0;

for ij = 1:numFrames

    dataInFrame =  reshape([0; dataSamples{ij}],vecSize,[]);
    len = size(dataInFrame,2);

    data = [];
    valid= [];
    varR = [];
    for ii = 1:len
        data = [data dataInFrame(:,ii) ...
            zeros(vecSize,idlecyclesbetweensamples)]; %#ok
        valid = [valid true(1,1) ...
            false(1,idlecyclesbetweensamples)];       %#ok
        varR = [varR varRtemp{ij}(1) ...
            zeros(1,idlecyclesbetweensamples)];       %#ok
    end

    sampleIn = cast([sampleIn,data, ...
            zeros(vecSize,idlecyclesbetweenframes)],"like",dataInFrame);
    validIn = logical([validIn,valid,zeros(1,idlecyclesbetweenframes)]);
    varRIn = fi([varRIn,varR,zeros(1,idlecyclesbetweenframes)],0,12,0);

end

sampleIn = sampleIn.';
sampletime = 1;
simTime = length(validIn);
```

Run Simulink Model

Run the model. Running the model imports the input signal variables from the MATLAB® workspace to the CIC Interpolator block in the model.

```
open_system(modelname);
sim(modelname);
```



Compare Simulink Block Output with MATLAB System Object Output

Capture the output of the Simulink block. Compare the captured output with the output of the `dsp.CICInterpolator` System object.

```
sampleOut = squeeze(sampleOut_ts.Data).';
validOut  = squeeze(validOut_ts.Data);
cicOutput = sampleOut(validOut);

fprintf('\nCIC Interpolator\n');
difference = (abs(cicOutput-cicFcnOutput(1:length(cicOutput)))>0);
fprintf(['\nTotal number of samples differed between Simulink block ' ...
    'output and MATLAB System object output is: %d \n'],sum(difference));
```

```
CIC Interpolator

Total number of samples differed between Simulink block output and MATLAB System object output is
```

See Also

Blocks
CIC Interpolator

Objects
`dsphdl.CICInterpolator`

Implement CIC Decimator Filter for HDL

This example shows how to use a CIC Decimator block to filter and downsample data. This block supports scalar and vector inputs. To work with scalar and vector inputs separately, this example uses two Simulink® models. You can generate HDL code from the subsystems in these Simulink models.

Set Up Input Data Parameters

Set up these workspace variables for the models to use. These variables configure the CIC Decimator block inside of them. This block supports fixed and variable decimation rates for scalar inputs and only a fixed decimation rate for vector inputs. The example runs the `HDLCICDecimatorModel.slx` model when you set the `scalar` value to `true` and runs the `HDLCICDecimatorModelVectorSupport.slx` model when you set the `scalar` value to `false`. For scalar inputs, choose a range for the input `varRValue` values and set the decimation factor value, `R`, to the maximum expected decimation factor. For vector inputs, the input data must be a column vector of size 1 to 64. `R` must be an integer multiple of input frame size.

```
R = 8;              % Decimation factor
M = 1;              % Differential delay
N = 3;              % Number of sections
scalar = true;      % true for scalar; false for vector
if scalar
    varRValue = [4,R];
    vecSize = 1;
    modelname = "HDLCICDecimatorModel";
else
    varRValue = R; %#ok
    fac = (factor(R));
    vecSize = fac(randi(length(fac),1,1));
    modelname = "HDLCICDecimatorModelVectorSupport";
end

numFrames = length(varRValue);
dataSamples = cell(1,numFrames);
varRtemp = cell(1,numFrames);
cicFcnOutput = [];
WL = 0;             % Word length
FL = 0;             % Fraction length
```

Generate Reference Output from `dsp.CICDecimator` System Object

Generate frames of random input samples and provide them as input to the `dsp.CICDecimator` System object™. The output generated from this System object is used as a reference data for comparison. This System object does not support variable decimation rates, so you must create and release this object for any change in the decimation factor value.

```
for i = 1:numFrames
    framesize = varRValue(i)*randi([5 20],1,1);
    dataSamples{i} = fi(randn(vecSize,framesize),1,16,8);
    varRtemp{i} = fi(varRValue(i)*ones(framesize,1),0,12,0);
    obj = dsp.CICDecimator("DifferentialDelay",M,"NumSections",N, ...
                           "DecimationFactor",varRValue(i));
    cicOut = step(obj,dataSamples{i}(:)).';
    WL = max([WL,cicOut.WordLength]);
    FL = max([FL,cicOut.FractionLength]);
```

```
    cicFcnOutput = [fi(cicFcnOutput,1,WL,FL),cicOut];
    release(obj);
end
```

Convert Input to Stream of Samples and Import them to Simulink Model

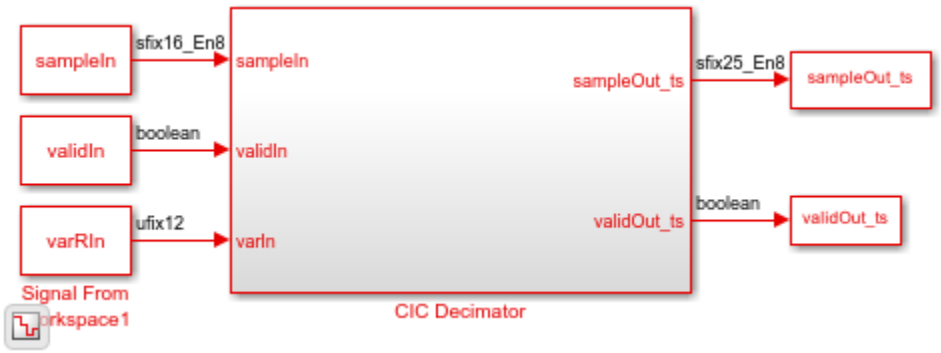
Generate a stream of samples by converting frames to samples. Provide those samples (`sampleIn`) and the valid signal (`validIn`) as inputs to the Simulink model. The latency of the block for scalar and vector inputs is calculated based on the type of input and the number of sections, `N`. For more information, see **"Latency"**. To flush remaining data, run the model by inserting the required number of idle cycles after each frame using the `idlecyclesbetweenframes` value.

```
idlecyclesbetweensamples = 0;
idlecyclesbetweenframes = floor((vecSize-1)*(N/vecSize))+ 1 + N + ...
                          (2+(vecSize+1)*N) + 9;

sampleIn = [];
validIn = [];
varRIn = [];
len = 0;

for ij = 1:numFrames

    dataInFrame = dataSamples{ij};
    if scalar
        len = length(dataInFrame);
    else
        len = size(dataInFrame,2);                    %#ok
    end

    data = [];
    valid=[];
    varR = [];
    for ii = 1:len
        data = [data dataInFrame(:,ii) ...
            zeros(vecSize,idlecyclesbetweensamples)]; %#ok
        valid = [valid true(1,1) ...
            false(1,idlecyclesbetweensamples)];       %#ok
        varR = [varR varRtemp{ij}(ii) ...
            zeros(1,idlecyclesbetweensamples)];       %#ok
    end

    sampleIn = cast([sampleIn,data, ...
        zeros(vecSize,idlecyclesbetweenframes)],"like",dataInFrame);
    validIn = logical([validIn,valid,zeros(1,idlecyclesbetweenframes)]);
    varRIn = fi([varRIn,varR,zeros(1,idlecyclesbetweenframes)],0,12,0);

end

sampletime = 1;
simTime = length(validIn);
```

Run Simulink Model

Run the model. Running the model imports the input signal variables from the MATLAB® workspace to the CIC Decimator block in the model.

```
open_system(modelname);
sim(modelname);
```



Compare Simulink Block Output with MATLAB System Object Output

Capture the output of the Simulink block. Compare that output with the output of the `dsp.CICDecimator` System object.

```
sampleOut = squeeze(sampleOut_ts.Data).';
validOut  = squeeze(validOut_ts.Data);
cicOutput = sampleOut(validOut);

fprintf('\nCIC Decimator\n');
difference = (abs(cicOutput-cicFcnOutput(1:length(cicOutput)))>0);
fprintf(['\nTotal number of samples that differed between Simulink block ' ...
    'output and MATLAB System object output: %d \n'],sum(difference));
```

```
CIC Decimator

Total number of samples that differed between Simulink block output and MATLAB System object outp
```

See Also

Blocks
CIC Decimator

Objects
`dsphdl.CICDecimator`

Implement Downsampler For HDL

This example shows how to use the Downsampler block to downsample data. This block supports scalar and vector inputs. You can generate HDL code from the subsystem used in this example.

Set Up Input Data Parameters

Set up workspace variables for the Simulink® model to use. These variables configure the Downsampler block inside the subsystem.

```
K = 8;                  % Downsample factor
O = 3;                  % Sample offset
scalar = true;
if scalar
    vecSize = 1;
    value = 1;
else
    vecSize = 16;  %#ok % Vector size must be a multiple or factor of K
    value = vecSize/K;
end
```

Generate Reference Data From Function

Generate frames of random input samples and apply to the `downsample` function. Use the output as reference data against which to compare the output of the block.

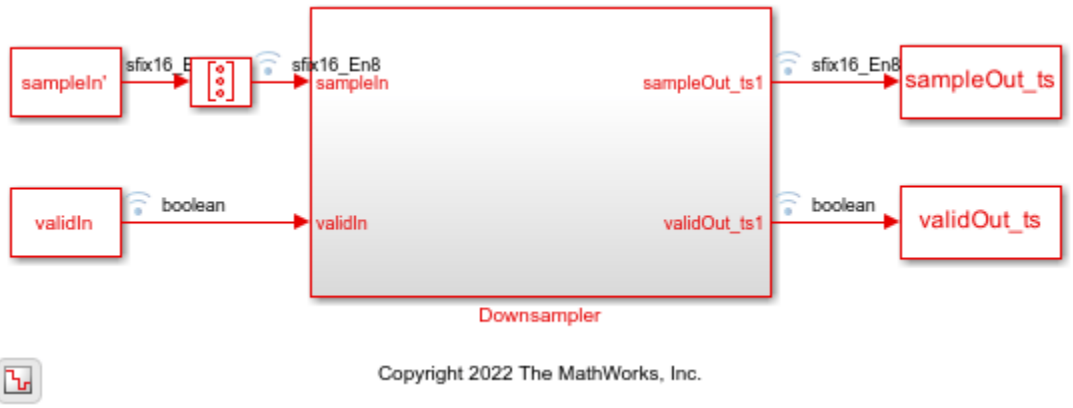
```
totalsamples = 0;
numFrames = 1;
dataSamples = cell(1,numFrames);
framesize = zeros(1,numFrames);
refOutput = [];
WL = 0;           % Word length
FL = 0;           % Fraction length

for i = 1:numFrames
    framesize(i) = randi([5 200],1,1)*vecSize;
    dataSamples{i} = fi(randn(vecSize,framesize(i)),1,16,8);
    ref_downsample= downsample((dataSamples{i}(:)),K,O);
    refOutput = [refOutput,ref_downsample]; %#ok
end
```

Run Model

Run the model to import the input signal variables from the MATLAB® workspace to the Downsampler block in the model. Insert the required number of idle cycles after each frame using the `latency` variable to avoid invalid output data. Initialize the size of the output to accommodate the output data.

```
latency = 2+O;
sampleIn = dataSamples{i};
validIn = true(1,sum(framesize));

sampletime = 1;
simTime = length(validIn)+latency;

modelname = "HDLDownsampler";
```

```
open_system(modelname);
sim(modelname);
```



Compare Block Output with Function Output

Compare the output of the block with the output of the `Downsampler` function.

```
sampleOut = squeeze(sampleOut_ts.Data);
validOut  = squeeze(validOut_ts.Data);
if scalar
    HDLOutput = sampleOut(validOut);
else
    HDLOutput = sampleOut(:,validOut); %#ok
end
HDLOutput = HDLOutput(:);

fprintf('\n Downsampler\n');
difference = (abs(HDLOutput-refOutput(1:length(HDLOutput)))>0);
fprintf(['\nTotal number of samples differed between Behavioral ' ...
    'and HDL simulation: %d \n'],sum(difference));
```

```
 Downsampler

Total number of samples differed between Behavioral and HDL simulation: 0
```

See Also

Blocks
Downsampler

Objects
`dsphdl.Downsampler`

Implement Upsampler for HDL

This example shows how to use the Upsampler block to upsample data. This block supports scalar and vector inputs. You can generate HDL code from the subsystem used in this example.

Set Up Input Data Parameters

Set up workspace variables for the Simulink® model to use. These variables configure the Upsampler block inside the subsystem.

```
L = 8;                  % Upsample factor
O = 0;                  % Sample offset

scalar = true;
if scalar
    vecSize = 1;
    outSize = 1;
    minCycles = L;
else
    vecSize = 16;   %#ok % Vector size must be a multiple or factor of L
    outSize = vecSize*L;
    minCycles = 1;
end
```

Generate Reference Data from Function

Generate frames of random input samples and apply the `upsample` function. Use the output as reference data against which to compare the output of the block.

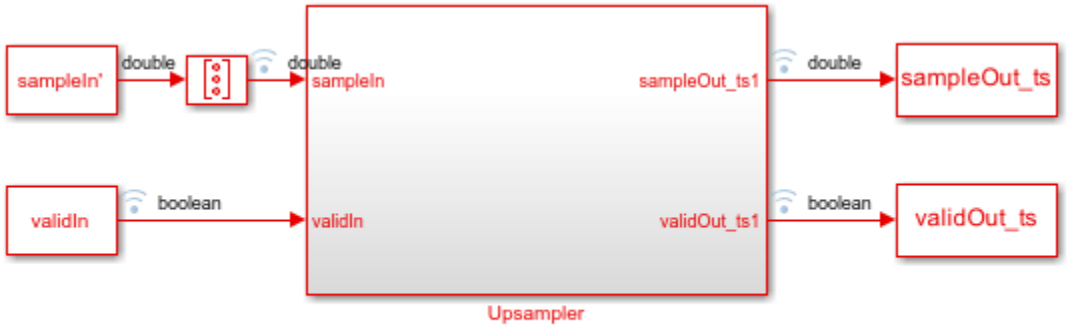
```
totalsamples = 0;
numFrames =1;
dataSamples = cell(1,numFrames);
framesize = zeros(1,numFrames);
refOutput = [];
WL = 0;            % Word length
FL = 0;            % Fraction length

for i = 1:numFrames
    framesize(i) = randi([5 200],1,1);
    dataSamples{i} = fi(randn(vecSize,framesize(i)),1,16,8);
    ref_upsample= upsample((dataSamples{i}(:)),L,O);
    refOutput = [refOutput,ref_upsample]; %#ok
end
```

Run Model

Run the model to import the input signal variables from the MATLAB® workspace to the Upsampler block in the model. Insert the required number of idle cycles after each frame using the `latency` variable to avoid invalid output data. Initialize the size of the output to accommodate the output data.

```
latency = 3;
sampleIn = zeros(vecSize,size(dataSamples{i},2)*minCycles);
validIn = upsample(true(1,length(dataSamples{i})),minCycles);
sampleIn(:,validIn) = dataSamples{i};

sampletime = 1;
```

```
simTime = length(validIn)+latency;

modelname = "HDLUpsampler";
open_system(modelname);
sim(modelname);
```



Compare Block Output with Function Output

Compare the output of the block with the output of the `Upsampler` function.

```
sampleOut = squeeze(sampleOut_ts.Data);
validOut  = squeeze(validOut_ts.Data);
if scalar
    HDLOutput = sampleOut(validOut);
else
    HDLOutput = sampleOut(:,validOut); %#ok
end
HDLOutput = HDLOutput(:);

fprintf('Upsampler\n');
difference = (abs(HDLOutput-refOutput(1:length(HDLOutput)))>0);
fprintf(['\nTotal number of samples differed between Behavioral ' ...
    'and HDL simulation: %d \n'],sum(difference));
```

```
Upsampler

Total number of samples differed between Behavioral and HDL simulation: 0
```

See Also

Blocks
Upsampler

Objects
`dsphdl.Upsampler`

Calculate Mean Square Error Performance Using LMS Filter

This example shows how use the LMS Filter block to calculate the mean square error performance in additive white Gaussian noise (AWGN). The block supports scalar and vector inputs of type real or complex. You can generate the HDL code from the `LMSFilter` subsystem in this Simulink® model.

Set Up Input Variables

Set up these workspace variables for the model. These variables configure the LMS Filter block.

```
filterLength = 64;
stepSize = 0.2/(1*filterLength); % recommended step size is less than
                                 % 1/(2 x power of input signal x filter length)
vecSize = 2;
inptType = 1;                    % 0 means real; 1 means complex
niter = 20;                      % number of iterations to run for calculating mean square error
frameSize = 2800;
simtime = frameSize/vecSize + 100;
errSumAbsSq = zeros(1,frameSize);
errRefSumAbsSq = zeros(1,frameSize);
```

Generate Input Data and Run Model

Generate input data for the LMS Filter block and the reference System object™ `dsp.LMSFilter` for `niter` number of iterations.

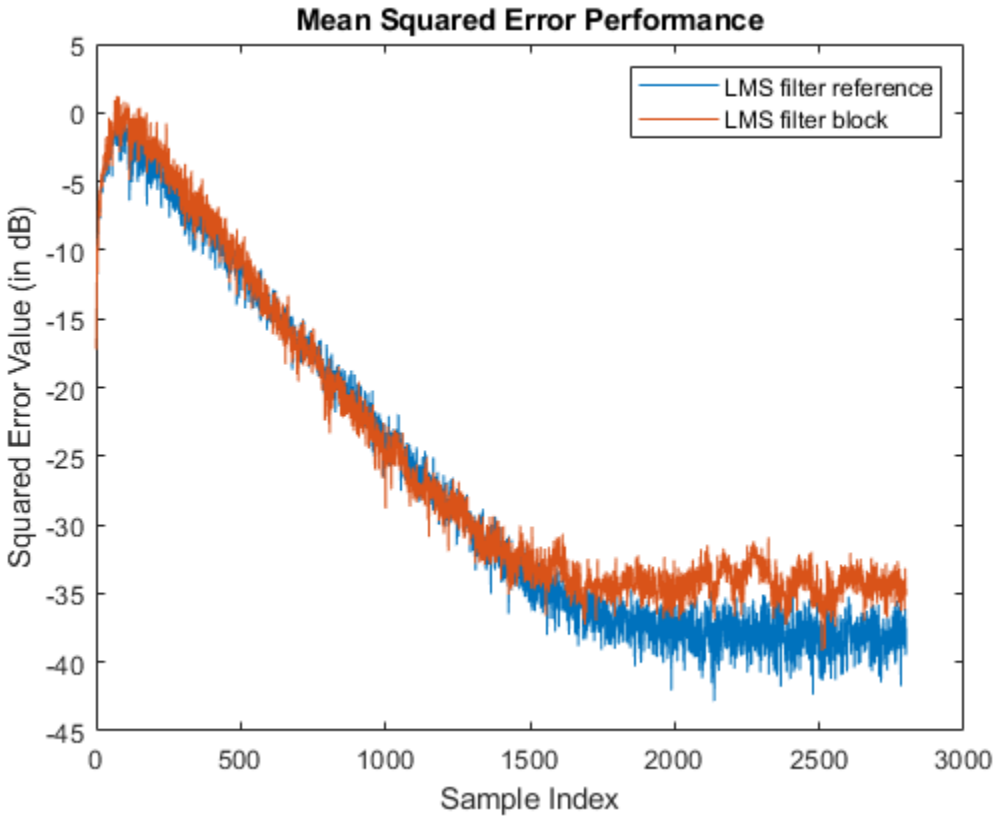
```
for k = 1:niter
    % Create filter object
    filterObj = dsp.FIRFilter;
    num = normalize(randn(1,filterLength) + inptType*1i*randn(1,filterLength))/sqrt(filterLength)
    filterObj.Numerator = num;

    % Use dsp.LMSFilter object for reference
    lmsfilt = dsp.LMSFilter(filterLength, 'StepSize', stepSize);

    % Generate input data
    observedSignal = fi(randn(frameSize,1,'like',num),1,18,15);
    noise = 0.012*randn(frameSize,1);
    validIn = true(frameSize,1);

    % Generate desired signal by adding noise to filtered data
    desiredSignal = fi(filterObj(double(observedSignal)) + noise,1,18,15);

    % Capture reference output values and calculate average of absolute square error
    [filtOutRef,errRef,wtsRef] = lmsfilt(double(observedSignal),double(desiredSignal));
    errRefSumAbsSq = errRefSumAbsSq + abs(errRef(1:frameSize)).^2.';

    % Simulate model
    LMSModel = 'HDLLMSBlock';
    load_system(LMSModel);% run this command to open the model
    lms = sim(LMSModel);

    % Capture model output and calculate average of absolute square error
    validOut = squeeze(lms.validOut);
    errOutLMS = squeeze(double(lms.errOut)).';
    if vecSize ==1
```

```
        errOutAbsSq = abs(errOutLMS(:,validOut)).^2.';
    else
       errOutAbsSq = abs(errOutLMS(validOut,:)).^2.';
    end
    errOutAbsSq = errOutAbsSq(:).';
    errSumAbsSq = errSumAbsSq + errOutAbsSq(1:frameSize);

    % To reset the internal states of the object
    reset(filterObj)
    reset(lmsfilt)
end
```

Plot Mean Square Error

Plot the mean square error in dB against the LMS Filter block output and the LMS filter reference output.

```
meanSqErrRefLMS_dB = 10*log10(errRefSumAbsSq/niter);
meanSqErrLMS_dB = 10*log10(errSumAbsSq/niter);
figure
plot(meanSqErrRefLMS_dB)
hold on
plot(meanSqErrLMS_dB)
title('Mean Squared Error Performance')
xlabel('Sample Index')
ylabel('Squared Error Value (in dB)')
legend('LMS filter reference','LMS filter block')
```



See Also

Blocks
LMS Filter

Objects
`dsphdl.LMSFilter`

HDL Code Generation and Deployment

Prototype DSP HDL Algorithms on Hardware

Support packages such as the HDL Coder Support Package for Xilinx RFSoC Devices enable you to design, prototype, and verify practical digital signal processing systems on hardware.

- Use an Xilinx RFSoC board as an I/O peripheral to transmit and receive real-time arbitrary waveforms using MATLAB® System objects or Simulink blocks.
- Acquire high-bandwidth signals by using burst mode.
- Run signal processing application examples to get started with targeting your designs to hardware.
- In Simulink, customize and prototype DSP algorithms. Target only the FPGA fabric of the device, or deploy partitioned hardware-software co-design implementations across the ARM® processor and the FPGA fabric of the device.

The “Polyphase Channelizer” (HDL Coder Support Package for Xilinx RFSoC Devices) example shows how to use the Channelizer block to process incoming analog-to-digital converter (ADC) samples and produce a spectrum that has 512 MHz of bandwidth. The “Pulse-Doppler Radar Using Xilinx RFSoC Device” (SoC Blockset Support Package for Xilinx Devices) example shows how to use the Discrete FIR Filter block in a pulse-Doppler radar system targeted on the Xilinx Zynq® UltraScale+™ RFSoC evaluation kit.

How to Install Support Packages

A support package is an add-on that enables you to use a MathWorks® product with specific third-party hardware and software. Support packages use the license of the base product. For instance, HDL Coder Support Package for Xilinx RFSoC Devices requires a license for HDL Coder.

Install support packages using the MATLAB **Add-Ons** menu. You can also use the **Add-Ons** menu to update installed support package software or update the firmware on third-party hardware.

To install support packages, on the MATLAB **Home** tab, in the **Environment** section, select **Add-Ons > Get Hardware Support Packages**. You can filter this list by selecting categories (such as hardware vendor or application area) or by performing a keyword search.

Search the **Add-Ons** list for these support packages:

- HDL Coder Support Package for Xilinx RFSoC Devices
- SoC Blockset™ Support Package for Xilinx Devices
- HDL Coder Support Package for Intel SoC Devices
- HDL Coder Support Package for Xilinx Zynq® Platform
- HDL Coder Support Package for Intel FPGA Boards
- Embedded Coder® Support Package for Xilinx Zynq Platform (needed only for hardware-software co-design)
- Embedded Coder Support Package for Intel SoC Devices (needed only for hardware-software co-design)

When the support package installation is complete, you must set up the host computer and radio hardware. For Windows® systems, the installer provides guided setup steps. For Linux® systems, the installer links to manual setup instructions.

See Also

More About

- Xilinx RFSoC Devices
- “Pulse-Doppler Radar Using Xilinx RFSoC Device” (SoC Blockset Support Package for Xilinx Devices)
- “Polyphase Channelizer” (HDL Coder Support Package for Xilinx RFSoC Devices)

Radar Application Examples

FPGA-Based Beamforming in Simulink: Algorithm Design

This example shows the first half of a workflow to develop a beamformer in Simulink® suitable for implementation on hardware, such as a field programmable gate array (FPGA). It also shows how to compare the results of the implementation model with those of a behavioral model.

The second part of the example “FPGA-Based Beamforming in Simulink: Code Generation” on page 5-9 shows how to generate HDL code from the implementation model and verify that the generated HDL code produces the correct results compared to the behavioral model.

This example shows how to implement an FPGA-ready beamformer to match a corresponding behavioral model in Simulink by using the Phased Array System Toolbox™, DSP System Toolbox™, and Fixed-Point Designer™ libraries. To verify the implementation model, this example compares the simulation output of the implementation model with the output of the behavioral model.

The Phased Array System Toolbox is used to design and verify the floating-point functional algorithm, which provides the behavioral reference model. The behavioral model is then used to verify the results of the fixed-point implementation model used to generate HDL code.

Fixed-Point Designer provides data types and tools for developing fixed-point and single-precision algorithms to optimize performance on embedded hardware. You can perform bit-true simulations to observe the impact of limited range and precision without implementing the design on hardware.

Partitioning Model for FPGA

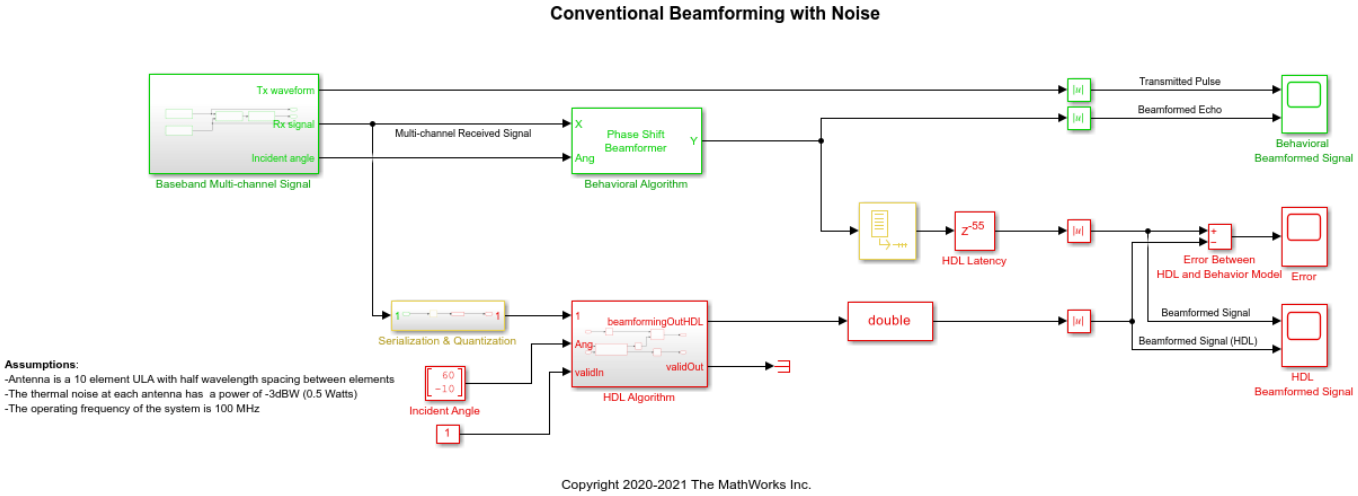
There are three key modeling concepts to keep in mind when preparing a Simulink model to target FPGAs:

- **Sample-based processing:** Also commonly referred to as serial processing, sample-based processing is an efficient data processing technique in hardware designs that enables you to tradeoff between resources and throughput.
- **Subsystem targeted for HDL code generation:** In order to generate HDL code from a model, the implementation algorithm must be inside a Simulink subsystem.
- **Time-aligned outputs of behavioral and implementation models:** For comparing the outputs of the behavioral and FPGA implementation models, you must time align their outputs by adding latency to the behavioral model.

Beamforming Algorithm

This example shows a phase-shift beamformer as the behavioral algorithm, which is re-implemented in the `HDL Algorithm` subsystem by using Simulink blocks that support HDL code generation. The beamformer calculates the phase required between each of the ten channels to maximize the received signal power in the direction of the incident angle. The figure shows the Simulink model with the behavioral algorithm and its corresponding implementation algorithm for an FPGA.

```
modelname = 'SimulinkBeamformingHDLWorkflowExample';
open_system(modelname);
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```



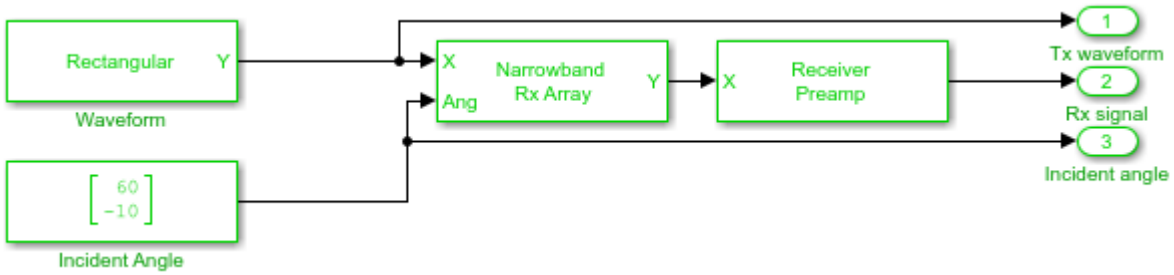
The Simulink model has two branches. The top branch is the behavioral, floating-point model of the algorithm and the bottom branch is the functionally-equivalent fixed-point version using blocks that support HDL code generation. Besides plotting the output of both branches to compare the two, this example also calculates and plots the difference, or error, between both outputs.

The model has a Delay (Z^{-55}) block at the output of the behavioral algorithm. This delay is necessary because the implementation algorithm uses 55 delays to implement pipelining which creates latency that needs to be accounted for. Accounting for this latency is called delay balancing and is necessary to time-align the output between the behavioral model and the implementation model to make it easier to compare the results.

Multi-Channel Receive Signal

To synthesize a received signal at the phased array antenna, the model includes a subsystem that generates a multi-channel signal. The `Baseband Multi-channel Signal` subsystem models a transmitted waveform and the received target echo at the incident angle captured via a 10-element antenna array. The subsystem also includes a receiver pre-amp model to account for receiver noise. This subsystem generates the input stimulus for our behavioral and implementation models.

```
open_system([modelname '/Baseband Multi-channel Signal']);
```

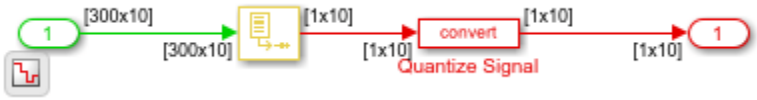


Serialization and Quantization

The model includes a `Serialization & Quantization` subsystem which converts floating-point, frame-based signals to fixed-point, sample-based signals necessary for modeling streaming data in

hardware. This model uses scalar streaming processing because this algorithm runs slower than 400 MHz. The hardware implementation can be optimized for resources rather than for higher throughput.

```
open_system([modelname '/Serialization & Quantization']);
set_param(modelname,'SimulationCommand','update')
```



The input signal to the serialization subsystem has 10 channels with 300 samples per channel or a 300x10 size signal. The subsystem serializes, or unbuffers, the signal producing a sample-based signal that's 1x10, i.e., one sample per channel, which is then quantized to meet the requirements of the system.

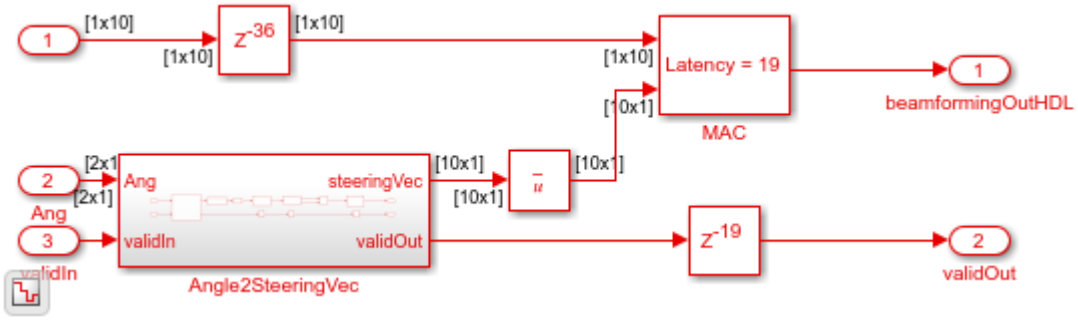
The output data type of the Quantize Signal block is `fixdt(1,12,9)`, which is a signed value with 12-bit word length and 9-bit fraction length precision. The word length is 12 bits because the design is targeted to a Xilinx® Virtex®-7 FPGA which is connected to a 12-bit ADC. The fraction length accommodates the maximum range of the input signal.

Designing the Implementation Subsystem

The `HDL Algorithm` subsystem, which is targeted for HDL code generation, implements the beamformer, which was designed using Simulink blocks that support HDL code generation.

The `Angle2SteeringVec` subsystem calculates the signal delay at each antenna element of a Uniform Linear Array (ULA). The delay is then fed to a multiply and accumulate (MAC) subsystem to perform beamforming.

```
open_system([modelname '/HDL Algorithm']);
```



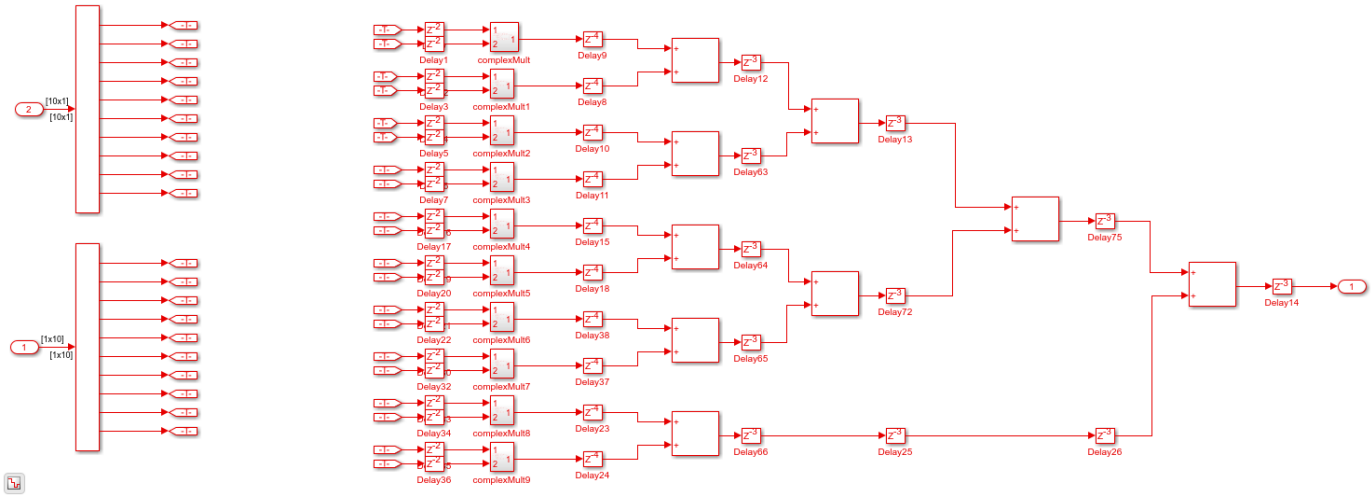
The algorithm in the `HDL Algorithm` subsystem is functionally equivalent to the phase-shift beamforming behavioral algorithm but can generate HDL code. There are three main differences that enable this subsystem to generate efficient HDL code.

1. The design processes inputs serially, i.e. using sample-based processing.
2. The subsystem uses fixed-point data types for any calculations.
3. The implementation includes Delays that enable pipelining by the HDL synthesis tool.

To ensure proper clock timing, any delay added to one branch of the implementation model must be matched to all other parallel branches as seen above. The `Angle2SteeringVec` subsystem, for

example, has 36 delays; therefore, the top branch of the `HDL Algorithm` subsystem includes a delay of 36 samples right before the `MAC` subsystem. Likewise, the `MAC` subsystem uses 19 delays, which must be balanced by adding 19 delays to the output of the `Angle2SteeringVec` subsystem. This figure shows the inside of the `MAC` subsystem to illustrate the 19 delays.

```
open_system([modelname '/HDL Algorithm/MAC']);
set_param(modelname,'SimulationCommand','update')
```

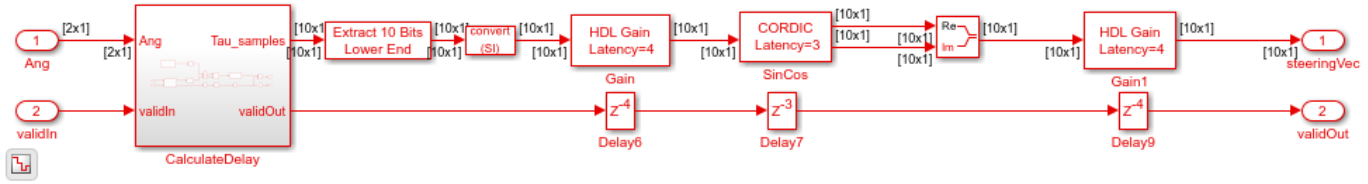


The bottom branch of the `MAC` subsystem has a Z^{-2} block, followed by the complex multiply block which contains a Z^{-1}, then a Z^{-4} block, followed by 4 delay blocks of Z^{-3} for a total of 19 delays. The delay values are defined in the PreLoadFcn callback in Model Properties.

Calculating the Steering Vector

The `Angle2SteeringVec` subsystem calculates the steering vector from the signal's angle of arrival. It first calculates the signal's arrival delay at each sensor by matrix-multiplying the antenna element position in the array by the incident direction of the signal. The delays are then passed to the `SinCos` subsystem which calculates the sine and cosine trigonometric functions by using the simple and efficient CORDIC algorithm.

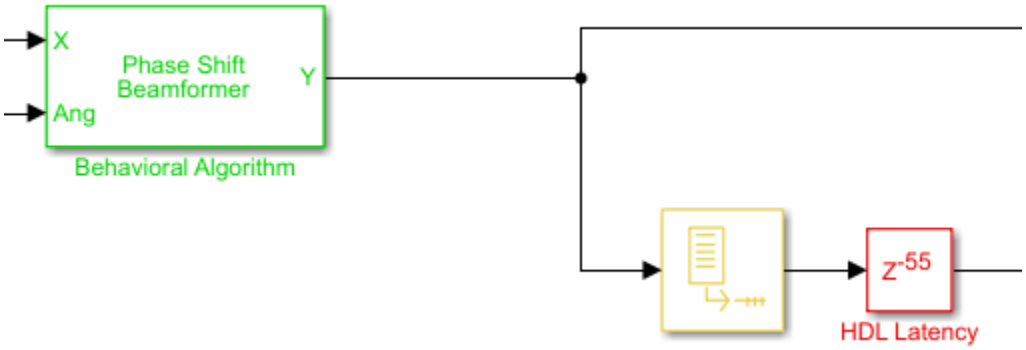
```
open_system([modelname '/HDL Algorithm/Angle2SteeringVec']);
```



Because this design consists of a 10 element ULA spaced at half-wavelength, the antenna element position is based on the spacing between each antenna element measured outwardly from the center of the antenna array. Define the spacing between elements as a vector of 10 numbers ranging from -6.7453 to 6.7453, i.e., with a spacing of 1/2 wavelength, which is 2.99/2. In fixed-point arithmetic, the data type used for the element spacing vector is `fixdt(1,8,4)`, i.e., a signed value with 8-bit word length and 4-bit fraction length.

Deserialization

To compare your sample-based fixed-point implementation with the floating-point frame-based behavioral design you need to deserialize the output of the implementation subsystem and convert it to a floating-point data type. Alternatively, you can compare the results directly with sample-based signals but then you must unbuffer the output of the behavioral model to match the sample-based signal output from the implementation algorithm, as shown in this figure.

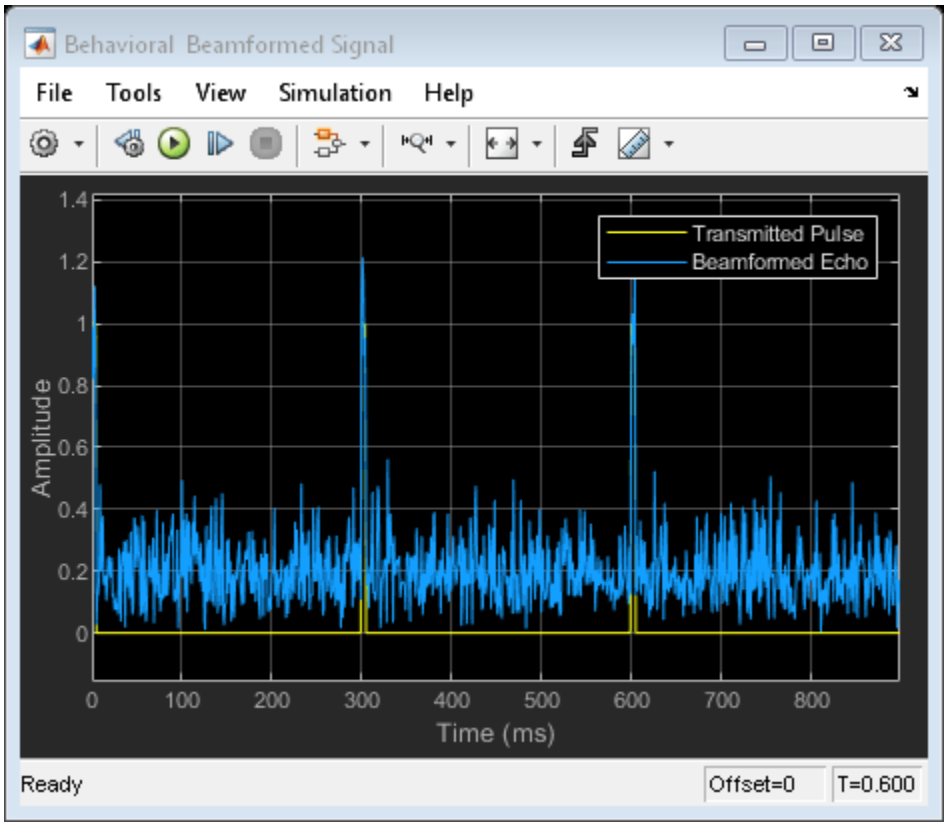


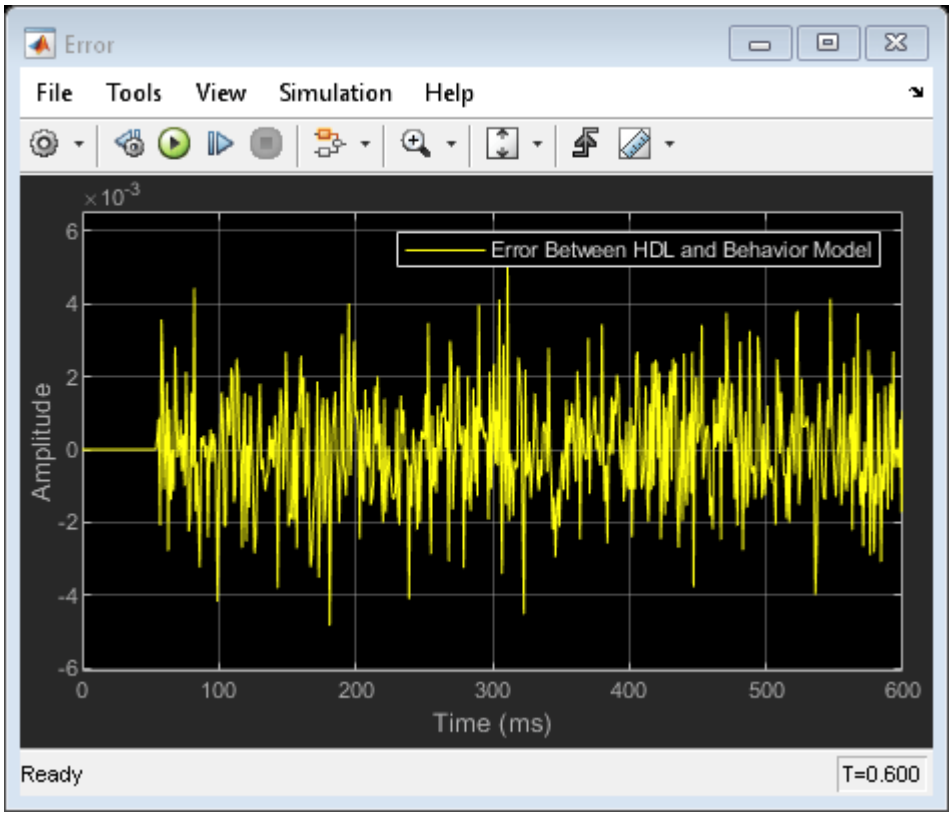
In this case, you only need to convert the output of the `HDL Algorithm` subsystem to floating-point by setting the output of the Data Type Conversion block to `double`.

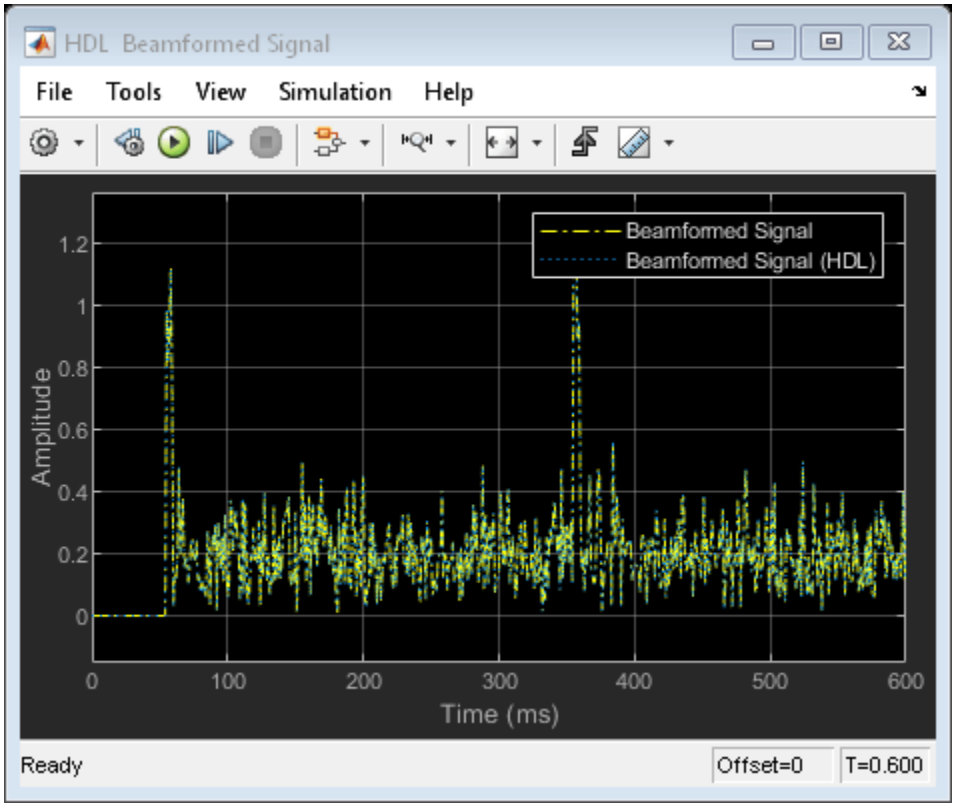
Comparing Output of HDL Model to Behavioral Model

Run the model to display the results. You can run the Simulink model by clicking the Play button or calling the sim command from the MATLAB® command line. Use the scopes to compare the outputs visually.

```
sim(modelname);
```


Behavioral Beamformed Signal
File Tools View Simulation Help
Transmitted Pulse
Beamformed Echo
Amplitude
Time (ms)
Ready
Offset=0
T=0.600


Error
File Tools View Simulation Help
×10^-3
Error Between HDL and Behavior Model
Amplitude
Time (ms)
Ready
T=0.600



The Time Scope of the Beamformed Signal and Beamformed Signal (HDL) shows that the two signals are nearly identical. The error is on the order of 10^-3 in the Error scope. This result shows that the `HDL Algorithm` subsystem is producing the same output as the behavioral model within quantization error. This verification is an important first step before generating HDL code.

Because the HDL model used 55 delays, the scope titled HDL Beamformed Signal is delayed by 55ms when compared to the original transmitted or beamformed signal shown on the Behavioral Beamformed Signal scope.

Summary

This example is the first of a two-part tutorial series on how to design an FPGA-ready algorithm, automatically generate HDL code, and verify the HDL code in Simulink. This example showed how to use blocks from the Phased Array System Toolbox to create a behavioral model to serve as a golden reference, and how to create a subsystem for hardware implementation using Simulink blocks that support HDL code generation. It also compared the output of the implementation model to the output of the corresponding behavioral model to verify that the two algorithms are functionally equivalent.

Once you verify that your implementation algorithm is functionally equivalent to your golden reference, you can use HDL Coder™ for “HDL Code Generation from Simulink” (HDL Coder) and HDL Verifier™ to “Generate a Cosimulation Model” (HDL Coder) test bench.

The second part of this two-part tutorial series “FPGA-Based Beamforming in Simulink: Code Generation” on page 5-9 shows how to generate HDL code from the implementation model and verify that the generated HDL code produces the same results as the floating-point behavioral model as well as the fixed-point implementation model.

FPGA-Based Beamforming in Simulink: Code Generation

This example shows the second half of a workflow to generate HDL code for a beamforming algorithm and verify that the generated code is functionally correct.

The first part of the example, “FPGA-Based Beamforming in Simulink: Algorithm Design” on page 5-2, shows how to develop an algorithm in Simulink® suitable for implementation on hardware, such as a field programmable gate array (FPGA), and how to compare the output of the fixed-point implementation model to that of the corresponding floating-point behavioral model.

This example uses HDL Coder™ to generate HDL code from the Simulink model developed in the first part and verifies the HDL code using HDL Verifier™. HDL Verifier is used to generate a cosimulation test bench model to verify the behavior of the generated HDL code. The test bench uses ModelSim® for cosimulation to verify the automatically generated HDL code.

The Phased Array System Toolbox™ Simulink blocks model operations on frame-based, floating-point data and provides the behavioral reference model. The example uses this behavioral model to verify the results of the implementation of the algorithm for hardware and the generated HDL code.

HDL Coder generates portable synthesizable Verilog® and VHDL® code for over 300 Simulink blocks that support HDL code generation. Those Simulink blocks operate on serial data using fixed-point arithmetic with delays included to enable pipelining by the synthesis tool.

HDL Verifier helps you test and verify Verilog and VHDL designs for FPGAs, ASICs, and SoCs. This example verifies HDL generated from a Simulink model against a test bench running in Simulink using cosimulation with an HDL simulator.

Implementation Model

This tutorial assumes that you have a Simulink model that contains a subsystem with a beamforming algorithm designed using Simulink blocks that use fixed-point arithmetic and support HDL code generation. The “FPGA-Based Beamforming in Simulink: Algorithm Design” on page 5-2 example shows how to create such a model.

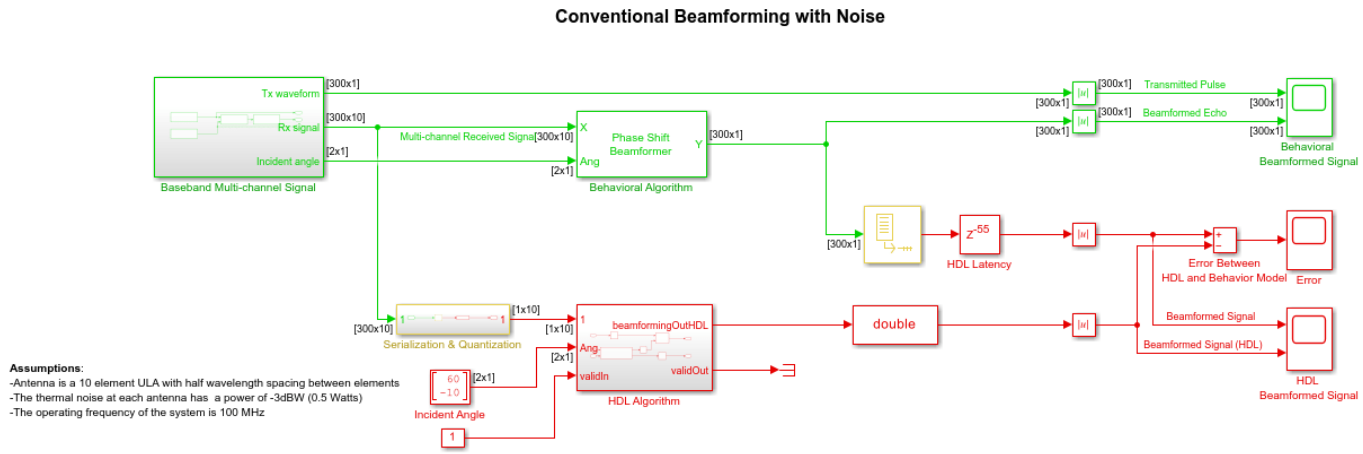
Alternatively, if you start with a new model, you can run `hdlsetup` (HDL Coder) to configure the Simulink model for HDL code generation. To configure the Simulink model for test bench creation, open the Model Settings, select **Test Bench** under **HDL Code Generation** in the left panel, and check **HDL test bench** and **Cosimulation model** in the **Test Bench Generation Output** properties group.

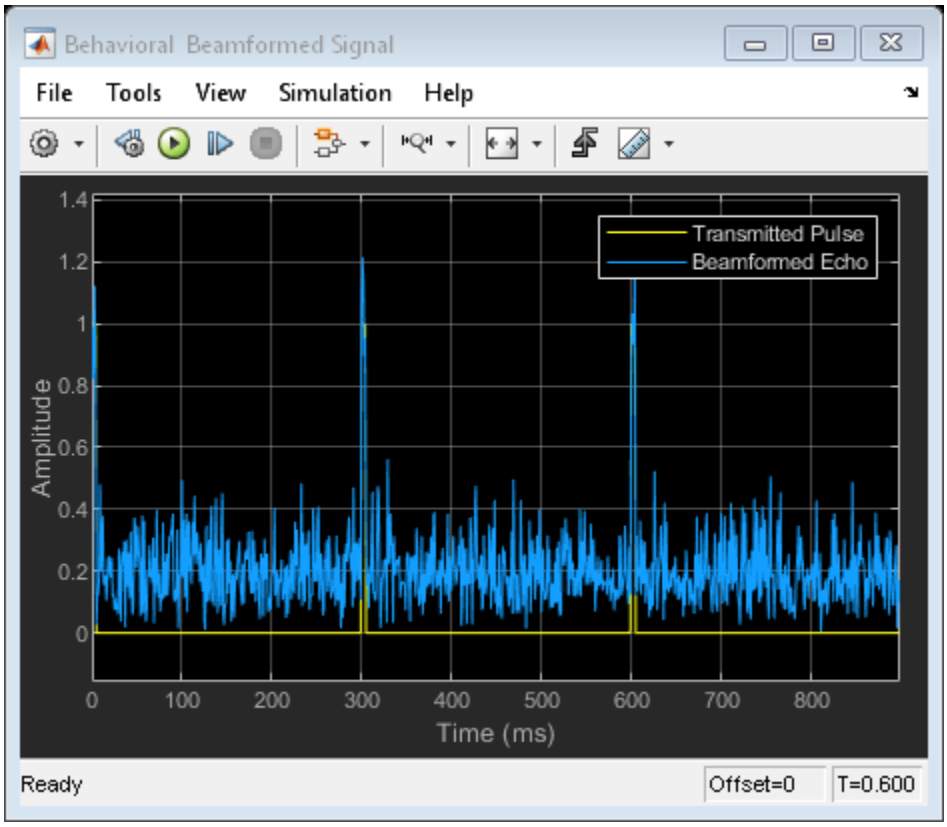
Comparing Results of Implementation Model to Behavioral Model

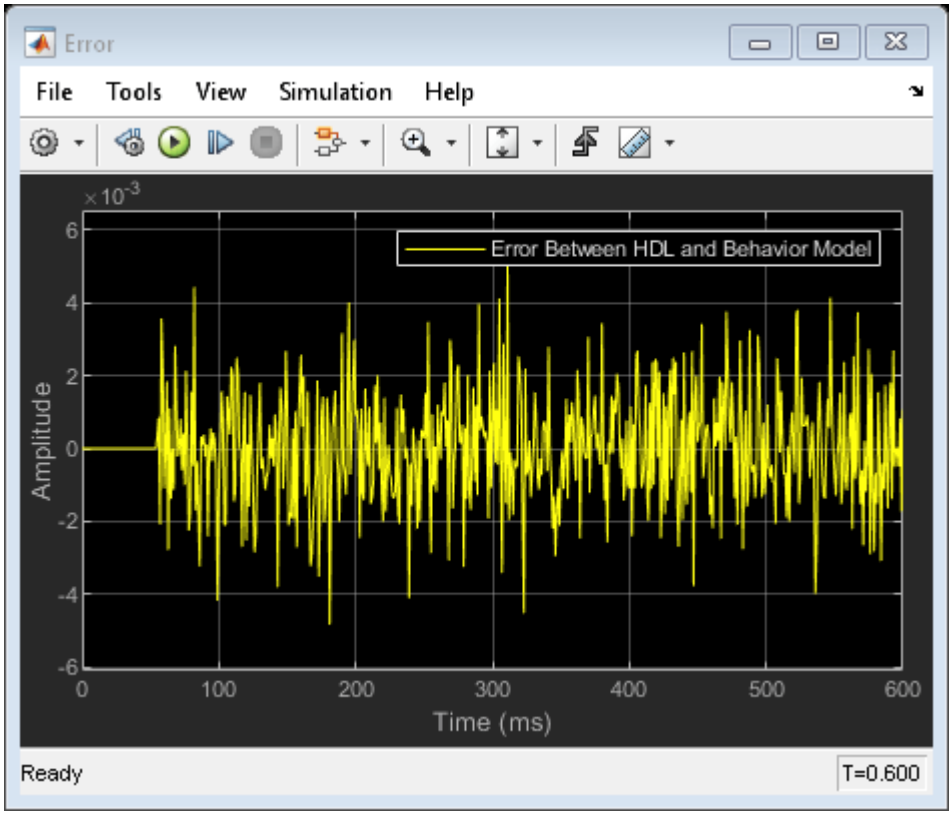
Run the model created in the “FPGA-Based Beamforming in Simulink: Algorithm Design” on page 5-2 example to display the results. You can run the Simulink model by clicking the Play button or calling the sim command from the MATLAB® command line. Use the Time Scope blocks to compare the output frames visually.

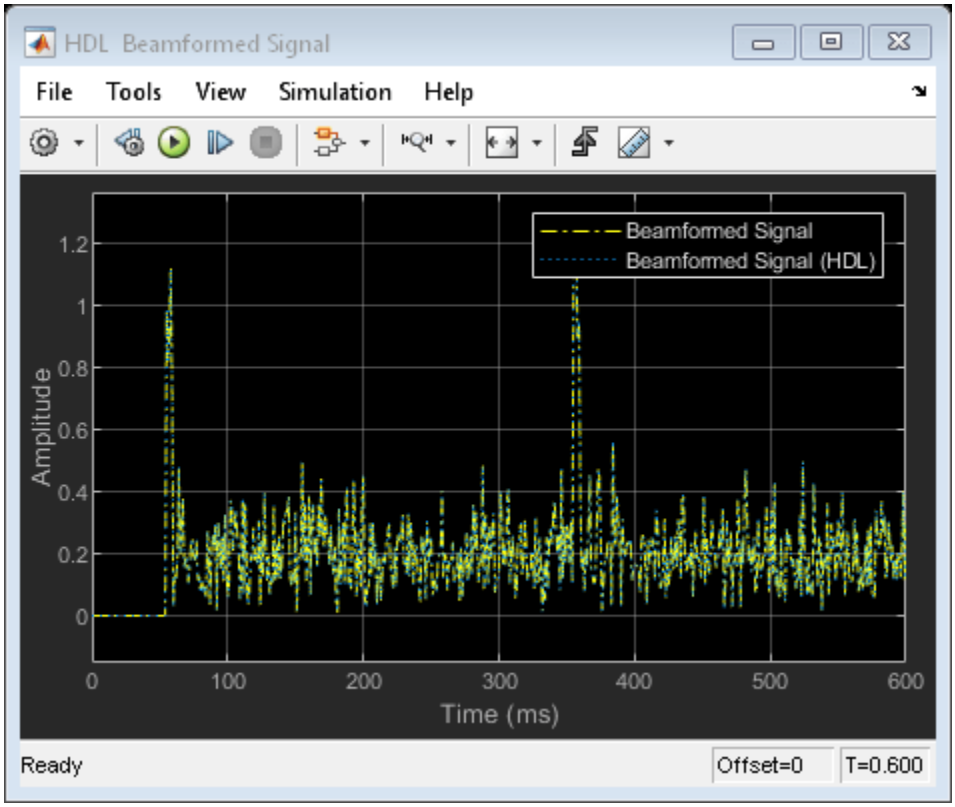
```
modelname = 'SimulinkBeamformingHDLWorkflowExample';
open_system(modelname);

% Ensure model is visible and not obstructed by scopes.
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```

```
sim(modelname);
```






Error
File Tools View Simulation Help
×10^-3
Error Between HDL and Behavior Model
Amplitude
Time (ms)
Ready
T=0.600


HDL Beamformed Signal
File Tools View Simulation Help
Beamformed Signal
Beamformed Signal (HDL)
Amplitude
Time (ms)
Ready
Offset=0
T=0.600

Model Settings

Once you verify that your fixed-point implementation model produces the same results as your floating-point behavioral model, you can generate HDL code and test bench. First, set the HDL Code Generation parameters in the Simulink Configuration Parameters dialog. For this example, set the following parameters in Model Settings under HDL Code Generation:

- **Target:** Xilinx Vivado synthesis tool; Virtex7 family; Device xc7vx485t; package ffg1761, speed -1; and target frequency of 300 MHz.
- **Optimization:** Uncheck all optimizations except Balance delays.
- **Global Settings:** Set the Reset type to Asynchronous.
- **Test Bench:** Select HDL test bench and Cosimulation model.

The reason the optimizations are disabled in this model is because some blocks used in our implementation are already HDL-optimized blocks, which could conflict with the HDL Coder optimizations.

HDL Code Generation and Test Bench Creation

Next, use HDL Coder to generate HDL code for the `HDL Algorithm` subsystem. For an example of how to generate HDL code, see “Generate HDL Code from Simulink Model” (HDL Coder).

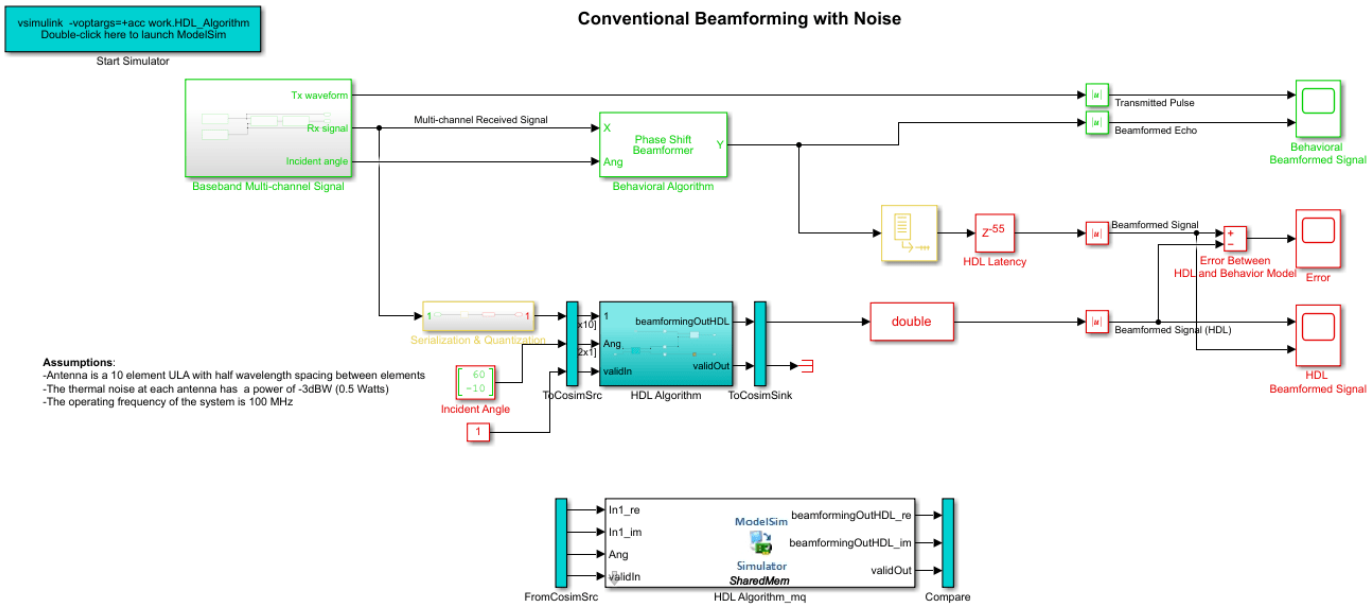
Use these commands to generate HDL code and test bench.

```
makehdl([modelname '/HDL Algorithm']);   % Generate HDL code
makehdltb([modelname '/HDL Algorithm']); % Generate Cosimulation test bench
```

When you call the `makehdl` command, the MATLAB command window shows the amount of delay added during the automatic code generation process. In this case, 24 delays are added which results in an extra delay of 24*1ms = 24ms. The final output is expected to be delayed this additional amount, for a total delay of 79ms.

Because of this extra delay added during code generation, the output of the floating-point behavioral model needs to be delay balanced by adding 24 delays to the original 55. This delay aligns the output of the behavioral model with the implementation model and the cosimulation output.

The process of generating the HDL code and test bench creates a new Simulink model in the working folder, named `gm_<modelname>_mq`. The model contains a ModelSim Cosimulation block.



Use these commands to open the test bench model.

```
modelname = ['gm_',modelname,'_mq'];
open_system(modelname);
```

In this new model, change the delay setting in the HDL Latency block to 79 to account for the 24 delays added by the code generation process. Using a delay of 79 ensures that the behavioral model output is time-aligned with the output of the implementation and cosimulation output.

HDL Code Verification via Cosimulation

Before opening ModelSim make sure that the command to start ModelSim, `vsim`, is on the path of your machine.

To run the cosimulation model, double-click the blue rectangular box in the upper-left corner of the Simulink test model to launch ModelSim.

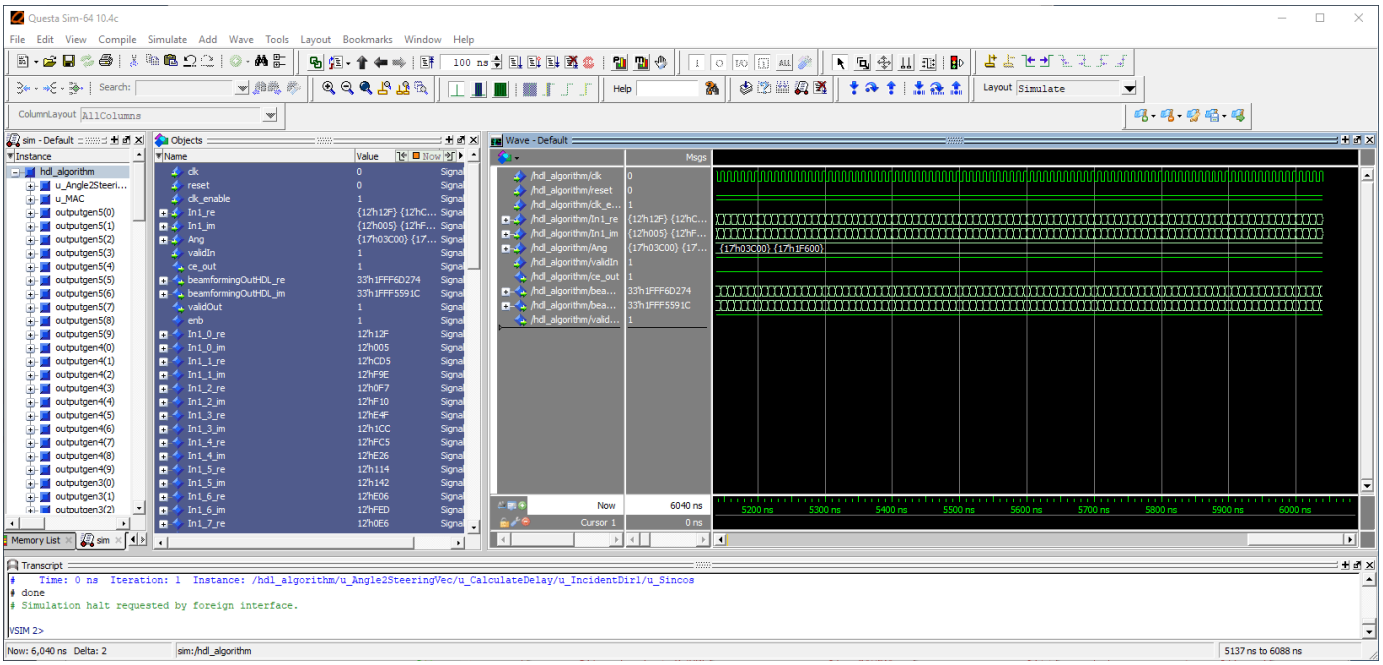
Run the Simulink test bench model to display the simulation results. You can run the Simulink model by clicking the Play button or calling the sim command on the MATLAB command line. The test bench model includes Time Scope blocks to compare the output of the cosimulation performed with ModelSim with the output of the HDL subsystem in Simulink.

If you have ModelSim on your path, use this command to run the test bench.

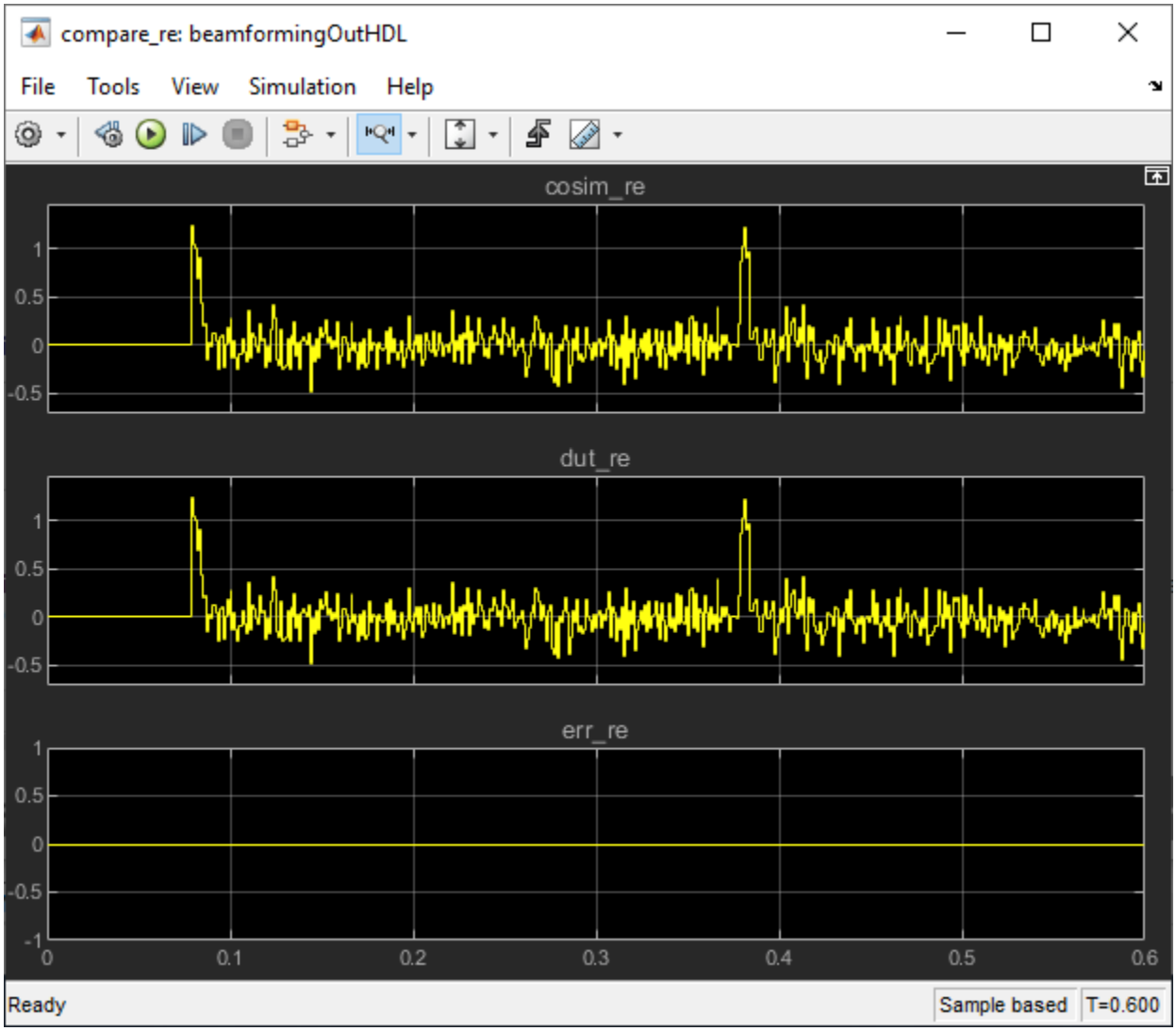
```
sim(modelname);
```

After starting ModelSim, running the Simulink test bench model will populate Questa Sim with the HDL model's waveforms and Time Scopes in Simulink. The figures show examples of the results in Questa Sim and Simulink scopes.

NOTE: You must restart Questa Sim each time you want to run the Simulink simulation. To restart, call `restart` at the Questa Sim command line. Alternatively, you can quit Questa Sim and re-launch it by double-clicking the blue box in the upper-left corner of the Simulink test bench model.



The Simulink scope shows both the cosimulation and HDL model (DUT) producing a 79ms delayed version of the original signal produced by the behavioral model, as expected, with no difference between the two waveforms. The 79ms delay is due to the original 55ms delay added in the `HDL Algorithm` subsystem to enable pipelining by the synthesis tool and an additional 24ms delay due to the delay balancing added by HDL code generation. The additional 24 delays are reported during the code generation step.



The Simulink scopes comparing the results of the cosimulation can be found in test bench model inside the `Compare` subsystem, which is at the output of the `HDL Algorithm_mq` subsystem.

Use this command to open the subsystem with the scopes.

```
open_system([modelname,'/Compare/Assert_beamformingOutHDL'])
```

Summary

This example is the second of a two-part tutorial series on how to automatically generate HDL code for a fixed-point, sample-based beamforming algorithm and verify the generated code in Simulink. The first part of the tutorial, “FPGA-Based Beamforming in Simulink: Algorithm Design” on page 5-2, shows how to develop an algorithm in Simulink suitable for implementation on an FPGA. This example showed how to set up a model to generate the HDL code and a cosimulation test bench for a Simulink subsystem created with blocks that support HDL code generation. It showed how to set up and launch ModelSim to cosimulate the HDL code and compare its output to the output generated by the HDL implementation model.

FPGA-Based Monopulse Technique: Algorithm Design

This example shows the first half part of a workflow to develop a monopulse technique where the signal is downconverted using digital downconversion (DDC). The model in this example is suitable for implementation on FPGA. This example focuses on the design of monopulse technique to estimate the azimuth and elevation of an object.

The second part of the example, “FPGA-Based Monopulse Technique: Code Generation” on page 5-27, shows how to generate HDL code from the implementation model and verify that the generated HDL code produces the correct results compared to the behavioral model. The entire algorithm is designed using fixed-point data types.

The example shows how to design an FPGA-ready monopulse technique to match a corresponding behavioral model in Simulink® using the Phased Array System Toolbox™, DSP HDL Toolbox™, and Fixed-Point Designer™. To verify the implementation model, the example compares the simulation output of the implementation model with the output of the behavioral model.

The Phased Array System Toolbox provides the floating-point behavioral model for the monopulse technique as a `phased.MonopulseFeed` System object™. DSP HDL Toolbox provides the FIR filter essential for the downconversion filtering.

Fixed-Point Designer provides data types and tools for developing fixed-point and single-precision algorithms to optimize performance on embedded hardware. Bit-true simulations can be performed to observe the impact of limited range and precision without implementing the design on hardware.

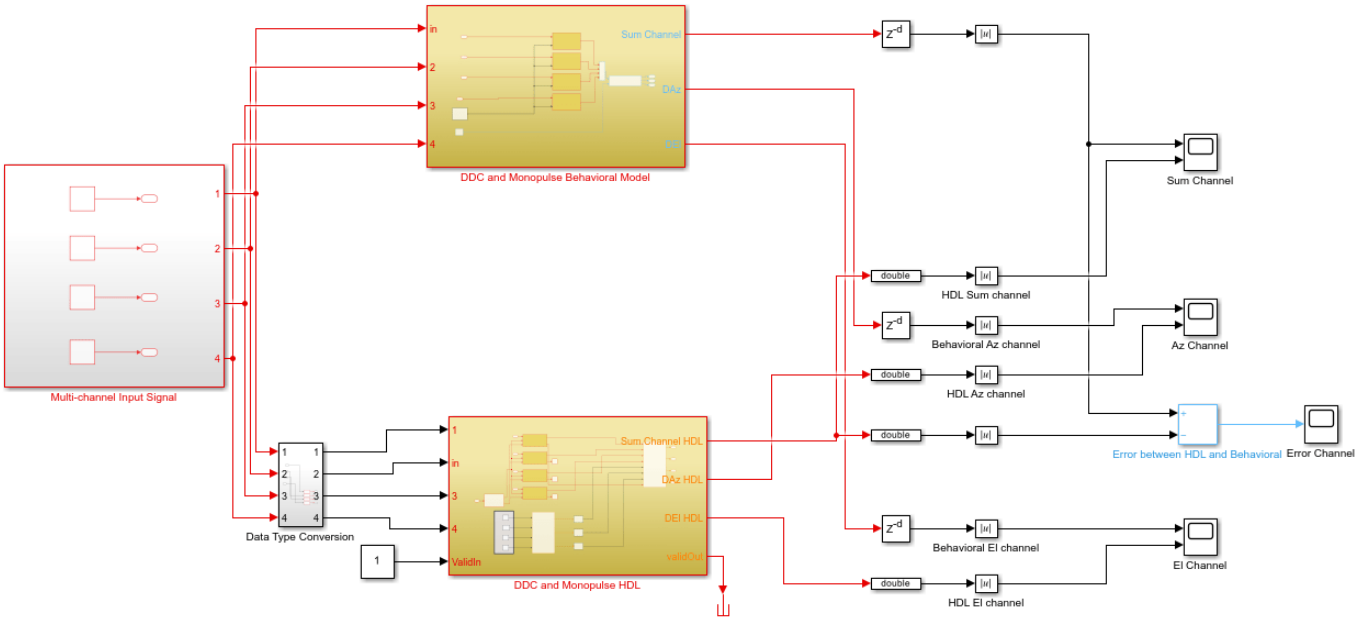
Monopulse is a technique where the received echoes from different elements of an antenna are used to estimate the direction of arrival (DOA) of a signal. This direction helps estimate the location of an object. The example uses DSP HDL Toolbox and Fixed-Point Designer to design the algorithm. This technique uses four beams to measure the angular position of the target. All the four beams are generated simultaneously and the difference of azimuth and elevation is achieved in a single pulse, hence, the name monopulse.

Designing the Subsystem

The algorithm is implemented by using Simulink® blocks that support HDL code generation. The model assumes that the signal is received from the 4-element uniform rectangular array (URA), so, the model has 4 sinusoids as inputs. Assuming a 4-element URA, the model is comprised of 4 receive channels from each of the elements of the URA. Once the signals are converted to the digital domain, DDC blocks ensure that the frequency of the received signal is lowered to reduce the sample rate for processing. The block diagram shows the subsystem which consists of the following modules.

1. Multi-Channel Input Signal
2. Digital Down-Conversion
3. Monopulse Sum and Difference Channels

```
modelname = 'SimulinkDDCMonopulseHDLWorkflowExample';
open_system(modelname);

% Ensure model is visible and not obstructed by scopes.
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```



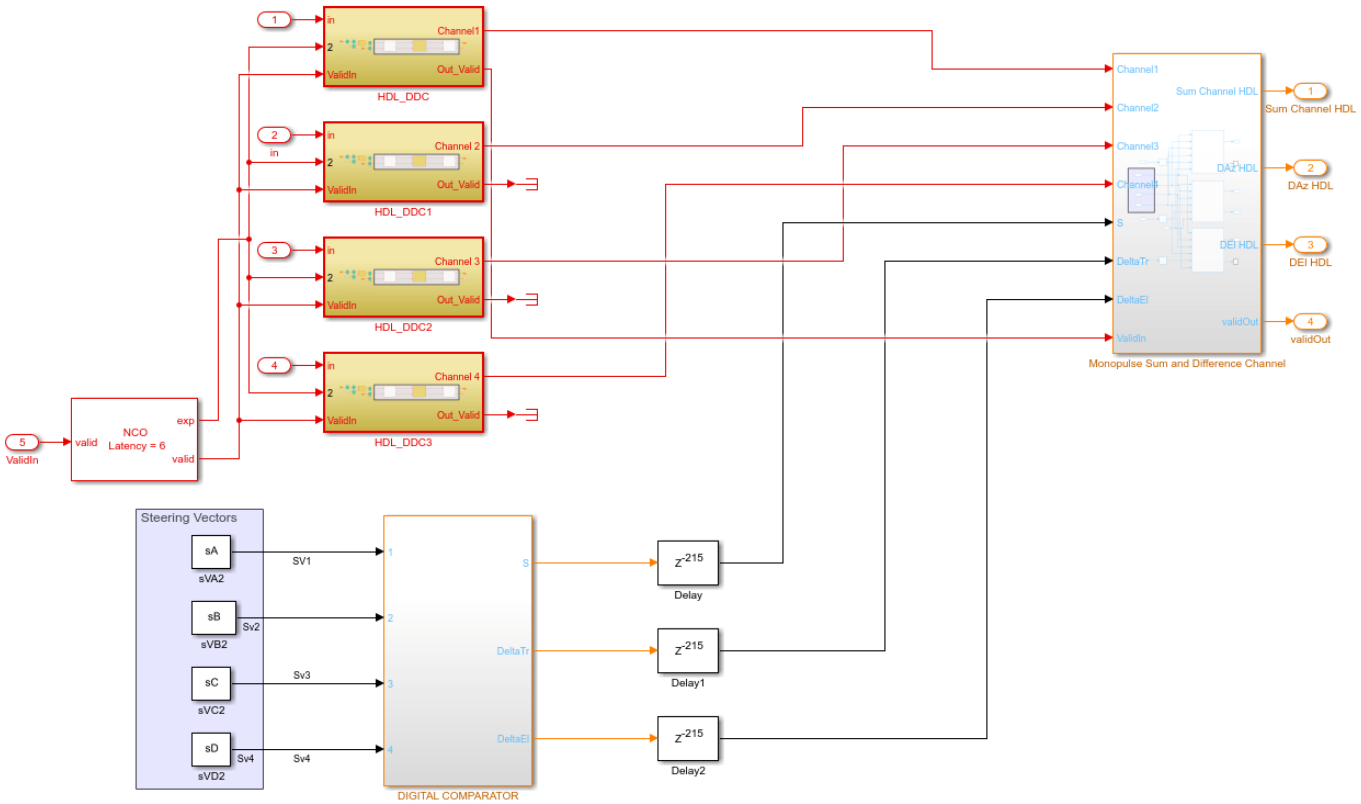
The Simulink model has two branches. The top branch is the behavioral floating-point model of the monopulse technique and digital downconversion algorithm and the bottom branch is the functionally equivalent fixed-point version using blocks that support HDL code generation. Apart from plotting the output of both branches to compare the two, the difference, or error, between sum channel of both the outputs has also been calculated and plotted.

The output of the behavioral model has a delay (Z^{-220}) block. This delay is needed because the implementation algorithm uses 220 delays to enable pipelining which creates latency that needs to be accounted for. This latency is necessary to time-align the output between the behavioral model and the implementation model.

Digital DownConversion (DDC)

The `DDC and Monopulse HDL` subsystem shows how the received signal sampled at 80 MHz and nearly 15 MHz carrier frequency is downconverted to baseband via the DDC and then passed to the monopulse sum and difference subsystem. A DDC module is a combination of a numerically controlled oscillator (NCO) and a set of low-pass filters. The NCO block provides the signal to mix and demodulate the incoming signal.

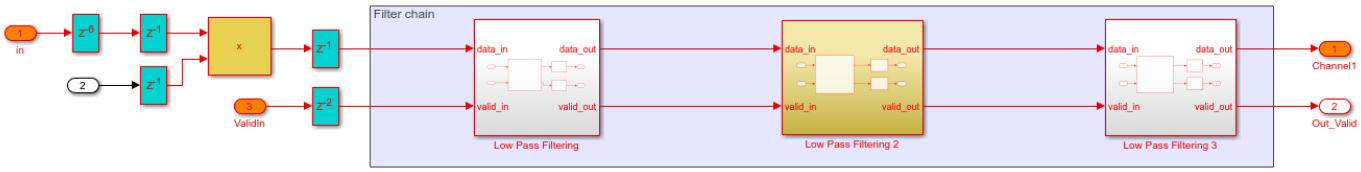
```
open_system([modelname '/DDC and Monopulse HDL']);
```



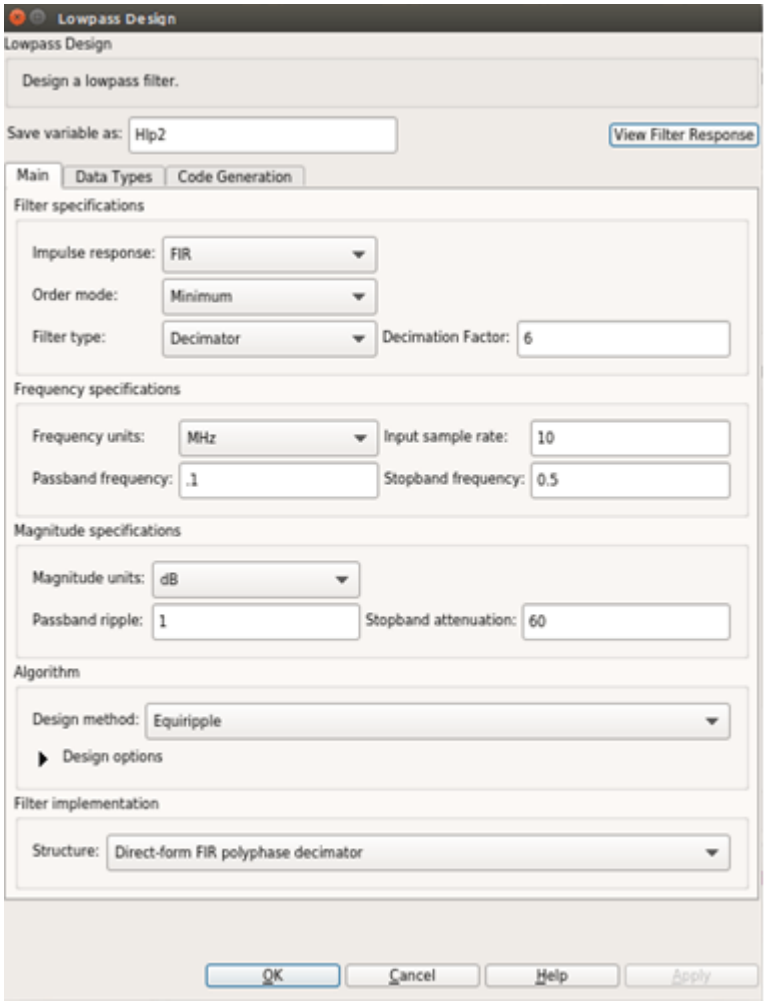
The delay of 215 ms at the sum and difference output of the steering vectors in the implementation subsystem compensates for the delay of the downconversion chain.

A DDC also contains a set of lowpass filters as shown in the figure. Once mixed, the lowpass filtering of the mixed signal is required to eliminate the high frequency components. This example uses a cascaded filter chain for lowpass filtering. The NCO generates the high-accuracy sinusoid for the mixer. The NCO block has a latency of 6 cycles. This signal is mixed with the incoming signal and is converted from a higher frequency to a relatively lower frequency as it progresses through the filter stages.

```
open_system([modelname '/DDC and Monopulse HDL/HDL_DDC']);
```



In this example, the incoming signal has a carrier frequency of 15 MHz and is sampled at 80 MHz. The downconversion process brings the sampled signal down to a few kHz. The coefficients for the lowpass FIR filters are designed using `filterBuilder`. The coefficient values must be chosen to satisfy the required passband criteria.

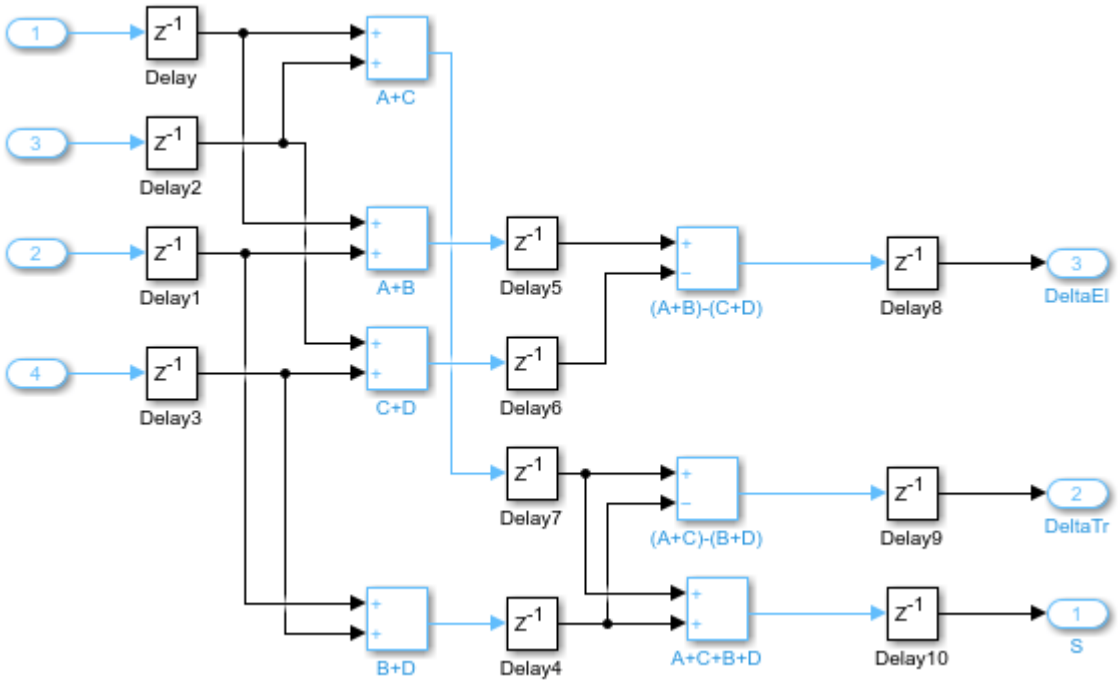


Once generated, the coefficients are used to configure the FIR Filter block.

Monopulse Sum and Difference Channels

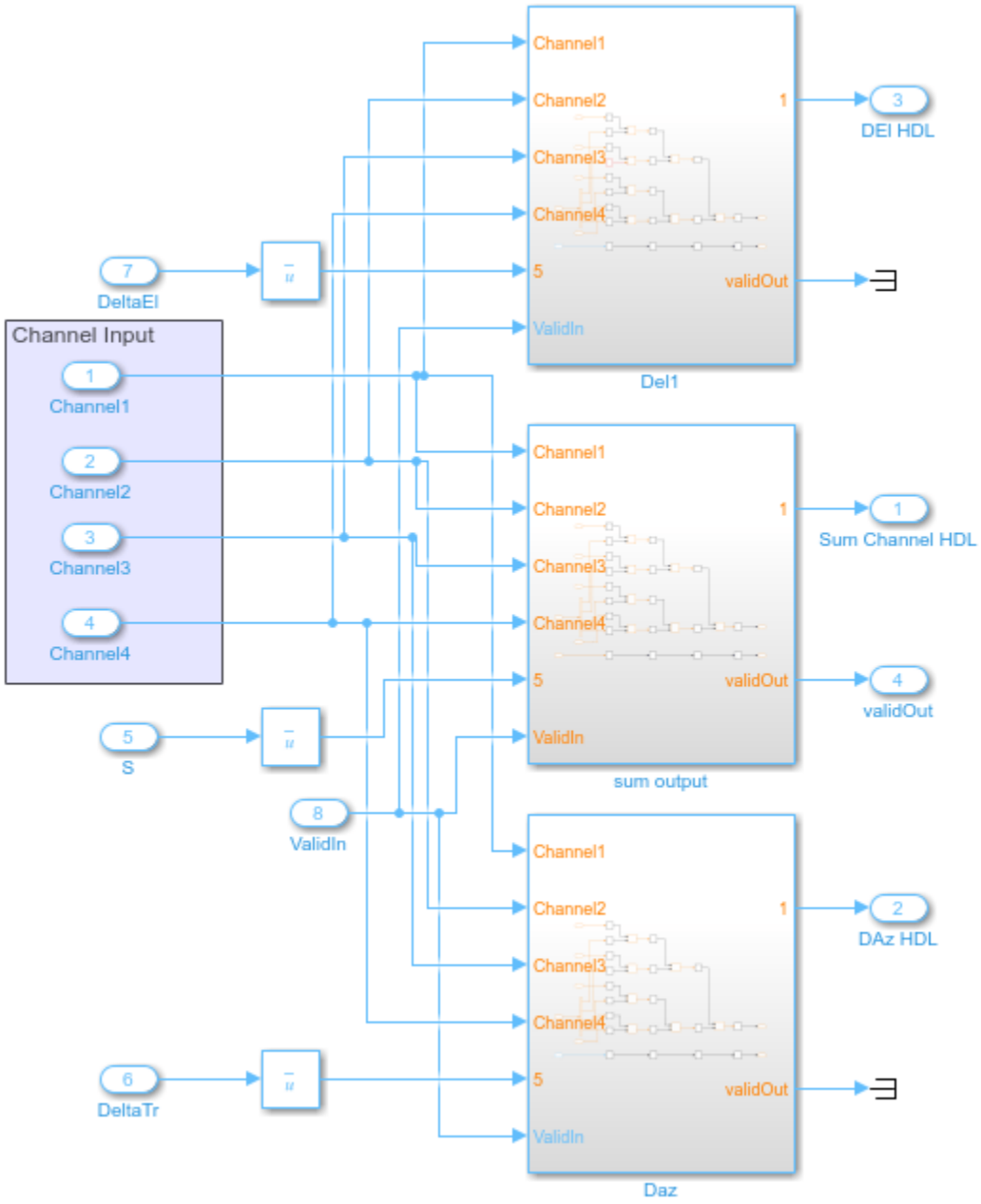
The monopulse algorithm must also generate a steering vector for different elements. The steering vectors have been generated for an incident angle of azimuth 30 degrees and elevation 20 degrees. The steering vectors are passed to the digital comparator to provide the desired sum and difference channel outputs. The downconverted signal is then multiplied by the conjugate of these vectors as shown in the figure. By processing the sum and difference channels, the DOA of the received signal can be found. The digital comparator that compares the steering vectors for the different elements of the antenna array is shown.

```
open_system([modelname '/DDC and Monopulse HDL/DIGITAL COMPARATOR']);
```



In the figure, the digital comparator takes the steering vectors and computes the sum and difference of the different steering vectors sVA, sVB, sVC and sVD respectively. You can also calculate the steering vectors by using the `phased.SteeringVector` System object or you can generate them using method similar to the one shown in the “FPGA-Based Beamforming in Simulink: Algorithm Design” on page 5-2. Once the sum and difference of various steering vectors corresponding to each element of the array has been done, the calculation of sum and difference channels for corresponding azimuth and elevation angles are performed. From the Sum and Difference Monopulse subsystem, 3 signals are obtained, namely the sum, the azimuth difference, and the elevation difference. The entire arithmetic is performed in fixed-point data types. Open the monopulse sum and difference channel subsystem by using this command.

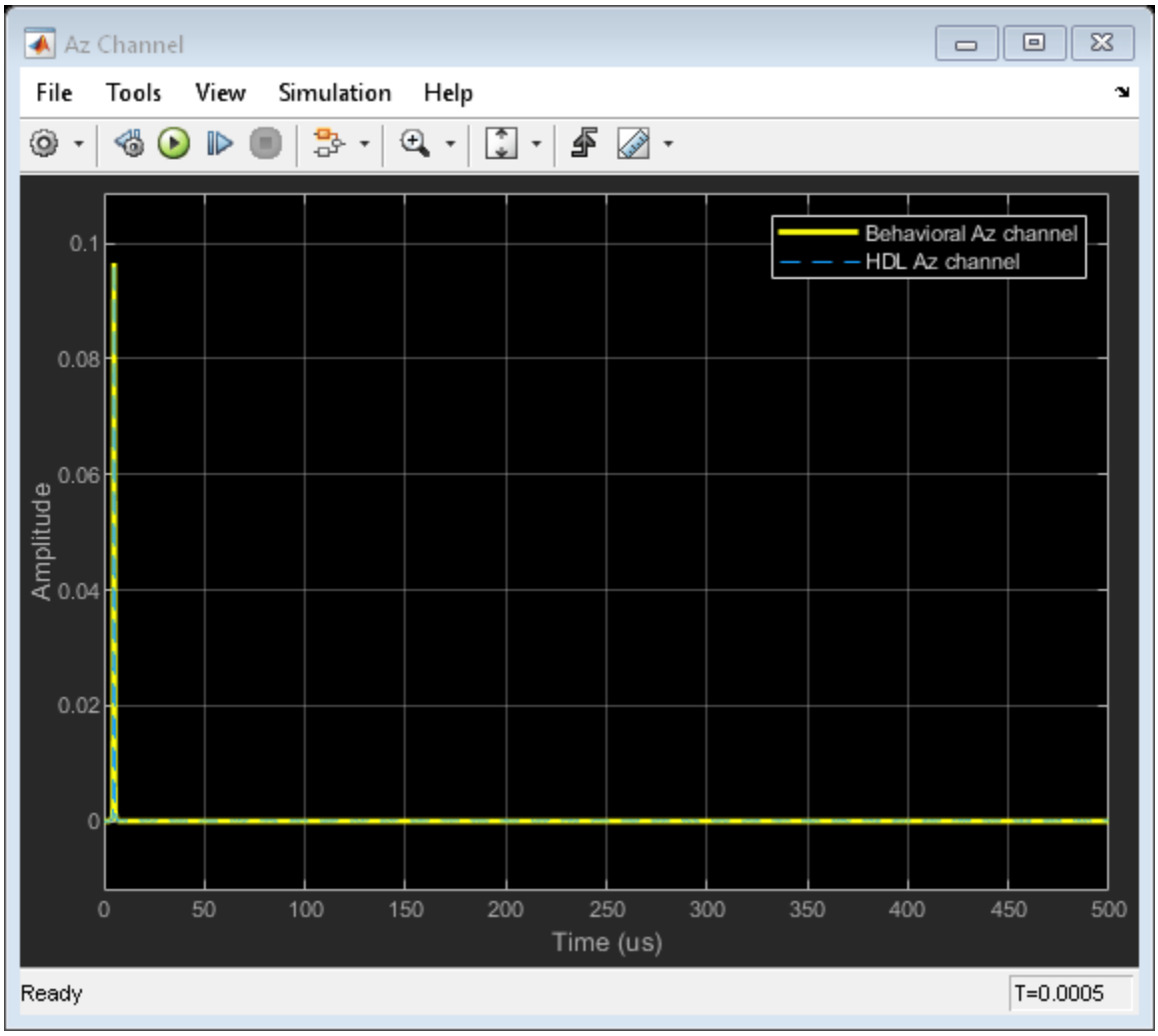
```
open_system([modelname '/DDC and Monopulse HDL/Monopulse Sum and Difference Channel']);
```

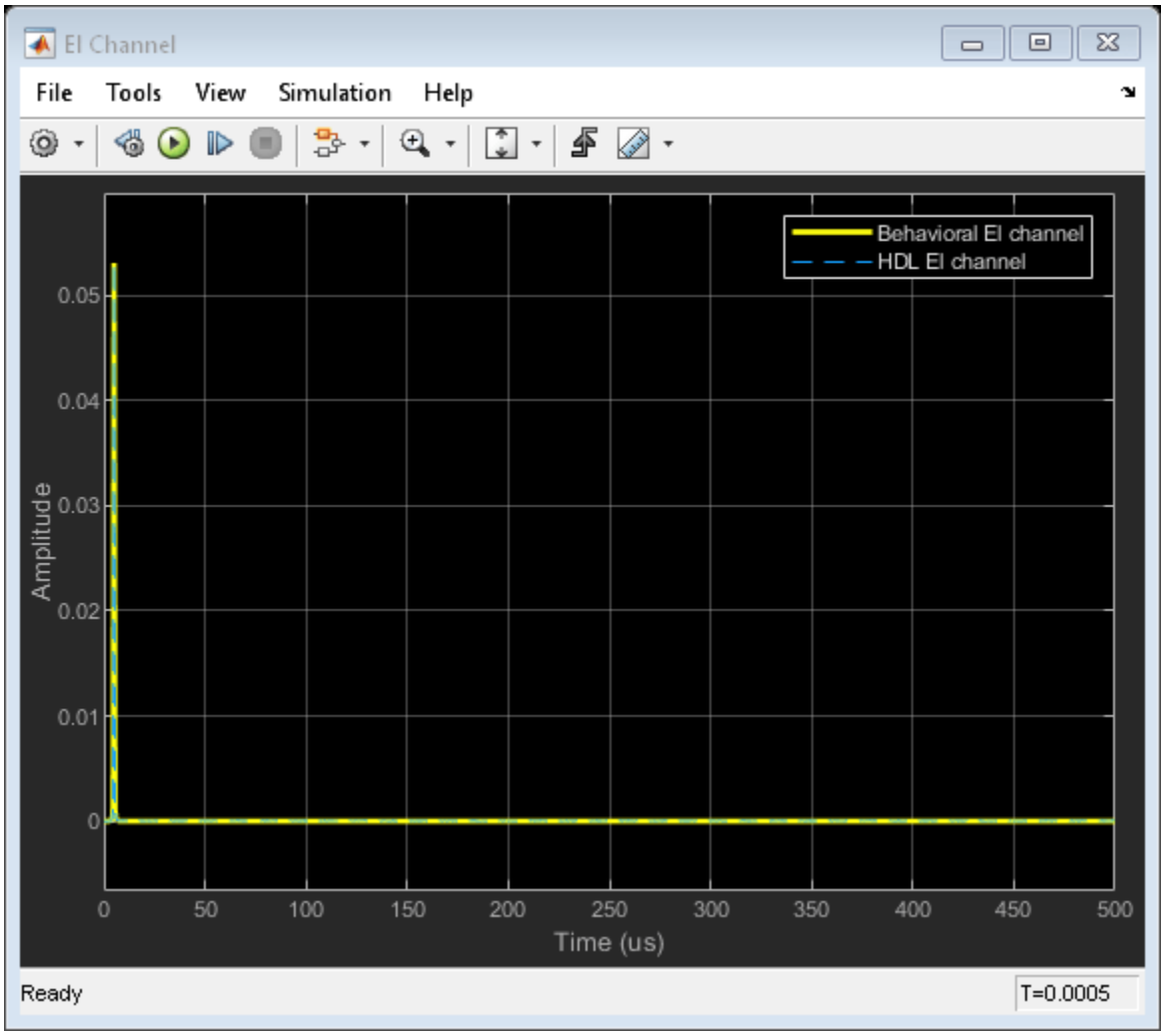


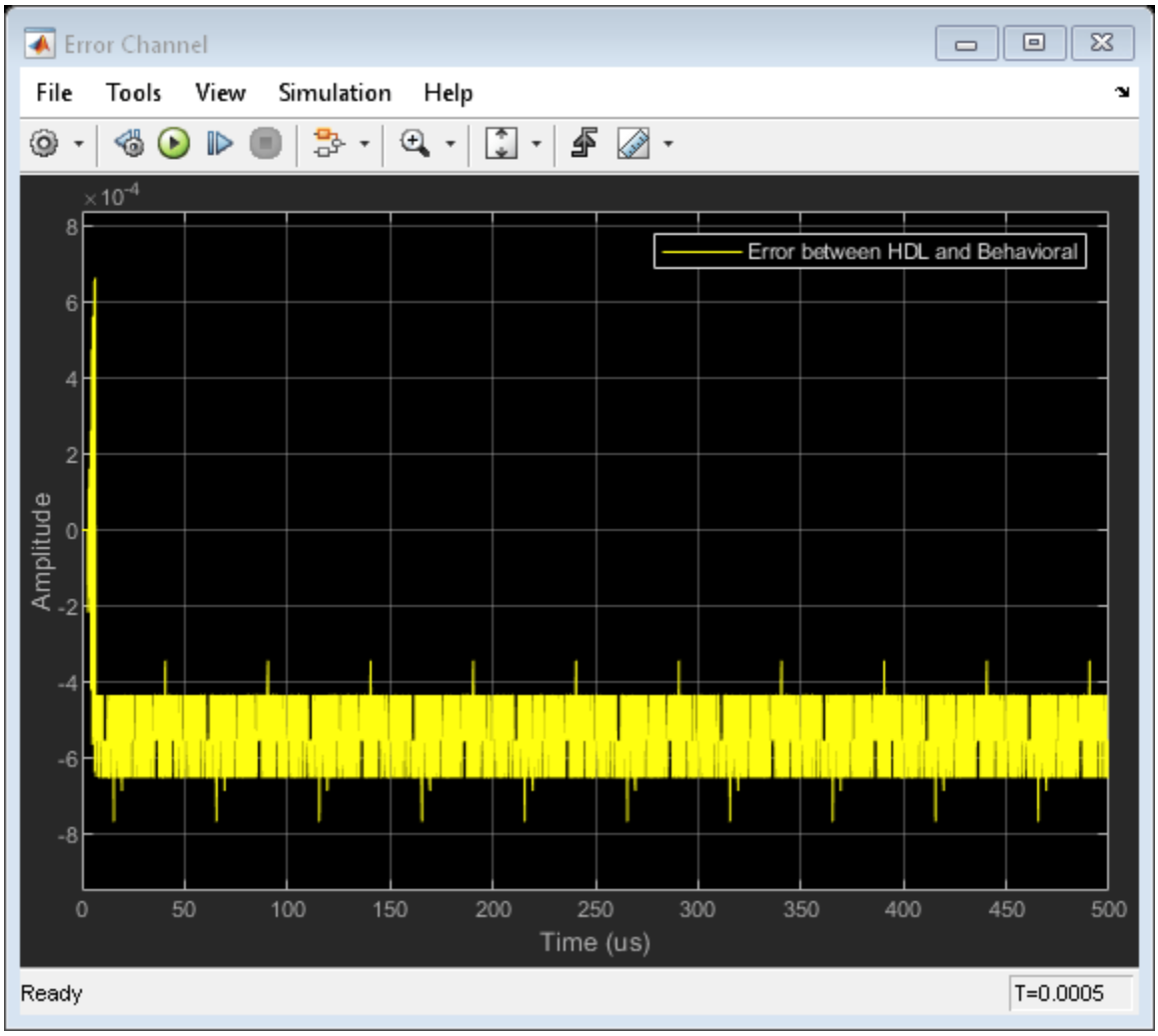
Comparing Results of Implementation Model to Behavioral Model

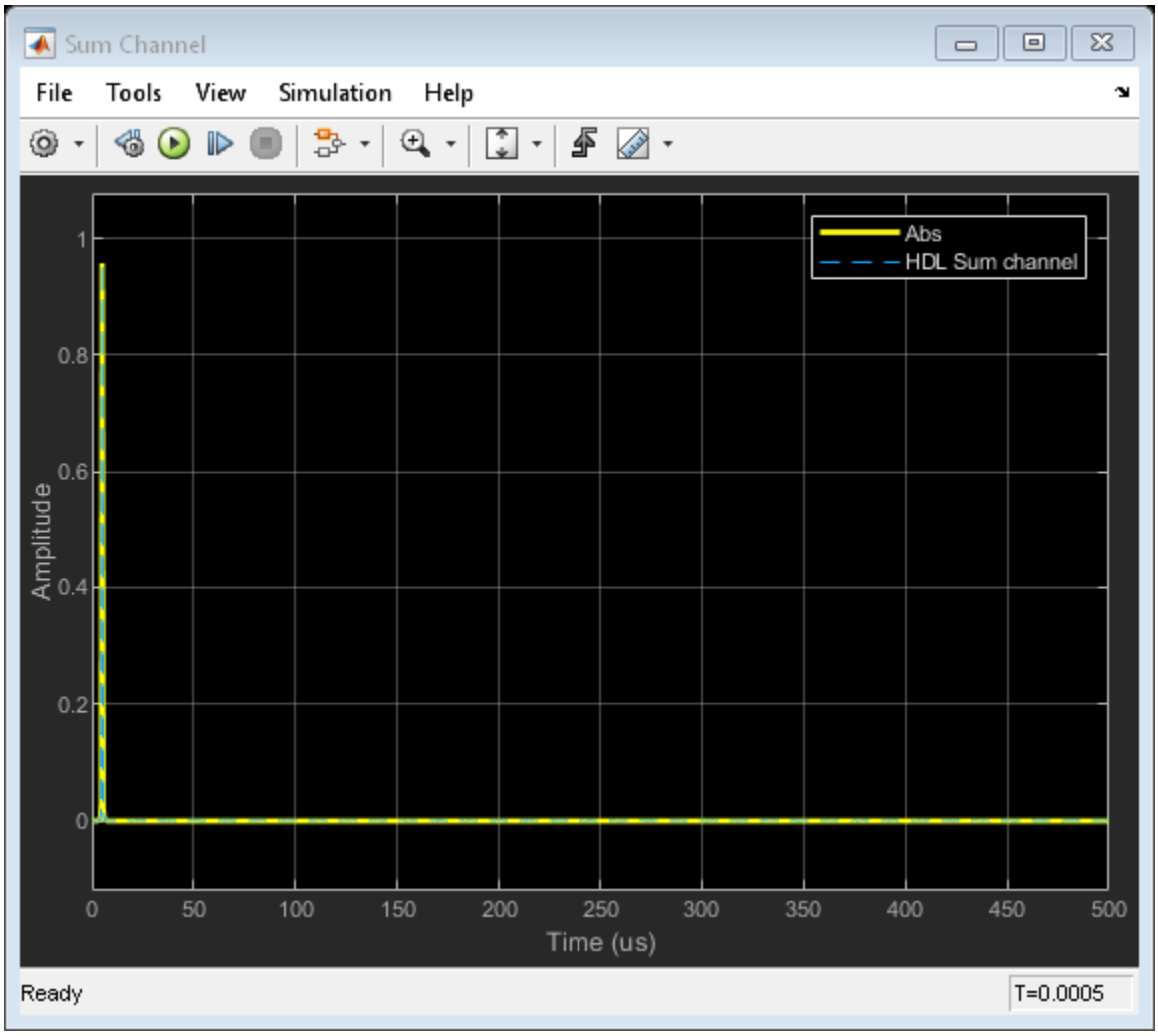
To compare results of the implementation model to the behavioral model, run the model created to display the results. You can run the Simulink model by clicking the Play button or calling the sim command on the MATLAB® command line. Use Scope blocks to compare the output frames.

```
sim(modelname);
```


Az Channel
File
Tools
View
Simulation
Help
Behavioral Az channel
HDL Az channel
Amplitude
0.1
0.08
0.06
0.04
0.02
0
0
50
100
150
200
250
300
350
400
450
500
Time (us)
Ready
T=0.0005


El Channel
File
Tools
View
Simulation
Help
Behavioral El channel
HDL El channel
Amplitude
0.05
0.04
0.03
0.02
0.01
0
0
50
100
150
200
250
300
350
400
450
500
Time (us)
Ready
T=0.0005


Error Channel
File
Tools
View
Simulation
Help
×10^-4
8
6
4
2
0
-2
-4
-6
-8
Amplitude
Error between HDL and Behavioral
0
50
100
150
200
250
300
350
400
450
500
Time (us)
Ready
T=0.0005



The plots show the output from sum and difference channels. These channels can be fed to an estimator to indicate the angle/direction of the object.

Summary

This example demonstrates how to design an FPGA-ready algorithm, automatically generate HDL code, and verify the HDL code in Simulink. The example illustrates the design of a Simulink model for a DDC and monopulse feed system, and verifies the results with an equivalent behavioral model from the Phased Array System Toolbox as the golden reference. Apart from the behavioral model, the example demonstrates how to create a subsystem for implementation using Simulink blocks that support HDL code generation. It also compared the output of the implementation model to the output of the corresponding behavioral model to verify that the two algorithms are functionally equivalent.

Once the implementation algorithm is functionally verified to be equivalent to golden reference, HDL Coder™ can be used for Simulink to HDL code generation and HDL Verifier™ can be used to “Generate a Cosimulation Model” (HDL Coder) test bench.

The second part of this two-part series shows how to generate HDL code from the implementation model and verify that the generated HDL code produces the same results as the floating-point behavioral model as well as the fixed-point implementation model.

FPGA-Based Monopulse Technique: Code Generation

This example shows the second half of a workflow to generate HDL code for a monopulse technique and verify that the generated code is functionally correct.

The first example in the workflow, “FPGA-Based Monopulse Technique: Algorithm Design” on page 5-16, shows how to develop an algorithm in Simulink® suitable for implementation on hardware, such as a field programmable gate array (FPGA), and how to compare the output of the fixed-point implementation model to that of the corresponding floating-point behavioral model.

This example uses HDL Coder™ to generate HDL code from the Simulink model developed in part one and verifies the HDL code using HDL Verifier™. HDL Verifier is used to generate a cosimulation test bench model to verify the behavior of the automatically generated HDL code. The test bench uses ModelSim® for cosimulation to verify the automatically generated HDL code.

The Phased Array System Toolbox™ Simulink blocks model operations on floating-point data and provides the behavioral reference model. This behavioral model is used to verify the results of the implementation model and the automatically generated HDL code.

HDL Coder™ generates portable, synthesizable Verilog® and VHDL® code for Simulink blocks that support HDL code generation.

HDL Verifier lets you test and verify Verilog and VHDL designs for FPGAs, ASICs, and SoCs. This example verifies HDL generated from the Simulink model against a test bench running in Simulink using cosimulation with an HDL simulator.

Implementation Model

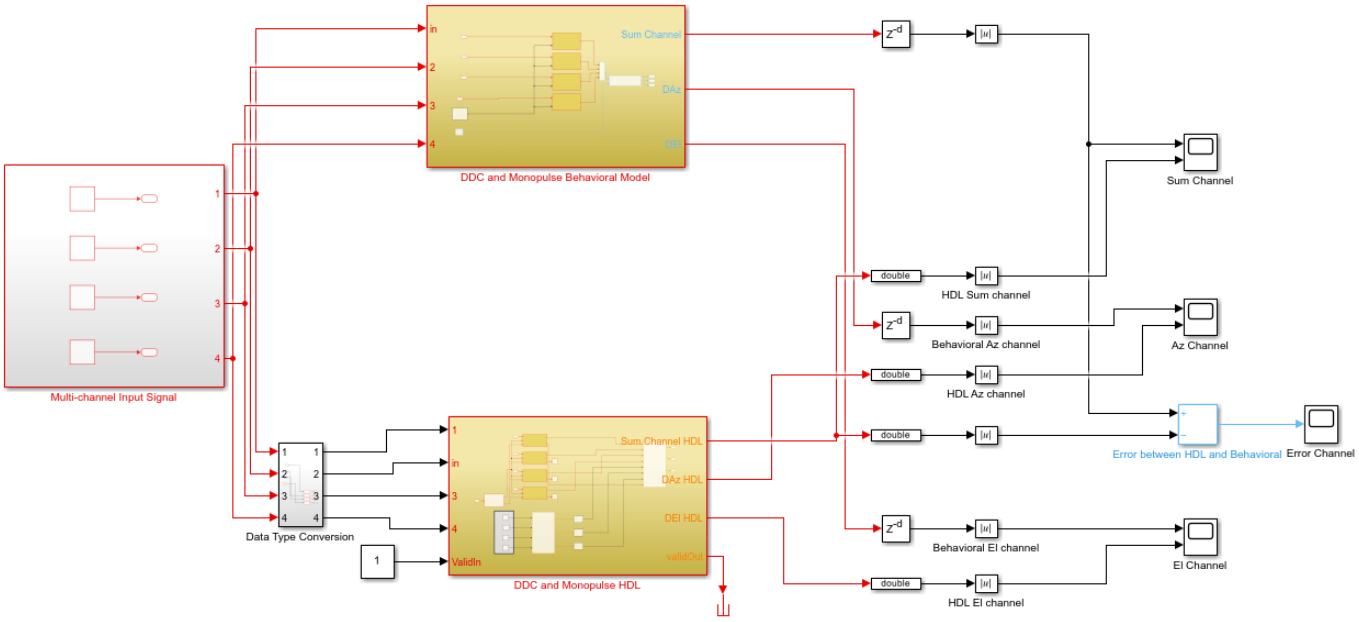
This example assumes that you have a Simulink model that contains a subsystem with a monopulse technique designed using Simulink blocks that use fixed-point arithmetic and support HDL code generation. “FPGA-Based Monopulse Technique: Algorithm Design” on page 5-16 shows how to create such a model.

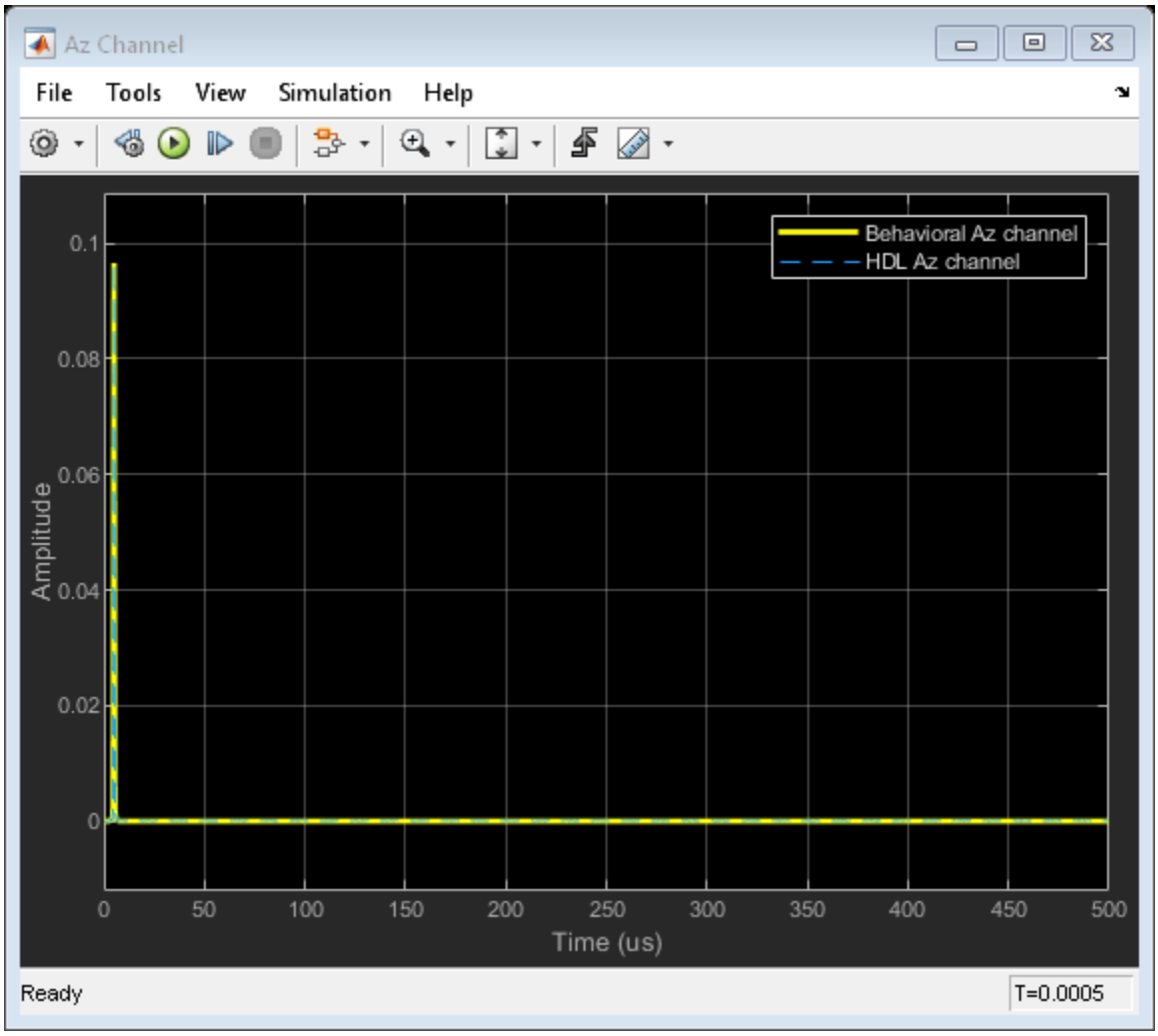
To start with a new model, run the `hdlsetup` (HDL Coder) function to configure the Simulink model for HDL code generation. Open the Model Settings to configure the Simulink model for test bench creation needed for verification. Select **Test Bench** under **HDL Code Generation** in the left panel, and check **HDL test bench** and **Cosimulation model** in the **Test Bench Generation Output** properties group.

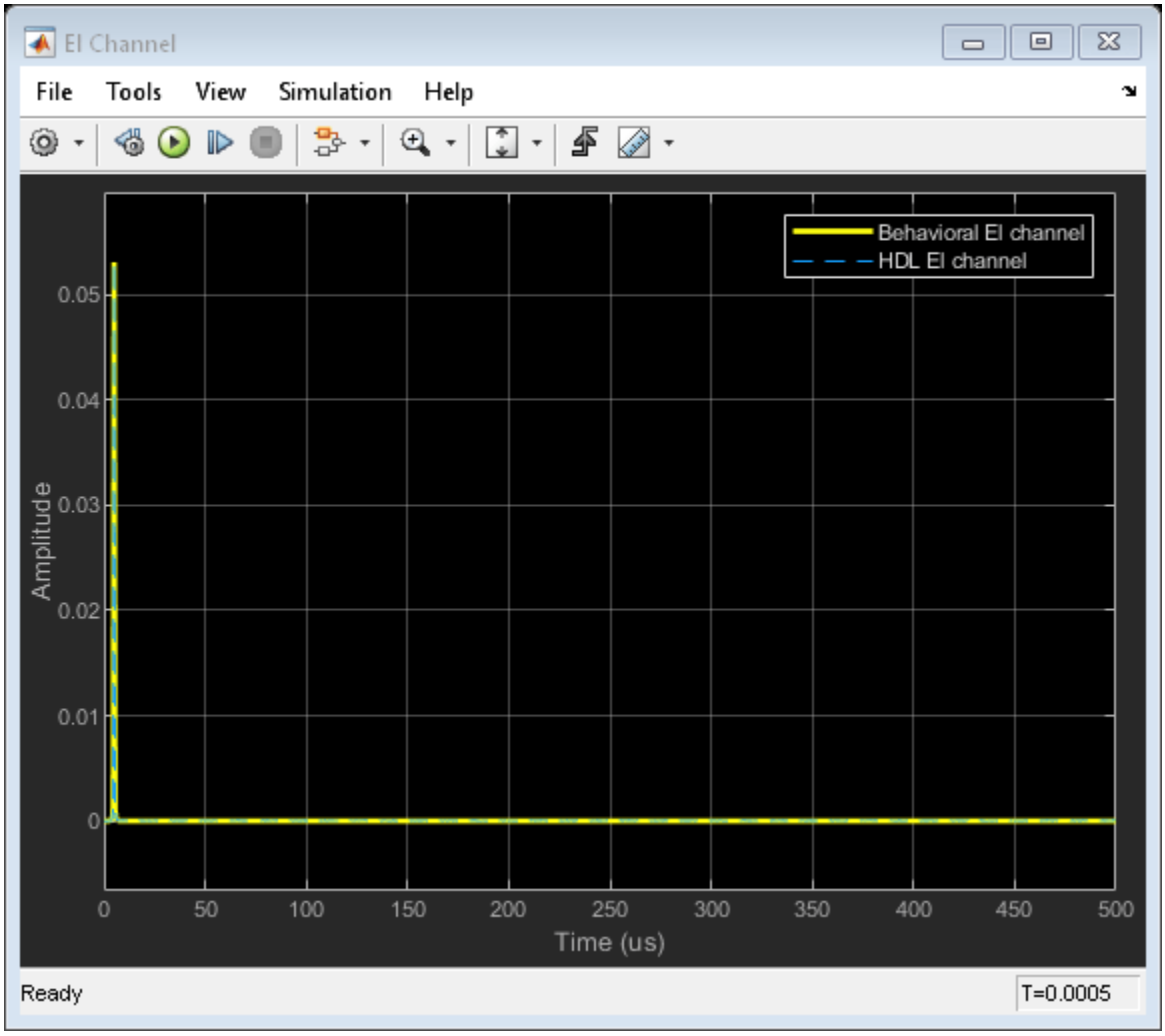
Comparing Results of Implementation Model to Behavioral Model

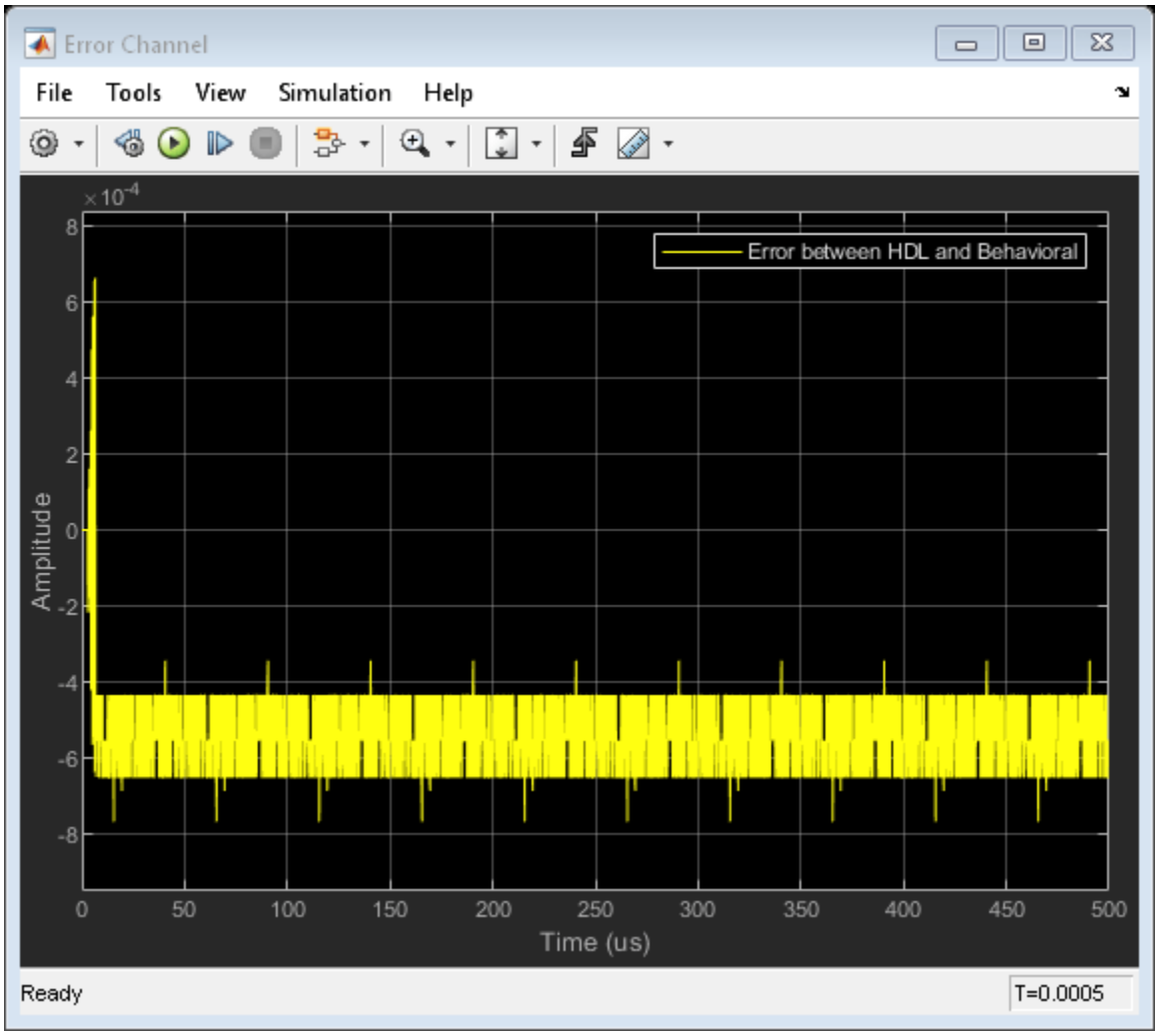
Run the model created in the “FPGA-Based Monopulse Technique: Algorithm Design” on page 5-16 example to display the results. You can run the Simulink model by clicking the Play button or calling the sim command on the MATLAB® command line. Use the Time Scope blocks to compare the output frames visually.

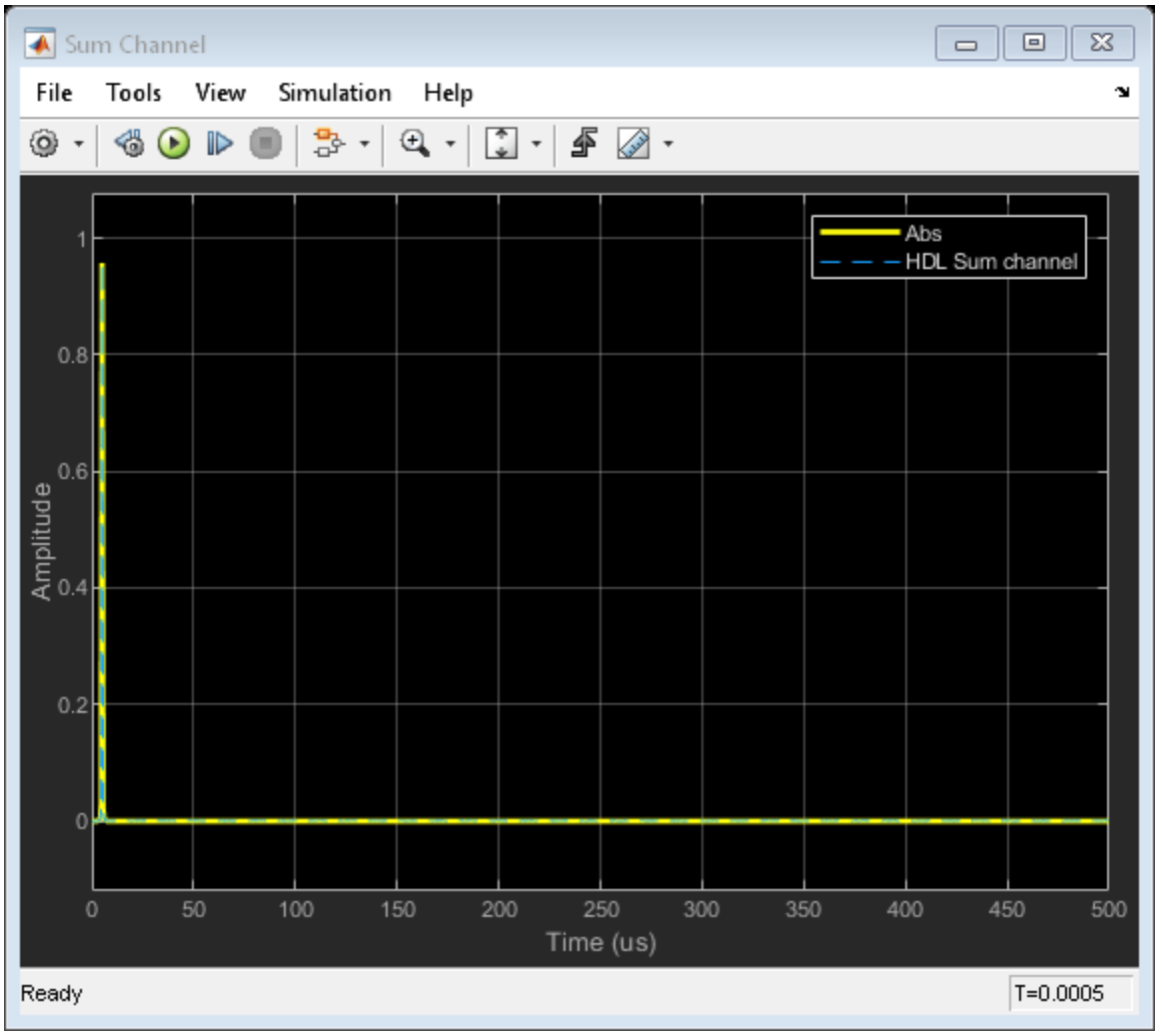
```
modelname = 'SimulinkDDCMonopulseHDLWorkflowExample';
open_system(modelname);

% Ensure model is visible and not obstructed by scopes.
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);

sim(modelname);
```




Az Channel
File
Tools
View
Simulation
Help
Behavioral Az channel
HDL Az channel
Amplitude
0.1
0.08
0.06
0.04
0.02
0
0
50
100
150
200
250
300
350
400
450
500
Time (us)
Ready
T=0.0005


El Channel
File
Tools
View
Simulation
Help
Behavioral El channel
HDL El channel
0.05
0.04
0.03
0.02
0.01
0
Amplitude
0
50
100
150
200
250
300
350
400
450
500
Time (us)
Ready
T=0.0005


Error Channel
File
Tools
View
Simulation
Help
×10^-4
8
6
4
2
0
-2
-4
-6
-8
Amplitude
Error between HDL and Behavioral
0
50
100
150
200
250
300
350
400
450
500
Time (us)
Ready
T=0.0005



Code Generation and Verification

This section shows how to generate HDL code for a DDC and monopulse technique and verify that the generated code is functionally correct. The behavioral model provides the reference values to ensure that the output from HDL is within tolerance limits.

After the fixed-point implementation is verified and the implementation model produces the same results as your floating-point, behavioral model, you can generate HDL code and test bench. For code generation and test bench, set these HDL Code Generation parameters in the Configuration Parameters dialog.

- **Target:** Xilinx Vivado synthesis tool; Virtex7 family; Device xc7vx485t; package ffg1761, speed -1; and target frequency of 300 MHz.
- **Optimization:** Uncheck all optimizations.
- **Global Settings:** Set the Reset type to Asynchronous.
- **Test Bench:** Select HDL test bench, Cosimulation model and SystemVerilog DPI test bench.

HDL Code Generation and Test Bench Creation

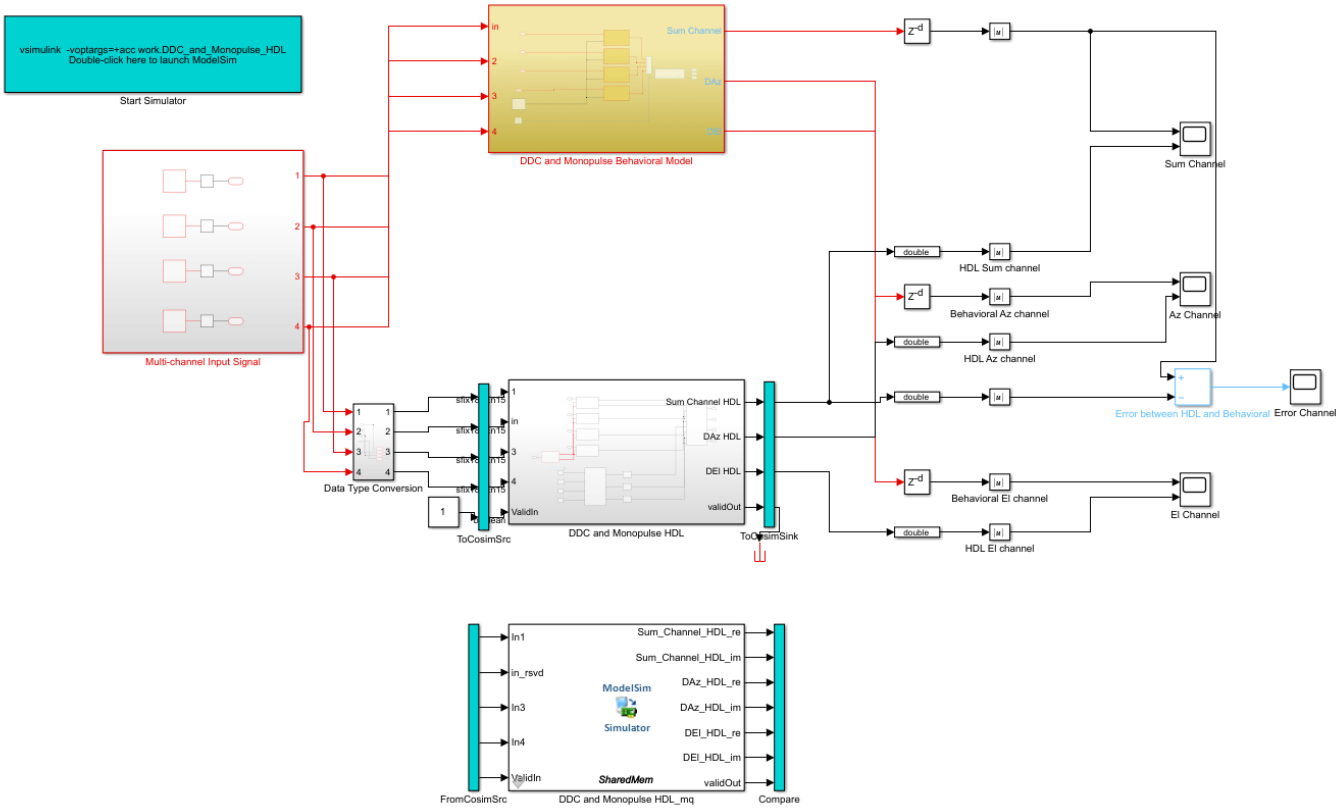
After Simulink Model Settings are updated, you can use HDL Coder to generate HDL Code for the `HDL Algorithm` subsystem and use HDL Verifier to generate test bench model.

Use these commands to generate HDL code and test bench.

```
makehdl([modelname '/DDC and Monopulse HDL']);   % Generate HDL code
makehdltb([modelname '/DDC and Monopulse HDL ']); % Generate Cosimulation test bench
```

Since the model has accounted for pipelining in the multiplications, and disabled code gen optimizations, there are no extra delays added to the model. To align the output of the behavioral model with the implementation model and the cosimulation, you must compensate for the pipeline delays. A delay of (Z^{-215}) is added to the output of the digital comparator. This delay is added to compensate for the latency in the DDC chain. Also, out of the 220 units delay, 215 unit delays compensates for the latency in the DDC chain and 5 units in the monopulse sum and difference subsystem.

When you generate HDL code and test bench, a new Simulink model named `gm_<modelname>_mq` that contains a ModelSim Cosimulation block is created in your working folder.



To open the test bench model, use this command.

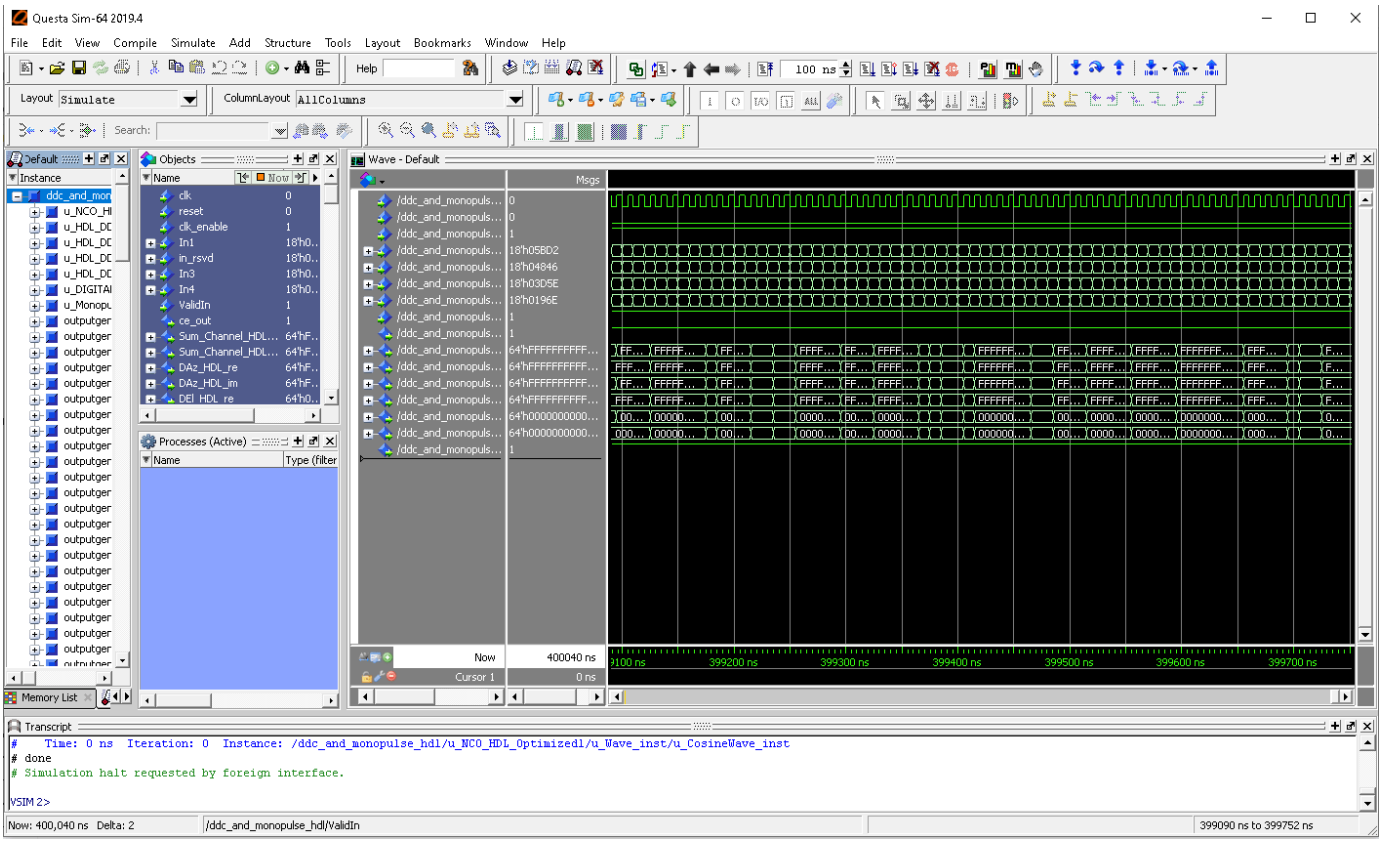
```
modelname = ['gm_',modelname,'_mq'];
open_system(modelname);
```

Launch ModelSim and run the cosimulation model to display the simulation results. You can click on the Play button on the top of Simulink canvas to run the test bench or run it from the MATLAB® command line.

Use this command to run the test bench.

```
sim(modelname);
```

The Simulink test bench model populates the Questa® Sim waveform with the HDL model's signal and Time Scopes in Simulink. The figures show examples of the results in Questa Sim and Simulink scopes.

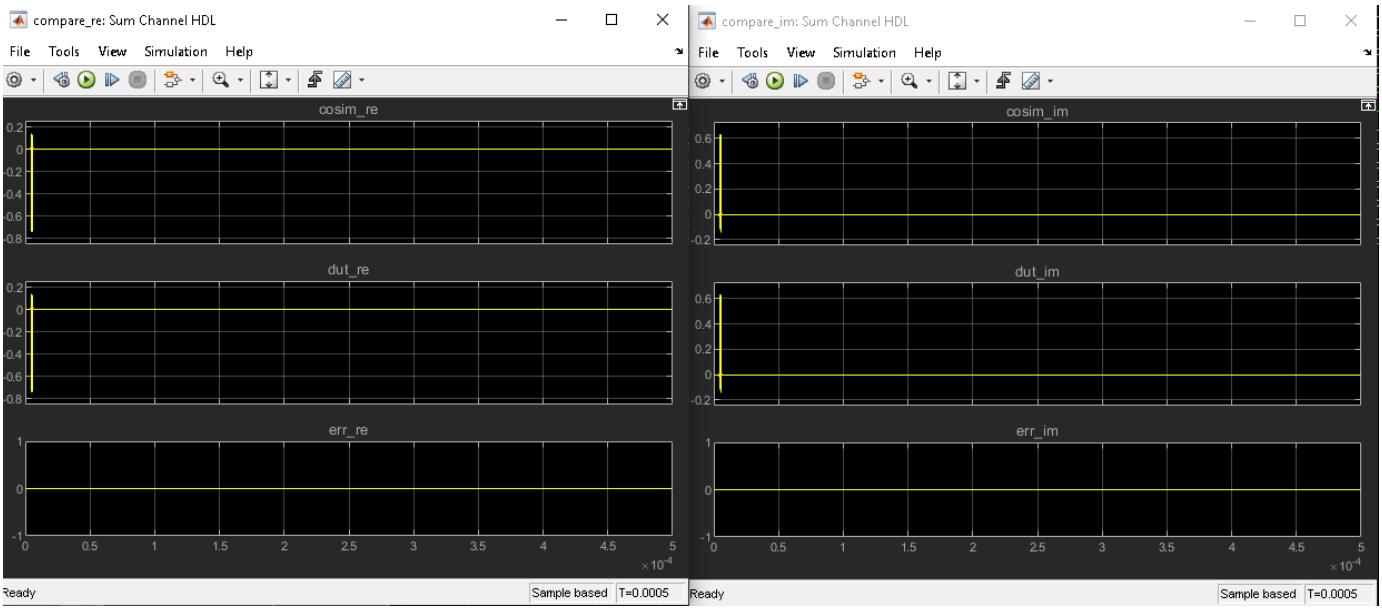


The Simulink scope shows real and imaginary parts for both the cosimulation and design under test (DUT) as well as the error between them.

The Simulink scopes comparing the results of the cosimulation can be found in test bench model inside the `Compare` subsystem, which is at the output of the `DDC and Monopulse HDL_mq` subsystem.

Use this command to open the subsystem with the scopes.

```
open_system([modelname,'/Compare/Assert_Sum Channel HDL'])
```



Summary

The generated HDL code as well as a cosimulation test bench for the Simulink subsystem were created with blocks that support HDL code generation. The example showed how to setup and launch ModelSim to cosimulate the HDL code. The cosimulation is performed via ModelSim for the HDL code and comparison of results to the output generated by the HDL model. The example helped in automatically generating HDL code for a fixed-point monopulse technique and verify the generated code in Simulink.

FPGA-Based Range-Doppler Processing - Algorithm Design and HDL Code Generation

This example shows how to design a range-Doppler response that is ready for FPGA (field programmable gate array) implementation. The example matches the FPGA implementation to a corresponding behavioral model in Simulink® using the Phased Array System Toolbox™.

To verify the functional correctness of the hardware implementation model, the example compares the simulation output of that model with the results of the behavioral model. The term deployment here implies designing a model that is suitable for implementation on an FPGA. The model is deployment-ready and this condition is verified in the example. The hardware implementation is designed using fixed-point data types.

The Phased Array System Toolbox provides the floating-point behavioral model for the range-Doppler response through the `phased.RangeDopplerResponse` System object™. This behavioral model is used to verify the correctness of the implementation model.

Fixed-Point Designer™ provides data types and tools for developing fixed-point and single precision algorithms to optimize performance on an embedded hardware. Bit-true simulations can be performed to observe the impact of limited range and precision without implementing the design in hardware.

This example uses HDL Coder™ to generate HDL code from the developed Simulink model and verifies the HDL code using HDL Verifier™ tools. HDL Verifier is used to generate a cosimulation test bench model to verify the behavior of the automatically generated HDL code. The testbench uses ModelSim® for cosimulation of the generated HDL code.

Range-Doppler Response Algorithm

The `phased.RangeDopplerResponse` System object generates the range-Doppler Response using this algorithm.

1. Fast-time dimension: Filters the signal with a matched filter to generate the range response. The matched filter is an FIR filter with the coefficients set to the time-reverse replica of the transmitted signal.
2. Slow-time dimension: Computes the FFT to generate the Doppler response.

The input data is a matrix of M-by-N values, where M is the number of cells and N is the number of pulses. To calculate the range response, use an FIR filter across the rows (fast-time) and compute the FFT across the columns (slow-time).

Hardware Implementation Model

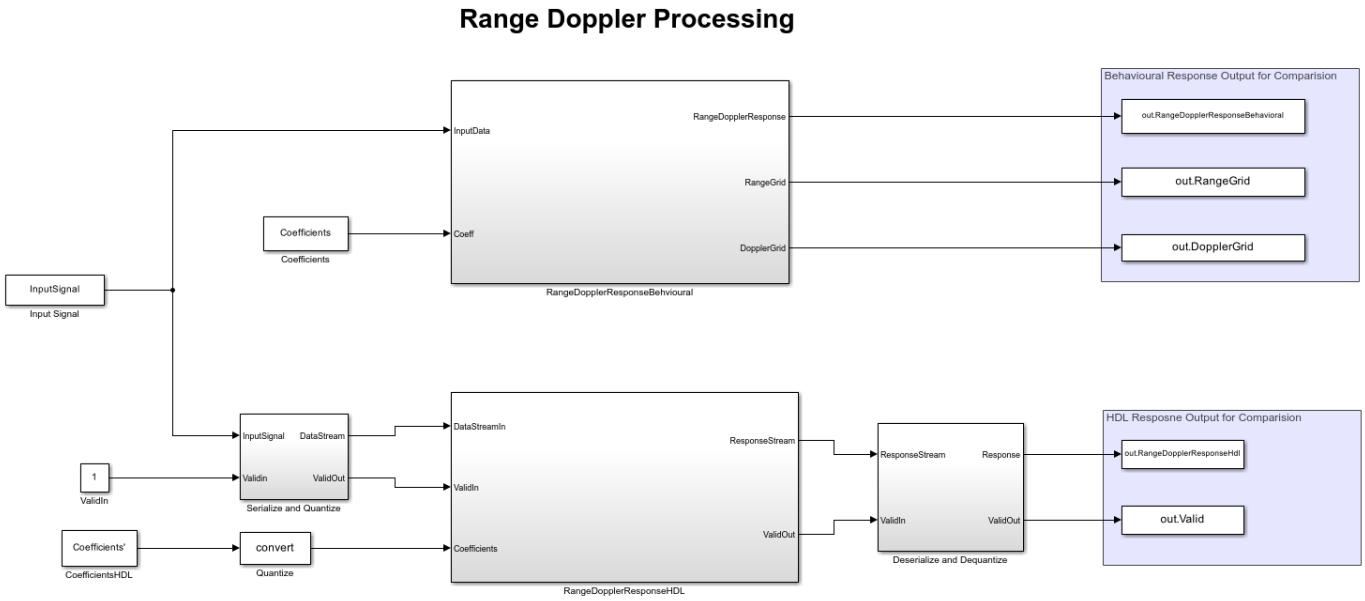
This example reuses the example input data and parameters from the `phased.RangeDopplerResponse (Phased Array System Toolbox)` reference page example.

Serialize and deserialize the signal by using the Serializer1D and Deserializer1D blocks, respectively. These blocks have input and output constraints for code generation, so set the FFT length to 64 and use a subset of the input data cube.

The implementation model uses a word length of 32 bits and a fraction length of 31 bits.

Open the Simulink model by using this command.

```
modelname = 'SimulinkRangeDopplerProcessingHDLWorkflowExample';
open_system(modelname);
% Ensure model is visible and not obstructed by scopes
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```



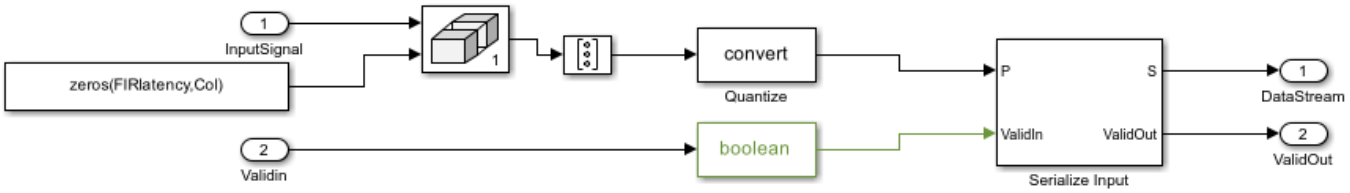
The Simulink model consists of two branches from the Input block. The top branch is the behavioral model with floating-point operations of the `phased.RangeDopplerResponse` System object. The bottom branch is the functionally equivalent implementation model using fixed-point data types, designed with blocks that support HDL code generation from the Simulink HDL Coder Library.

The input and coefficients are generated from the range-Doppler example data. The input is a M-by-N matrix, where M is the number of range cells (fast-time dimension) and N is the number of pulses (slow-time dimension). The output of the `phased.RangeDopplerResponse` object is a M-by-L matrix, where M is the number of range cells and L is the FFT length. Since the input and output of the implementation model have to be data streams, the input is serialized and quantized in the `Serialize and Quantize` subsystem and the output is deserialized to form a range-Doppler matrix map at the output in the `Deserialize and Dequantize` subsystem.

Preprocessing and Postprocessing Data

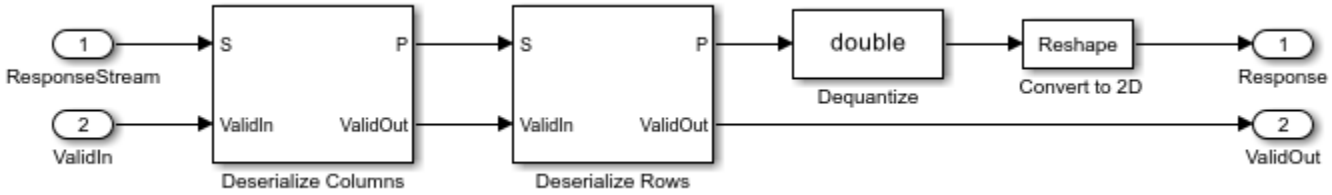
Open the Serialize and Quantize subsystem.

```
open_system([modelname '/Serialize and Quantize'])
```



The input data is zero-padded to accommodate the FIR filter latency by using the Matrix Concatenate block. The data is then serialized by a Serializer1D block used in cascade with the reshape and data-type convert (quantize to fixed-point) block.

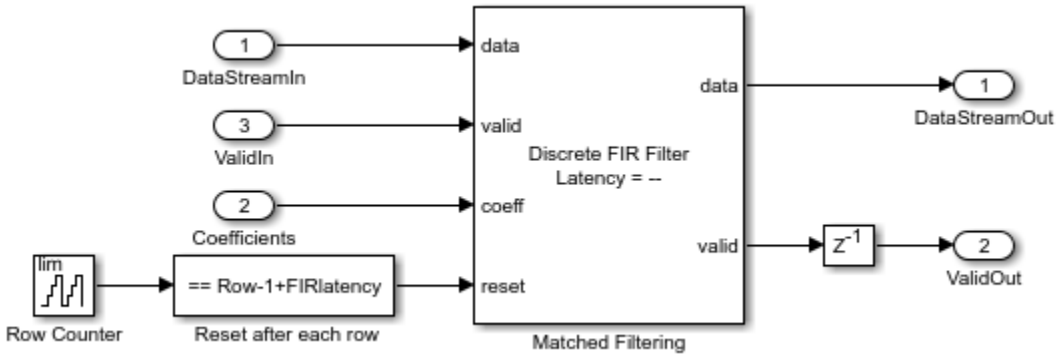
```
open_system([modelname '/Deserialize and Dequantize'])
```



This subsystem uses a Deserializer1D block which uses the data-type convert and reshape blocks to convert the output stream to a range-Doppler map.

Matched Filtering - Range Processing Subsystem

```
open_system([modelname '/RangeDopplerResponseHDL/Matched Filtering - Range Processing'])
```

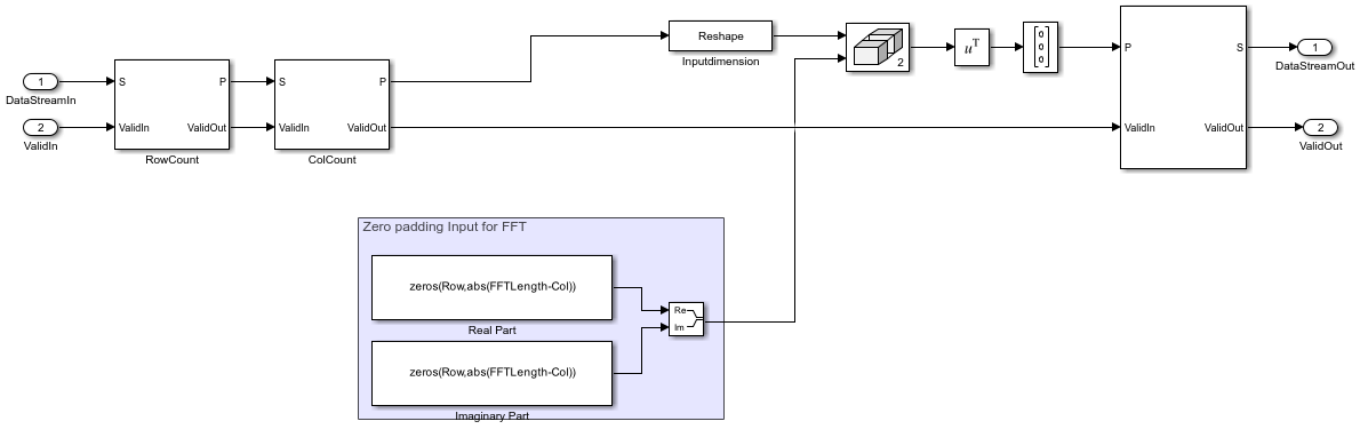


The input stream is processed with a Discrete FIR Filter block with the matching coefficients in the fast-time (row) dimension to get the range response. The block is suitable for hardware implementations and has a latency of seven cycles. The registers in the FIR filter have to be reset to an initial value of 0 after every row. This reset is performed by using a Boolean square wave implemented using a Counter and a Compare To Constant block.

Buffer & Transpose-Column

Open the Buffer & Transpose-Column subsystem.

```
open_system([modelname '/RangeDopplerResponseHDL/Buffer & Transpose - Column'])
```

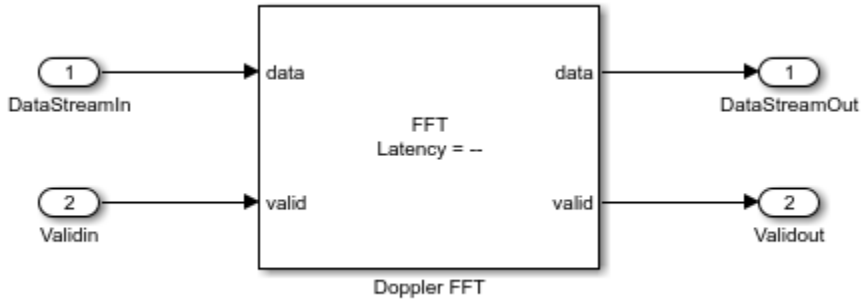


The range-processed data is deserialized using cascaded Deserializer1D blocks and is converted to matrix format with a reshape block. This data is now zero padded and transposed for computing the FFT across the slow-time (column) dimension. The range-processed and transposed data is now serialized again for FFT computation (Doppler processing).

FFT-Doppler Processing

Open the FFT-Doppler Processing subsystem.

```
open_system([modelname '/RangeDopplerResponseHDL/FFT - Doppler Processing'])
```

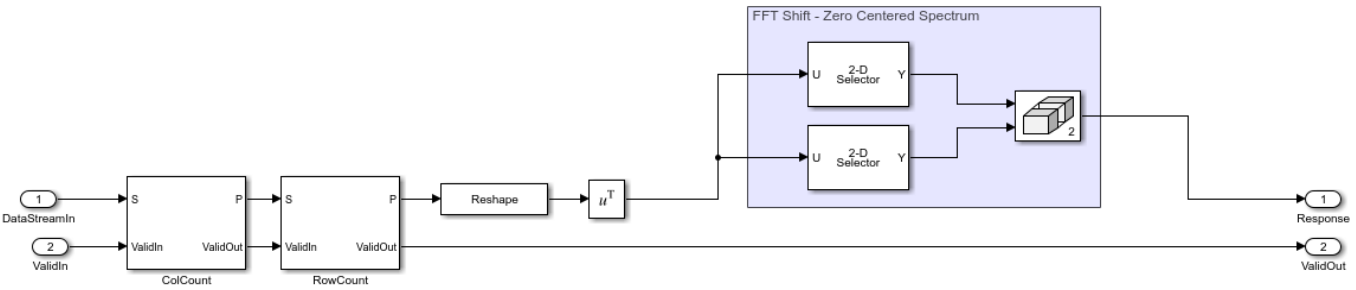


The FFT of the streamed (across column) data is calculated using an FFT block which has a latency of 173 cycles. A streaming radix 2^2 architecture is used with an FFT length of 64.

Buffer-FFT Shift

Open the Buffer-FFT Shift subsystem.

```
open_system([modelname '/RangeDopplerResponseHDL/Buffer - FFT Shift'])
```

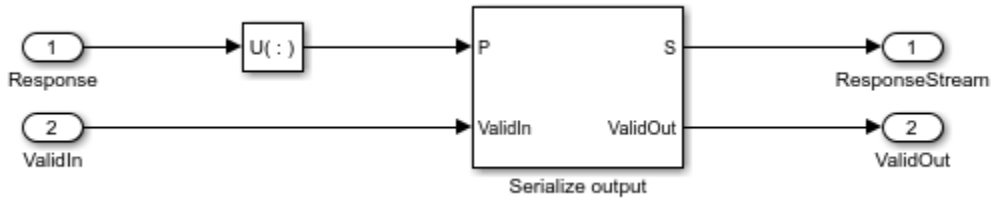


The Doppler-processed serial data is deserialized using Deserialized1D blocks and transposed. The FFT-processed data needs to be rearranged to have a zero-centric spectrum data, which is emulated by using a combination of selector blocks and a matrix concatenate.

Serialize Output

Open the Serialize Output subsystem.

```
open_system([modelname '/RangeDopplerResponseHDL/Serialize Output'])
```



The range-Doppler map data after processing is in the form of a matrix, which is then serialized using a Reshape and a Serializer1D block for the output stream.

Compare of Implementation Model and Behavioral Model Results

Simulate the model by clicking the `Play` button or by using the `sim` command.

```
sim(modelname);
```

To verify the functional correctness of the implementation model, subtract the response matrix of the behavioral model from the response matrix of the implementation model and check that the difference (error) is close to zero (or, within the quantization in the implementation model).

Export the response data from Simulink to the MATLAB® workspace, in the array format. Subtract the behavioral response vector from the implementation model response, reshape the error matrix into a 1-D array, and plot the error with the element index in the x-axis and error in the y-axis. Use the `imagesc` function to display the range-Doppler map. Use the following script to plot the response and error.

```
% Uncomment the following lines of code to visualize the response
% Behavioral Output
behavioralResponse = out.RangeDopplerResponseBehavioral(:,:,1);  % Response from To Workspace Blo
behavioralResponsedB = mag2db(abs(behavioralResponse));          % Convert to dB
rangeGrid = out.RangeGrid(:,1,1);                                % Range Grid
dopplerGrid = out.DopplerGrid(:,1,1);                            % Doppler Grid
```

```
% Visualize the range-Doppler (Behavioral) map

f1 = figure(1);                                                 % Figure handle
f1.Name = 'Behavioral';                                         % Figure Name
fax1 = axes;                                                    % Axis Handle
imagesc(fax1,dopplerGrid,rangeGrid,behavioralResponsedB)

xlabel(fax1,'Doppler');
ylabel(fax1,'Range')
title(fax1,'Behavioral Response')

% HDL Output
idx = find(out.Valid);                                          % Search for valid output
HdlResponse = out.RangeDopplerResponseHdl(:,:,idx(1));          % Response from To Workspace Blo
HdlResponsedB = mag2db(abs(HdlResponse));                       % Convert to dB

% Visualize the range-Doppler (HDL) map

f2 = figure(2);                                                 % Figure handle
f2.Name = 'HDL';                                                % Figure Name
fax2 = axes;                                                    % Axis handle
imagesc(fax2,dopplerGrid,rangeGrid,HdlResponsedB);              % Use behavioral output of range

xlabel(fax2,'Doppler')
ylabel(fax2,'Range')
title(fax2,'HDL Response')

% Error
% Subtract the Behavioral Response from the HDL response
errorMatrix = abs(HdlResponse - behavioralResponse);            % Matrix Subtract

% Convert the error matrix into a row vector which can be visualised on a 2D axis

errorStream = reshape(errorMatrix,1,[]);                        % Convert error matrix to 1D, ro

% Find the index and the maximum error between HDL and behavioral results

Ymax = max(errorStream);                                        % Find Maximum Error
Xmax = find(errorStream == Ymax);                               % Find index of maximum error

% Plot the error on a 2D plot and annotate the maximum error
% between HDL and behavioral response.
f3 = figure(3);                                                 % Figure handle
f3.Name = 'Error';                                              % Figure Name
fax3 = axes;                                                    % Axis handle
plot(fax3,errorStream)                                          % Plot

ylabel(fax3,'Error');
xlabel(fax3,'Data Point Index')
title(fax3,'Error between Behavioral and HDL Model');

textstr = strcat('     ErrorMax = ',num2str(Ymax));
text(fax3,Xmax,Ymax,textstr);
```

The figures show the range-Doppler map and the error between the behavioral and implementation models.

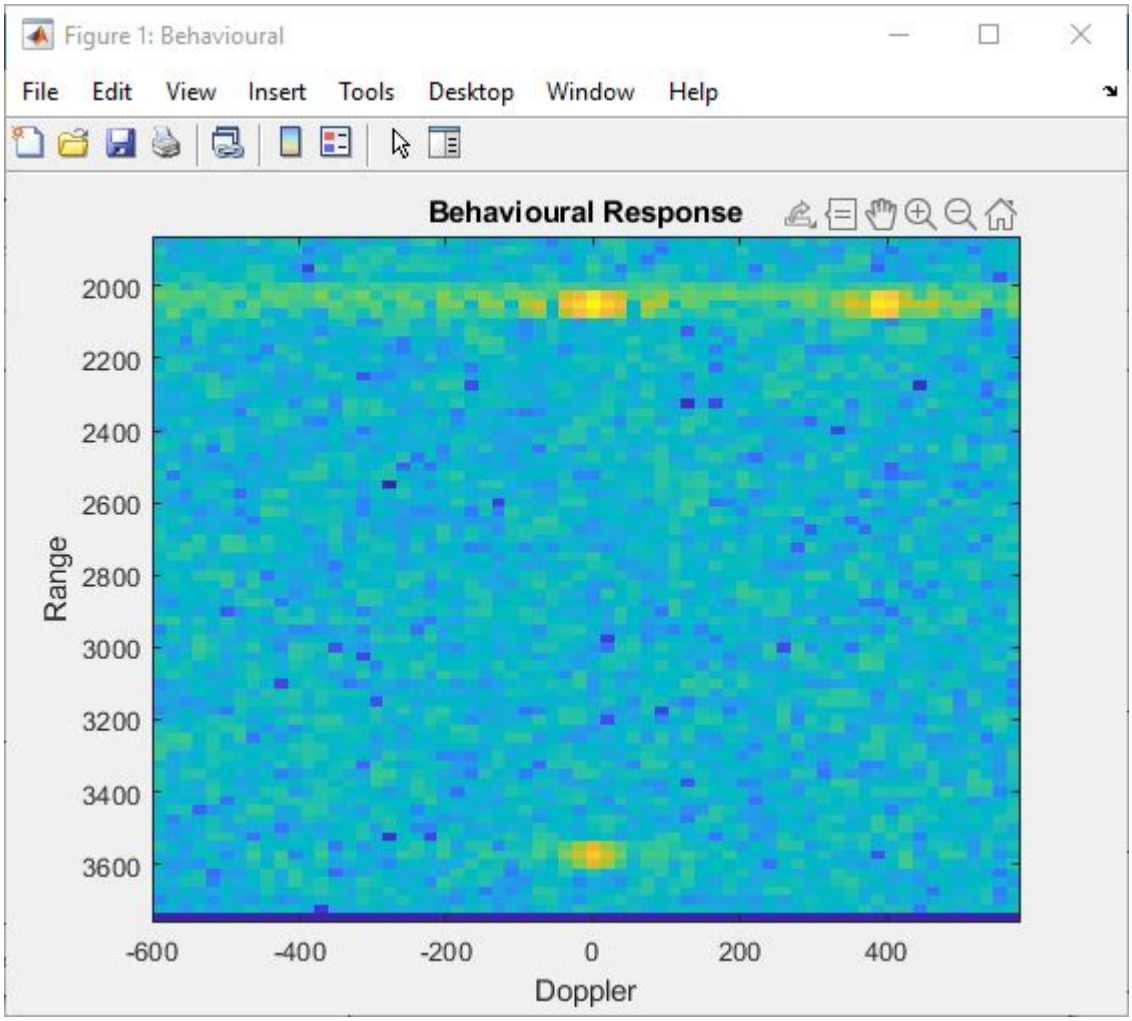


Figure 1: Behavioural
File Edit View Insert Tools Desktop Window Help
Behavioural Response
Range
2000
2200
2400
2600
2800
3000
3200
3400
3600
-600
-400
-200
0
200
400
Doppler

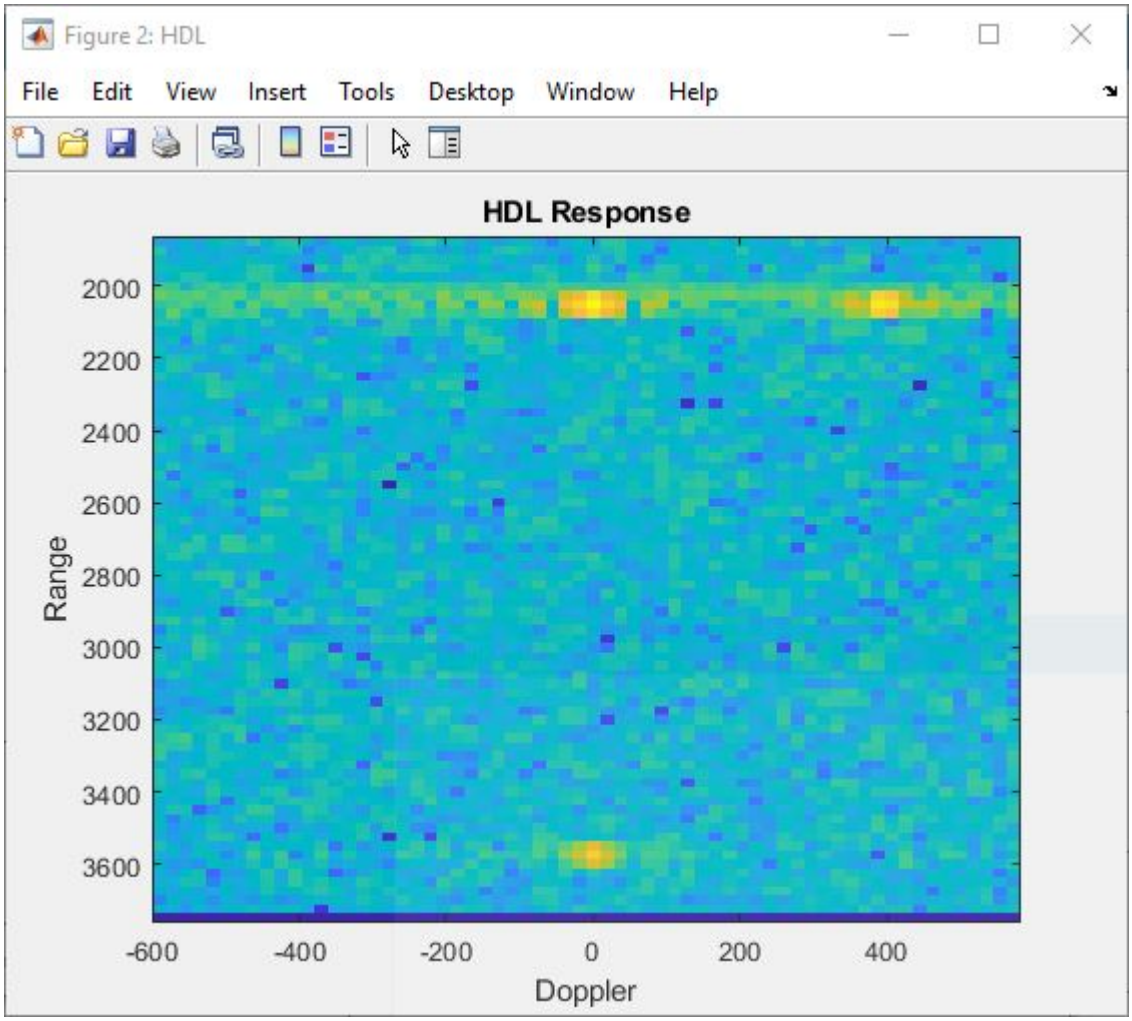
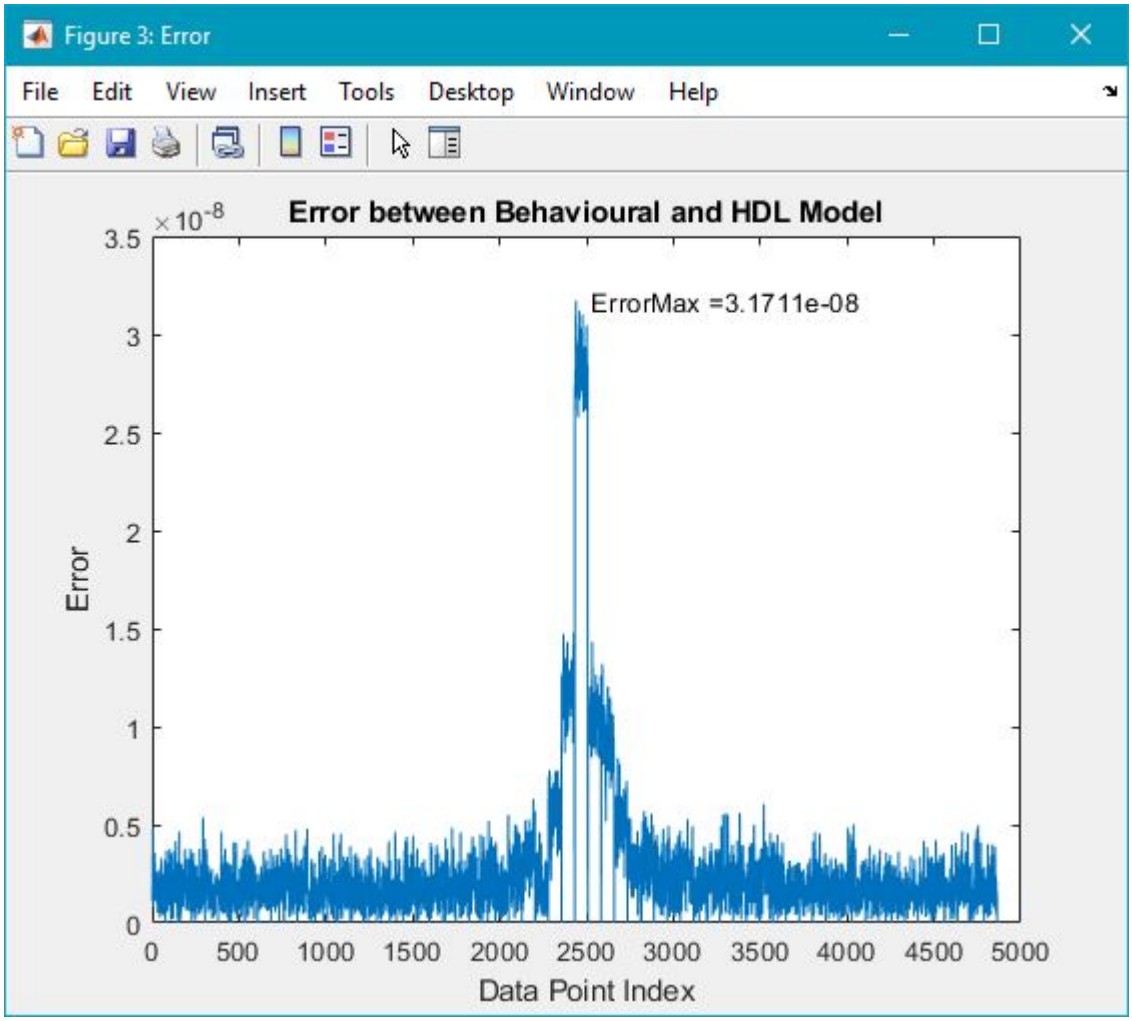


Figure 2: HDL
File Edit View Insert Tools Desktop Window Help
HDL Response
Range
2000
2200
2400
2600
2800
3000
3200
3400
3600
-600
-400
-200
0
200
400
Doppler



Code Generation and Verification

This section covers the procedure to generate HDL code for the range-Doppler response implementation model and verifying the functional correctness. The behavioral model provides the reference values to ensure that the output from HDL model is within tolerance limits. Based on the Simulink model setup described in the earlier sections, the implementation model is designed using fixed-point arithmetic blocks that support HDL code generation. Alternatively, if you start with a new model, you can run the `hdlsetup` function to configure the Simulink model for HDL code generation. To configure the Simulink model for test bench creation, open **Model Settings**, select **Test Bench** under **HDL Code Generation** in the left panel, and check **HDL test bench** and **Cosimulation model** in the **Test Bench Generation Output** properties group.

Model Settings

After the fixed-point implementation is verified and the implementation model produces the same results as your floating-point behavioral model, you can generate HDL code and test bench. For code generation and test bench, set these HDL Code Generation parameters in the **Configuration Parameters** dialog. Set the following parameters in Model Settings under HDL Code Generation:

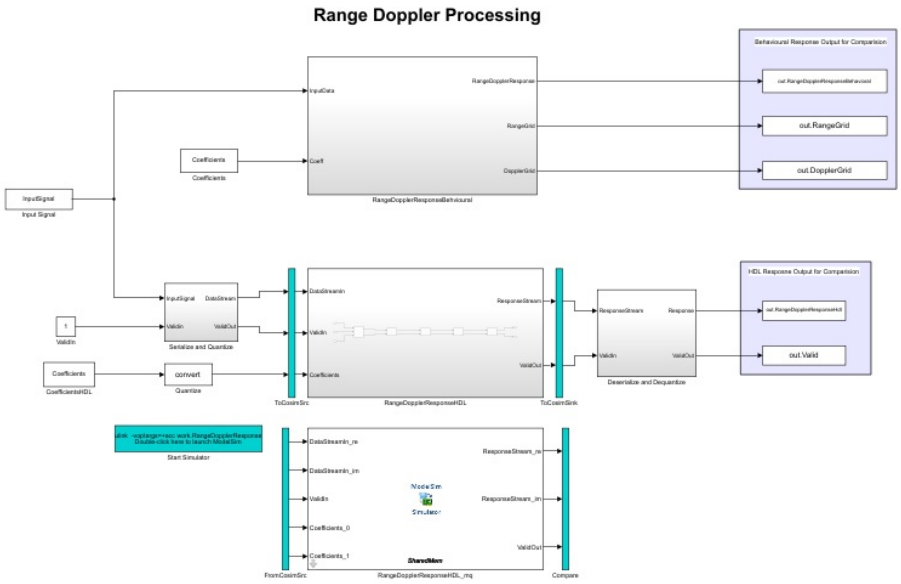
- **Target:** Xilinx Vivado synthesis tool; Virtex7 family; Device xc7vx485t; package ffg1761, speed -1; and target frequency of 300 MHz.
- **Optimization:** Clear all optimizations.
- **Global Settings:** Set the Reset type to Asynchronous.
- **Test Bench:** Select HDL test bench, Cosimulation model, and System Verilog DPI test bench.

HDL Code Verification Using Cosimulation

After the Model is set up, use the **HDL Workflow Advisor** to generate the HDL code using the HDL Coder tools, and generate a System Verilog DPI test bench to test the model using HDL Verifier. To start the HDL Workflow Advisor, right-click on the `RangeDopplerResponseHDL` subsystem, navigate to **HDL Code**, and click **HDL Workflow Advisor**. Alternatively, you can use these commands to generate HDL code and System Verilog test bench.

```
% makehdl([modelname '/RangeDopplerResponseHDL']);   % Generate HDL code
% makehdltb([modelname '/RangeDopplerResponseHDL']); % Generate Cosimulation test bench
```

After generating HDL code and the test bench, a new Simulink model named `gm_<modelname>_mq` containing a ModelSim® Cosimulation block is created in your working directory. This figure shows the generated model.



To open the test bench model use these commands.

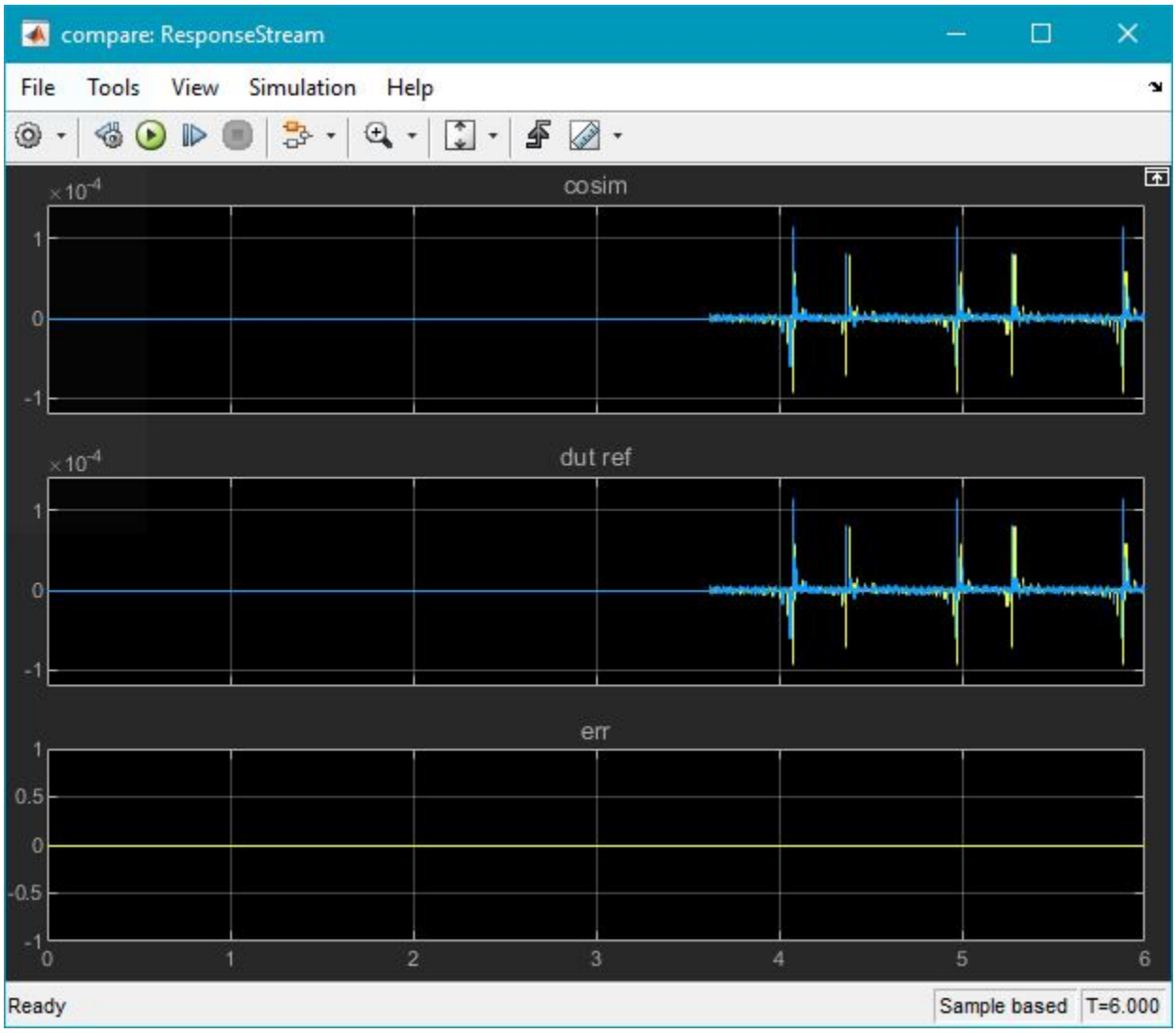
```
% modelname = ['gm_',modelname,'_mq'];
% open_system(modelname);
```

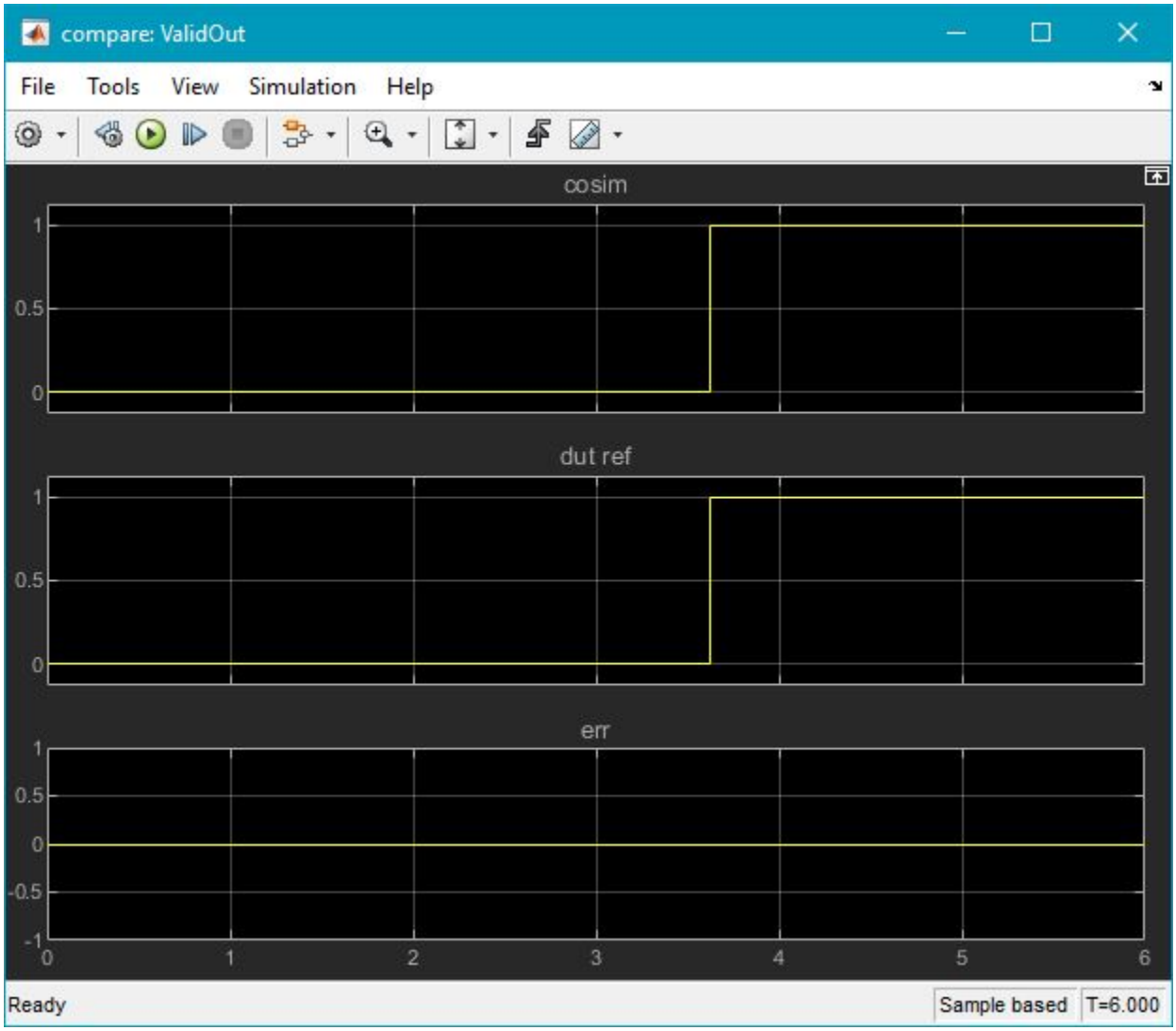
Launch ModelSim and run the cosimulation model to display the simulation results. You can click the Play button to run the test bench, or you can run it via command window using the following command.

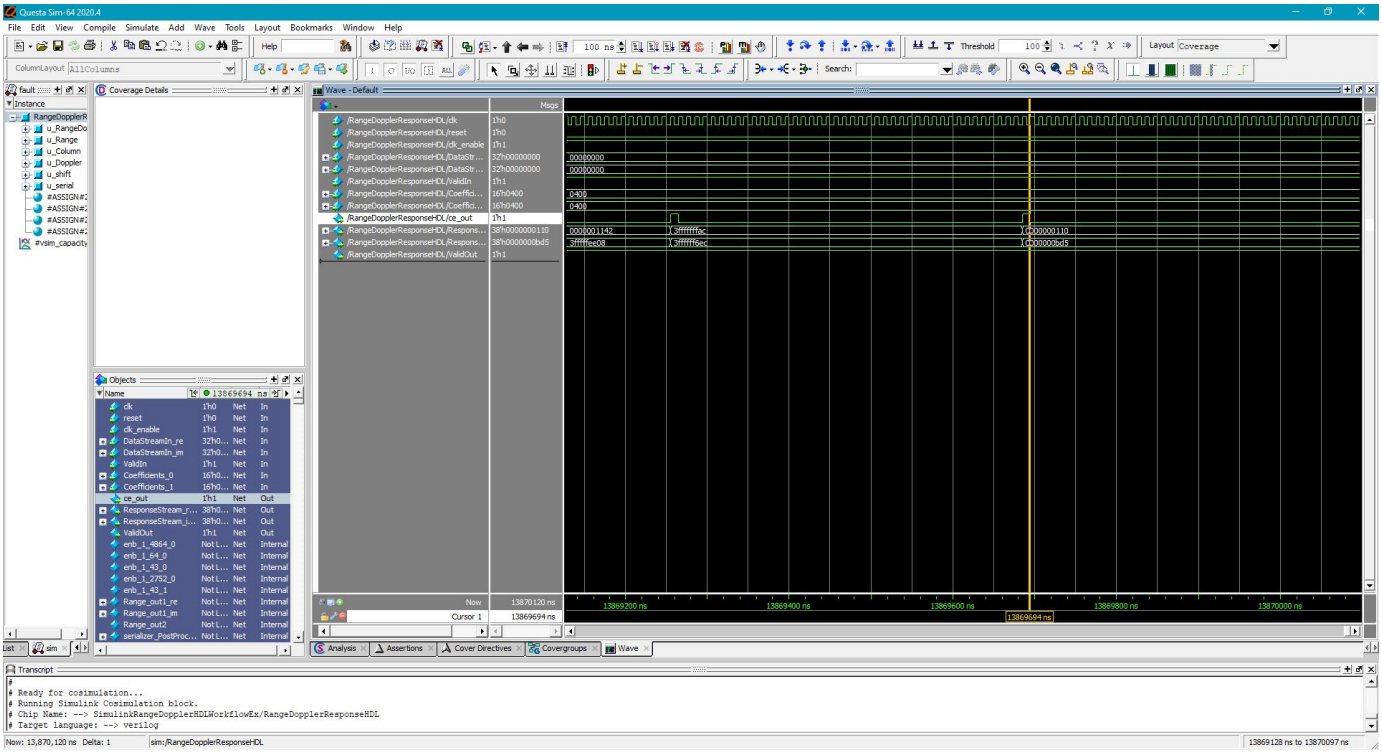
```
% sim(modelname);
```

The Simulink test bench model configures QuestaSim® with the signal from the HDL model and Time Scope in Simulink.

The test bench scopes shows the output of the complex-valued response vector from the implementation model and the cosimulation output as well as the error between the two outputs.



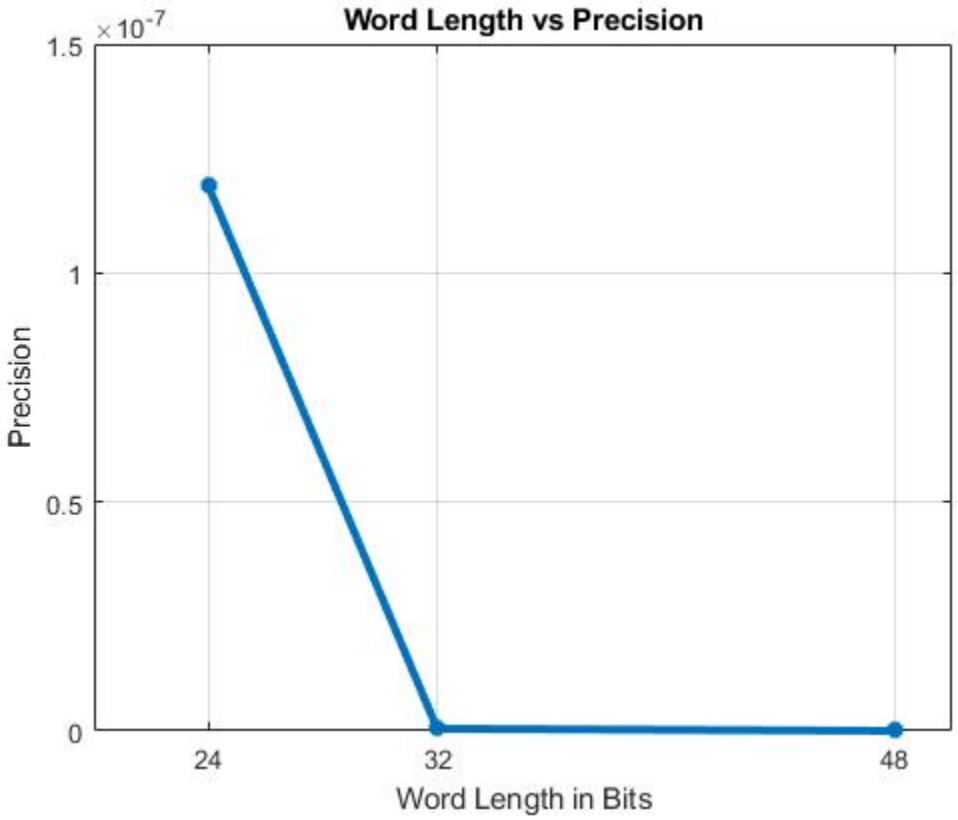

compare: ValidOut
File
Tools
View
Simulation
Help
cosim
1
0.5
0
dut ref
1
0.5
0
err
1
0.5
0
-0.5
-1
0
1
2
3
4
5
6
Ready
Sample based
T=6.000



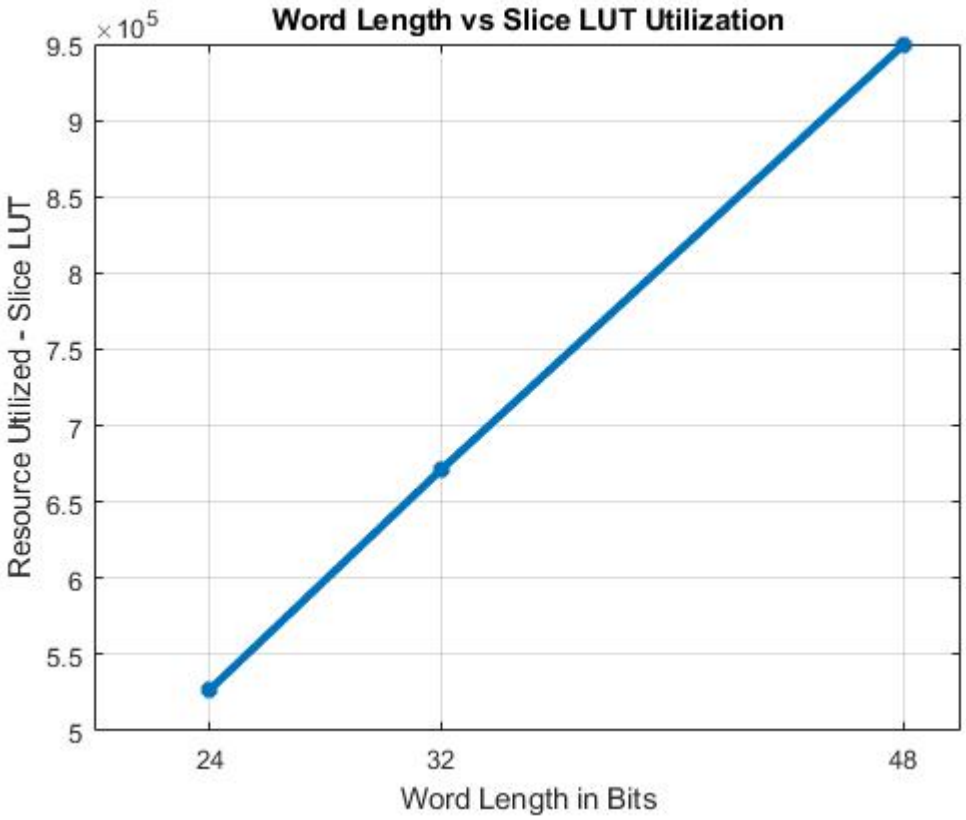
Fixed-Point Word Length (Precision) and Resource Utilization Tradeoffs

This example uses a word length of 32 bits and a fraction length of 31 bits for design, simulation, and implementation. There are tradeoffs associated with increasing the data precision with respect to resource utilization.

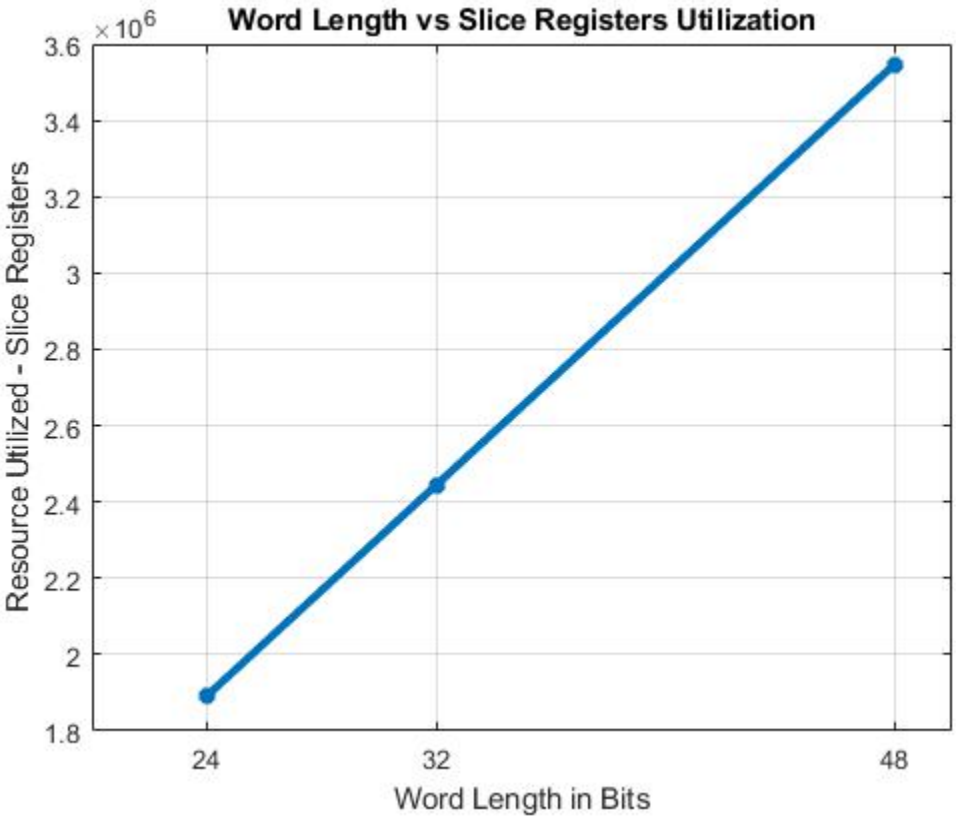
This figure shows the precision with respect to the word length.



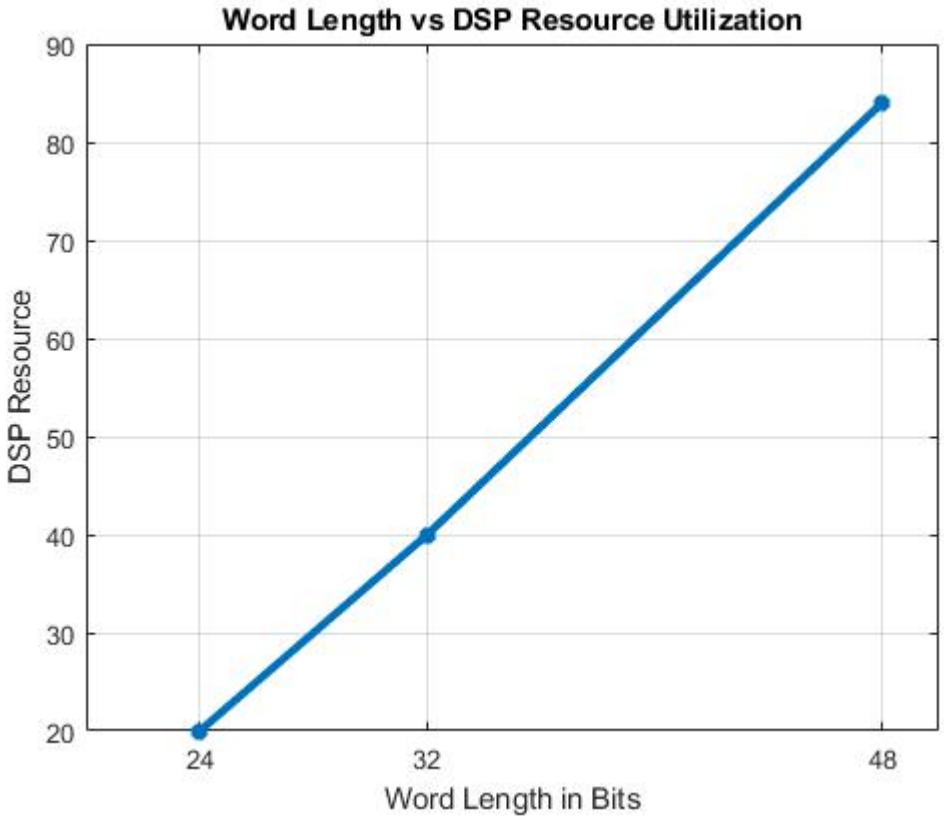
The next figure shows the slice LUT utilization with respect to the word length.



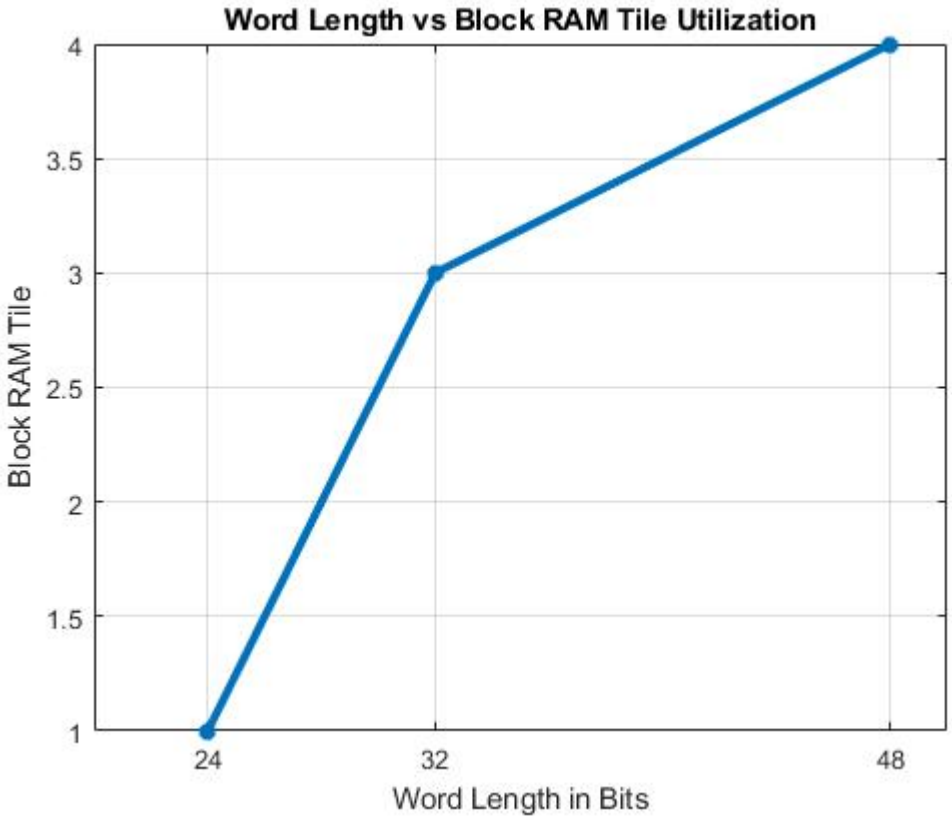
The next figure shows the slice registers utilization with respect to the word length.



The next figure shows the DSP block utilization with respect to the word length.



The next figure shows the block RAM tile utilization with respect to the word length.



Summary

This example demonstrated a workflow for designing a Simulink model for a hardware-compatible range-Doppler response block, and verifying the results with an equivalent behavioral setup from the Phased Array System Toolbox. The example also shows how to generate HDL code for a fixed-point implementation and verifying the generated code in Simulink for functional correctness. This example showed how to set up and launch ModelSim to co-simulate the HDL code and compare its output to the output generated by the HDL implementation model. The cosimulation used ModelSim for the HDL code simulation and compared results to the output generated by the implementation model.

FPGA-Based Cell-Averaging Constant False Alarm Rate (CA-CFAR) Detector

This example shows how to design a CA-CFAR detector suitable for hardware.

To verify the implementation model is functionally correct, we compare the simulation output of the implementation model with the output of a CFAR based behavioral model using Phased Array System Toolbox™. The term deployment here implies designing a model that is suitable for implementation on a FPGA. The model is implementation ready and this will be verified in the example. The HDL workflow is designed in fixed-point.

The Phased Array System Toolbox provides the floating point behavioral model for the CFAR detector through the phased.CFARDetector System object™. This behavioral model is used to verify the results of the implementation model and the automatically generated HDL code as well.

Fixed-Point Designer™ provides data types and tools for developing fixed-point and single precision algorithms to optimize performance on an embedded hardware. Bit true simulations can be performed to observe the impact of limited range and precision without implementing the design in hardware.

This example uses HDL Coder™ to generate HDL Code from the developed Simulink® model and verifies the HDL Code using the HDL Verifier™. HDL Verifier is used to generate a co-simulation test bench model to verify the behavior of the automatically generated HDL Code. The test bench uses ModelSim® for Co-simulation to verify the generated HDL code.

Algorithm Design

In a radar system, target detection is achieved by comparing the received signal power to a global threshold. If the received power is greater than the threshold, it marks the presence of a target else a target is said to be absent. This makes the choice of threshold a critical characteristic. The appropriate threshold value depends on maximizing the detection and minimizing the false alarm.

The threshold is chosen based on apriori knowledge (estimate) of the interferer power. The interferer power is affected by many external factors, hence, the variance will be a large value when measured globally. When the threshold is constant, the increase in interferer power can lead to an increase in false detections and at the same time, if the interferer power drops significantly, the target might not be detected.

The CFAR detector, as the name suggests, maintains the specified false alarm rate by means of an ***Adaptive Thresholding***, wherein the threshold is calculated based on the locality of the Cell Under Test (CUT) and this defines the cell for which detection is required. The interference power of the neighboring cells is used to calculate the threshold for a CUT. The detection threshold is calculated as

$$T = -P_n^2.ln(P_{FA}) = -P_n^2\alpha$$

where, P_{FA} is the Probability of False Alarm, α is the Threshold Factor, P_n is the Interference power level.

Cell-Averaging(CA) CFAR Detector In a CA CFAR, the lead and lag cells are used to calculate the interferer estimate. The number of lead cells are the same as that of the number of lag cells. CA-CFAR assumes that, the neighboring cells to the CUT contains the same interference statistic -

Homogenous Interference and the target is present in only one CUT. To reinforce the second assumption, guard cells are placed immediately after the CUT.

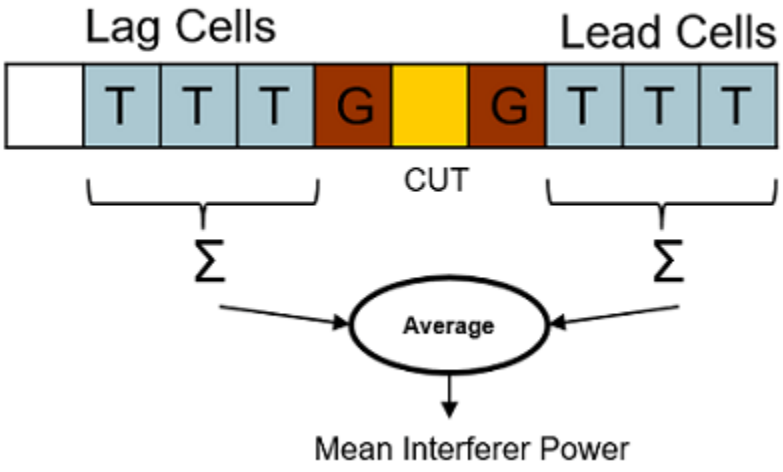
For a CA-CFAR with an independent and identically distributed (i.i.d) Gaussian interference (standard normal), the average noise power is just the mean of output of square law detector of all the training cells, which is

$$\widehat{P_n^2} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Here x_i is the signal from the i-th training cell. For a given Probability of False Alarm, the threshold factor can be calculated as,

$$\alpha = N.(P_{FA}^{(\frac{-1}{N})} - 1)$$

The training cell and guard cell along with the CUT is called as the CFAR Window. The following figure shows a representation of a CFAR Window.



Implementation Model

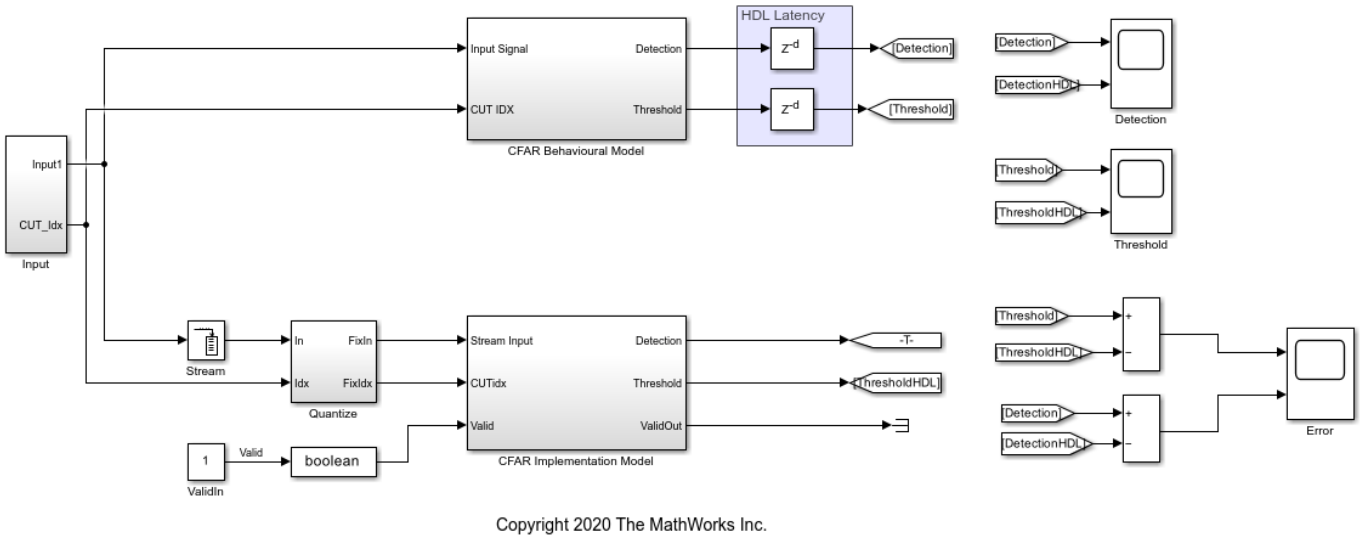
The implementation model is designed using the HDL Coder compatible blocks from the Simulink HDL Coder library. For this example, we have chosen the following values for the parameters:

- No. of Training Cells = 50
- No. of Guard Cells = 2
- Probability of False Alarm = 0.005
- Total No. of Cells = 1000

The following command is used to open the Simulink model.

```
modelname = 'SimulinkCFARHDLWorkflowExample';
open_system(modelname);
%
% Ensure model is visible and not obstructed by scopes
```

```
scopes = find_system(modelname,'BlockType','Scope');
close_system(scopes);
```



The Simulink model consists of two branches from the input block. The top branch is the behavioral model with floating point operations of the phased.CFARdetector System object. The bottom branch is the functional equivalent implementation model with fixed-point version.

The input to the behavioral model is a (NCell x 1) 1000x1 matrix. The input is passed through the Square-Law sub-system which performs the square-law operation which is then forwarded into the CFAR Detector behavioral model.

The input to the implementation model is provided via buffer which converts a multi-dimensional signal into single dimensional data stream for a deployable model. The input data type is then converted to fixed-point using the Quantize block. The fixed-point has a word-length of 24 bits and a fraction length of 12 bits. The tradeoff between different fixed-point setting with resource utilization and accuracy is discussed later in this example. The input is then passed on to the CFAR implementation model sub-system which performs the CFAR Detection.

The output of the CFAR Detector behavioral model is passed through a delay of 125 cycles to compensate the delay for the output of the implementation model.

The scope block plots the threshold and detection outputs of the behavioral model and implementation model. In addition, the error between the threshold of implementation model and behavioral model, and the error between the detection of implementation model and behavioral model is also calculated and plotted.

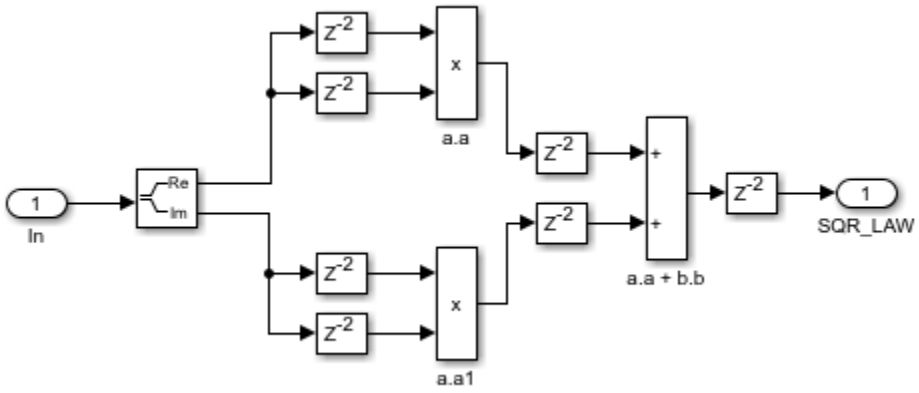
The implementation model contains the following sub-systems:

1. Square-Law HDL
2. Alpha HDL
3. CFAR Core HDL
4. Validate

Square-Law HDL

The following command is used to open the Square-Law HDL subsystem model

```
open_system([modelname '/CFAR Implementation Model/Square Law HDL']);
```



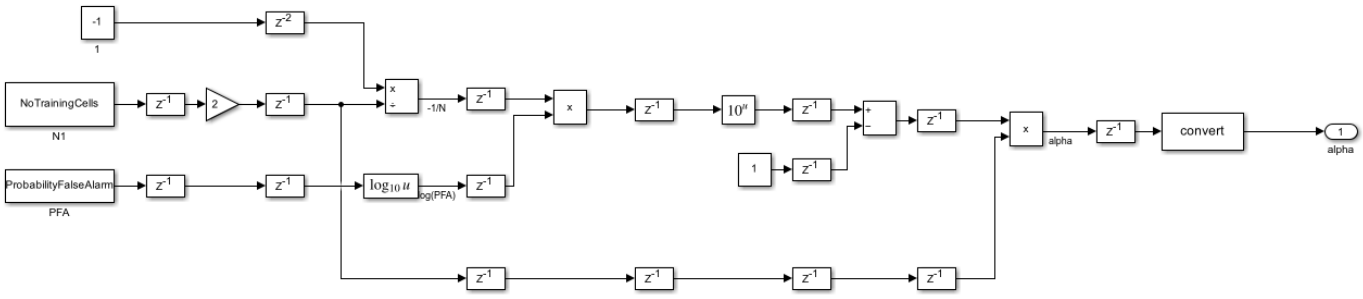
The model computes the square-law envelope of the complex input signal.

For the implementation model the square-law is designed as a deployable model, with additional pipelining registers. This model is implemented using adders and multipliers which account for a latency of 6 cycles.

Alpha HDL

The following command is used to open the Alpha HDL subsystem model

```
open_system([modelname '/CFAR Implementation Model/Alpha HDL']);
```

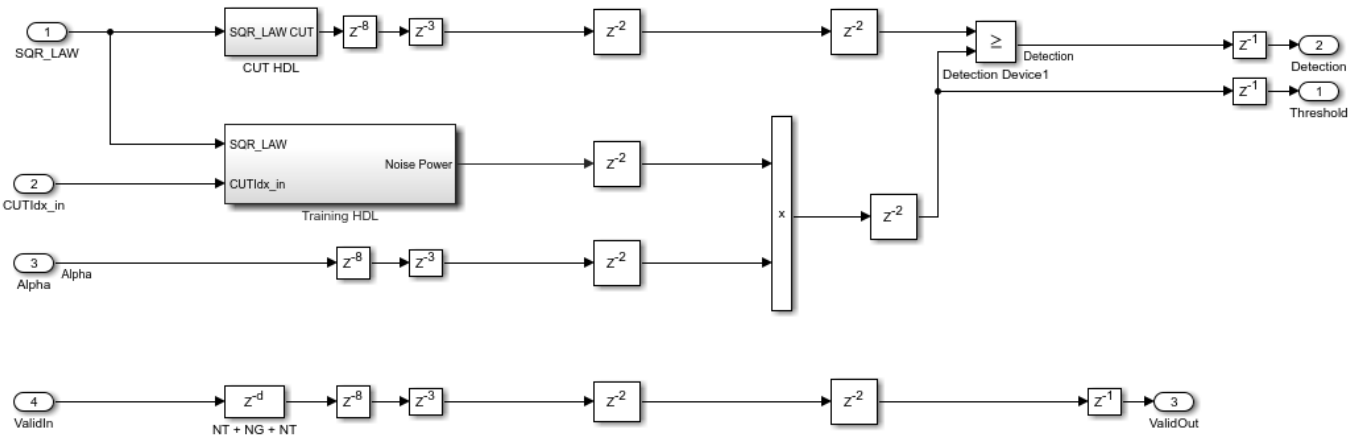


The Alpha HDL block utilizes the **No. of Training Cells** and **Probability of False Alarm** value to calculate the threshold factor (α).

This subsystem uses Math HDL blocks and works in single precision for the native floating-point division operation which is then converted to fixed-point at the output. This block accounts for a pipeline latency of 7 Cycles.

CFAR Core HDL

```
%The following command is used to open the CFAR Core subsystem model
open_system([modelname '/CFAR Implementation Model/CFAR Core']);
```



This subsystem extracts the training cells from the input stream and calculates the noise power. The noise power is then multiplied by alpha to generate the threshold which is later used in the detection process. There are two outputs from this block, namely, threshold and detection.

The threshold is the direct output of the product block whereas the detection output is provided through a comparator block which compares the threshold and signal value of CUT and returns true if the CUT signal is greater than the threshold.

This subsystem accounts for an input streaming delay of 102 (2*NumberTrainingCells + NoGuardCells) clock cycles with an additional pipelining latency of another 23 cycles. The total HDL latency is 125 cycles.

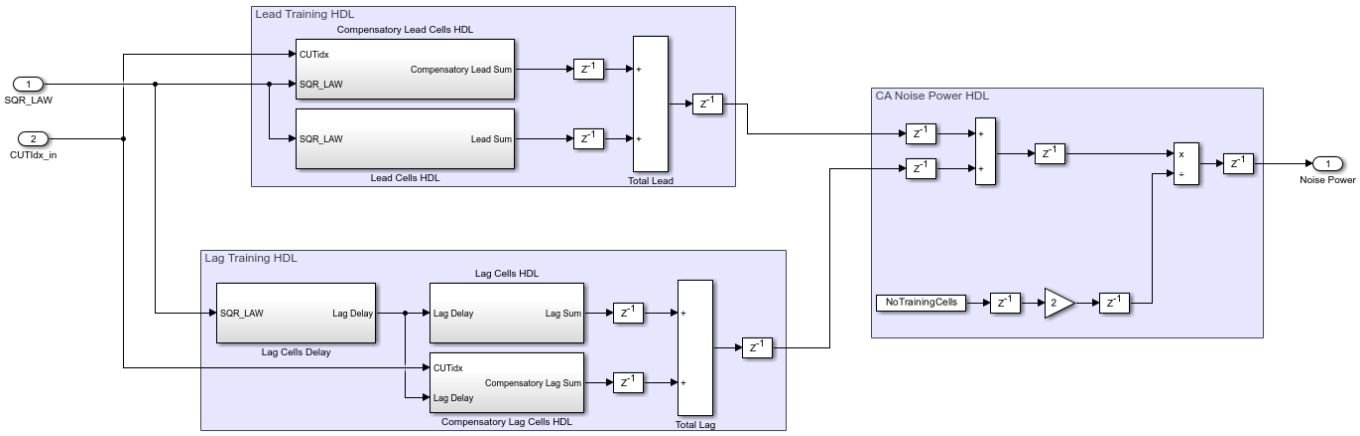
This block contains the following subsystems:

1. Training HDL
2. CUT HDL

Training HDL

The following command is used to open the Training HDL subsystem model.

```
open_system([modelname '/CFAR Implementation Model/CFAR Core/Training HDL']);
```



The lead training HDL subsystem extracts the lead cells of the CUT and performs a running sum. At the same time the lag training HDL subsystem pulls out the lag cells of the CUT and performs a running sum with a latency of 8 cycles each. The implementation of the lead training HDL and lag training HDL is very much analogous to a moving average filter the difference is that instead of the average we use the sum of window elements.

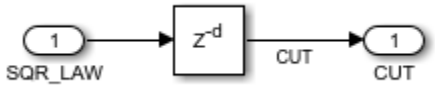
The CA noise power HDL subsystem sums up the lead power value and the lag power value and estimates the average noise power by dividing the sum by 100 (2*NoTrainingCells). This blocks accounts for a delay of 3 clock cycles.

The output of the training HDL subsystem is the noise power which is used to calculate the threshold.

CUT HDL

The following command is used to open the CUT HDL subsystem model

```
open_system([modelname '/CFAR Implementation Model/CFAR Core/CUT HDL']);
```

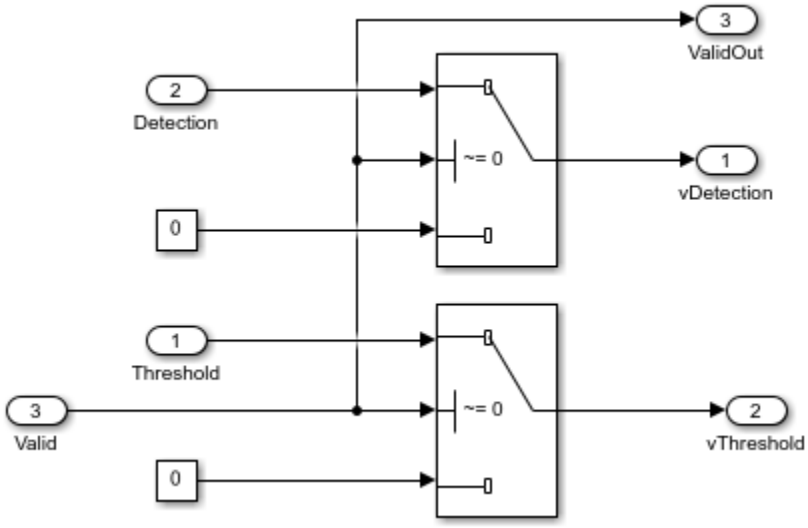


This subsystem uses a single delay block with a delay of 102 (2*NumberTrainingCells + NoGuardCells) cycles to time-align the CUT with the generated threshold value from the training HDL block. The above delay is the minimum delay required before which the CFAR Detector can detect the target at the first cell.

Validate

The following command is used to open the validate subsystem model

```
open_system([modelname '/CFAR Implementation Model/Validate']);
```

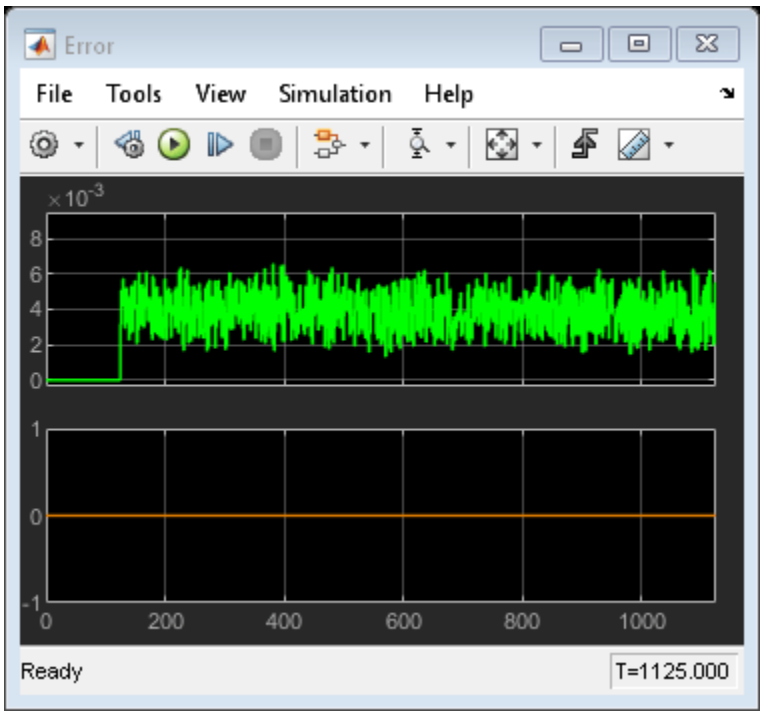


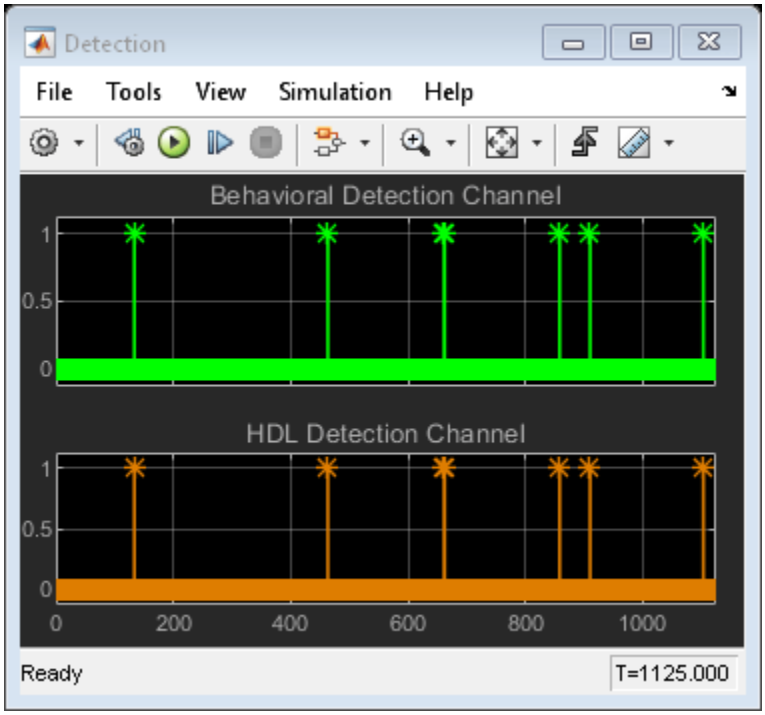
The valid input along with the latency is used to check the validity of the output. When the output is not valid this subsystem sends zero to the output.

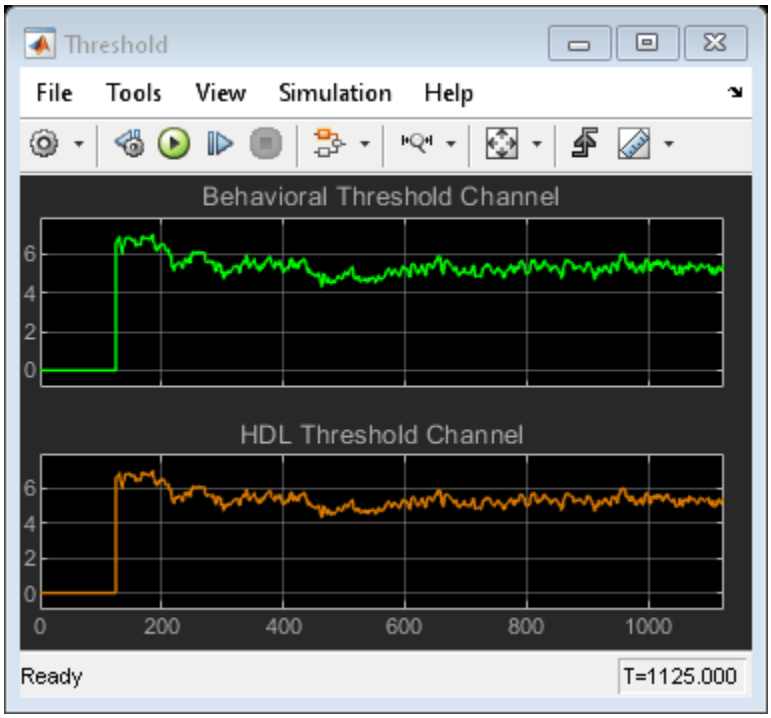
Comparing the Results of Implementation Model to the Behavioral Model

The model can be simulated by clicking the Play button or using the sim command as shown below,

```
sim(modelname);
```







The Scope blocks are used to compare the output of implementation and behavioral model. The scope displays the detection and threshold from both the behavioral and implementation model and an additional scope displays the calculated the error.

The implementation model has a data streaming latency of 102 cycles and pipelining latency of 23 cycles. This in total accounts for an overall latency of 125 cycles. To time align the behavioral model with the implementation model we use an additional delay of 125 to the output of behavioral model.

With the 24 bit fixed-point of fraction length 12 bits, we have an error bounded by approximately 0.006 between the behavioral model and the implementation model threshold. Since the detection is Boolean we have no significant error in the detection output.

Code Generation and Verification

This section covers the procedure to perform HDL codegeneration for the implementation model. It also covers the verification that the generated code is functionally correct. The behavioral model provides the reference values to validate the output from HDL model.

If you start with a new model, you can run hdlsetup (HDL Coder) to configure the Simulink model for HDL code generation. To configure the Simulink model for test bench creation, *open* Simulink's **Model Settings** , *select* **Test Bench** under HDL Code Generation in the left panel, and *check* **HDL test bench** and **Co-simulation model** in the Test Bench Generation Output properties group.

Model Settings

After the fixed-point implementation is verified and the implementation model produces the same results as your floating-point, behavioral model, you can generate HDL code and test bench. For code generation and test bench, set the HDL Code Generation parameters in the **Configuration Parameters** dialog. The following parameters in Model Settings are set under HDL Code Generation:

- **Target:** Xilinx Vivado synthesis tool; Virtex7 family; Device xc7vx485t; package ffg1761, speed -1; and target frequency of 300 MHz.
- **Optimization:** Uncheck all optimizations
- **Global Settings:** Set the Reset type to Asynchronous
- **Test Bench:** Select HDL test bench, Co-simulation model and System Verilog DPI test bench

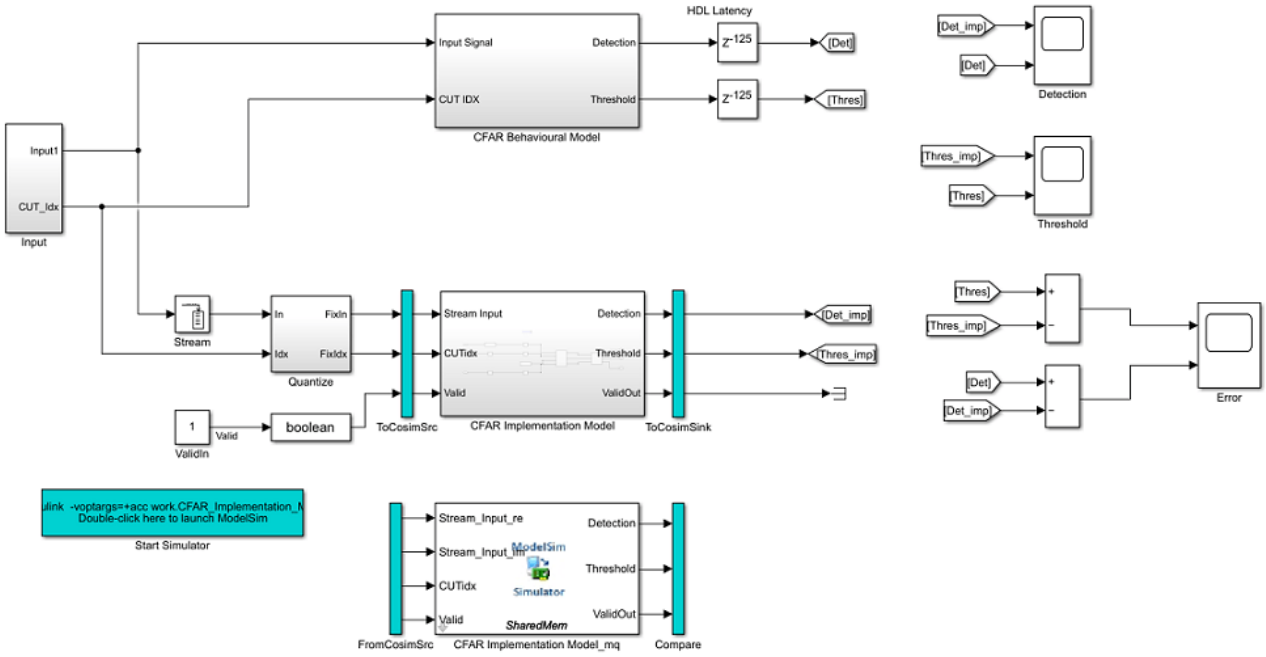
HDL Code Verification via Co-Simulation

After the model is set up, **HDL Workflow advisor** can be invoked to generate the HDL code using the HDL Coder also use the HDL Verifier to generate a System Verilog DPI Test Bench to test the model. To invoke HDL Workflow advisor *right-click* on the **CFAR Implementation model** subsystem and *navigate* to **HDL Code** and *left-click* **HDL Workflow advisor**. Instead of using HDL Workflow advisor the following lines of code can also be used to generate HDL code and System Verilog Test Bench.

```
% Uncomment the following two lines to generate HDL code and test bench.
% makehdl([modelname '/CFAR Implementation Model']);   % Generate HDL code
% makehdltb([modelname '/CFAR Implementation Model ']); % Generate Cosimulation test bench
```

Since all the optimizations are unchecked we do not have to add extra delays to the behavioral output other than the HDL latency previously added. (This is because all the critical paths are manually pipelined in the implementation model).

After generating HDL code and test bench a new Simulink model named gm_<modelname>_mq containing a ModelSim Simulator block is created in your working directory, which looks like this:

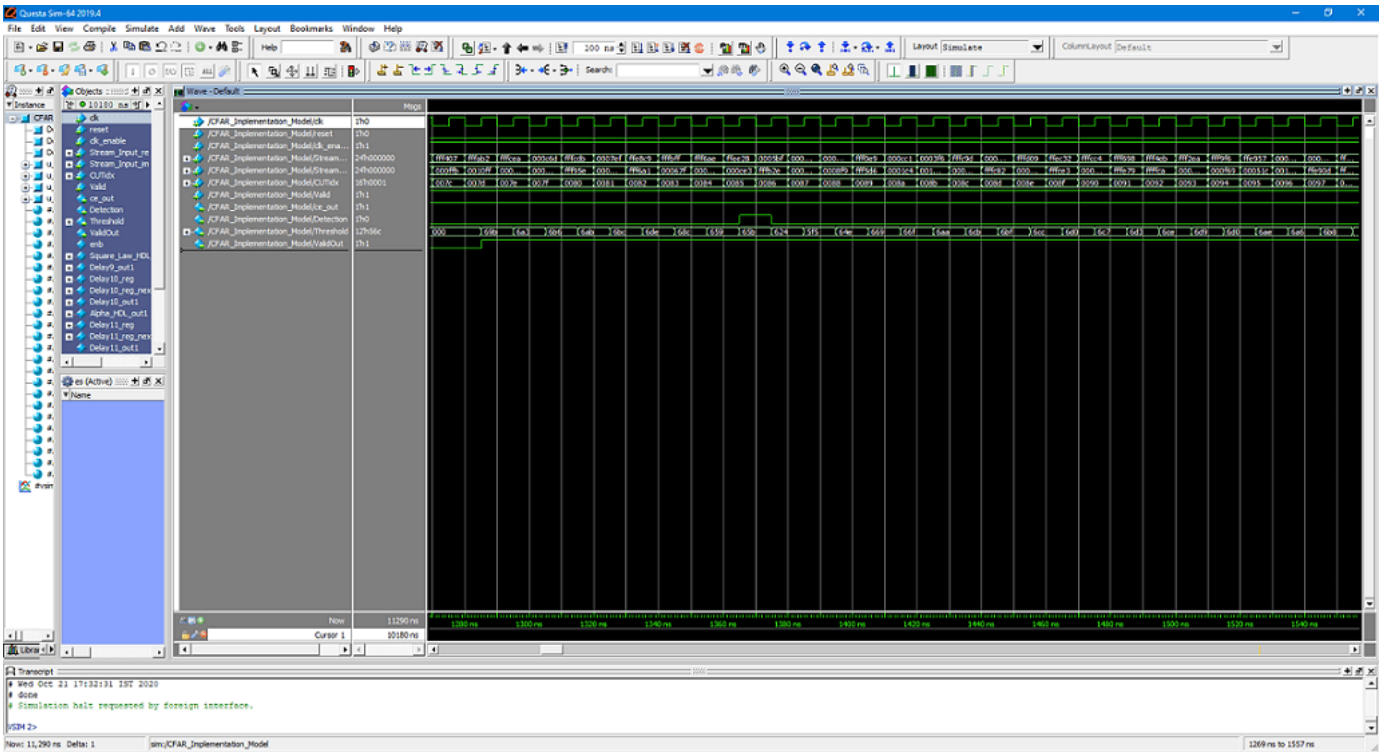



```
% To open the test bench model, uncomment the following lines of code
% modelname = ['gm_',modelname,'_mq'];
% open_system(modelname);
```

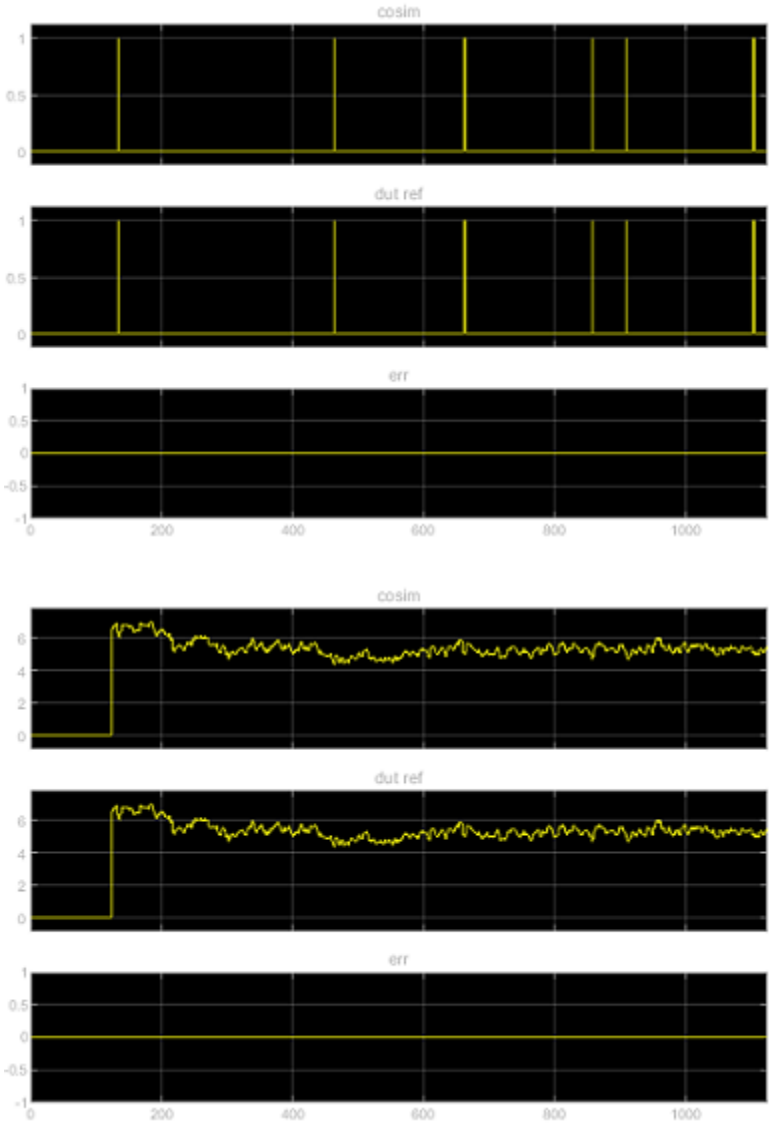
Launch ModelSim and run the co-simulation model to display the simulation results. You can click on the Play button on the top of Simulink canvas to run the test bench or you can do it via command window from the code below.

```
% Uncomment the following line, to run the test bench.
% sim(modelname);
```

The Simulink test bench model will populate the QuestaSim® with the HDL model's signal and Time Scopes in Simulink.



The Simulink scope shows detection output and threshold output for both the co-simulation and Design Under Test (DUT) as well as the error between them. The scopes comparing the results of the co-simulation can be found in test bench model inside the Compare subsystem, which is at the output of the CFAR_HDL_mq subsystem.

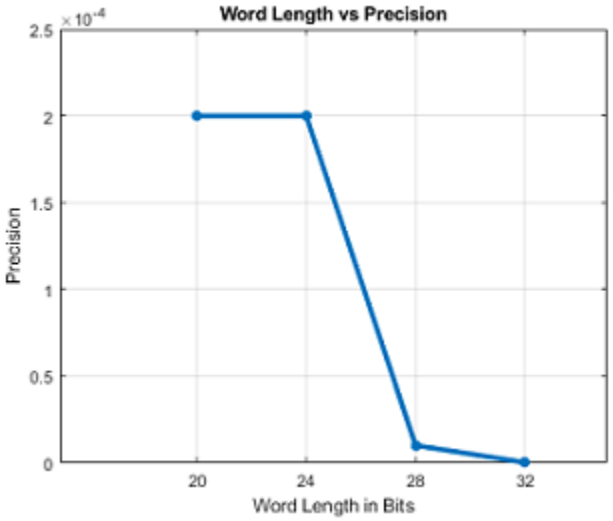


Fixed-Point Word Length and Fraction Length Tradeoffs

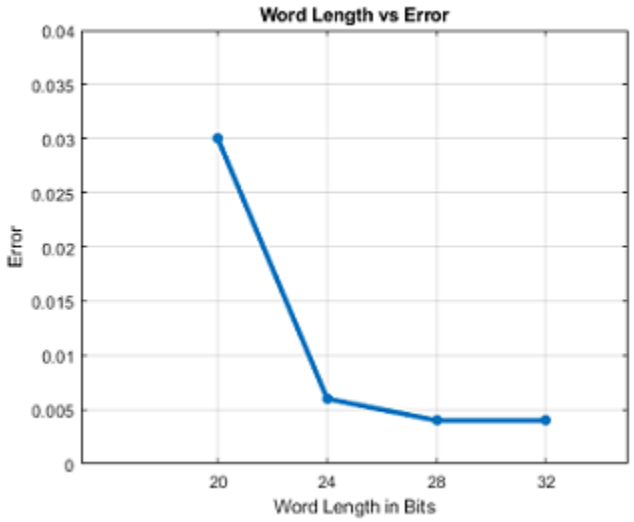
For this example a Fixed-Point word length of 24 bit and a fraction length of 12 bit is used for simulation, and implementation. The following figures show the trade-off with choosing a longer fraction length which would increase the precision (reduces the Error) but also increases the resource utilization.

The following plot shows fraction length associated with chosen word length.

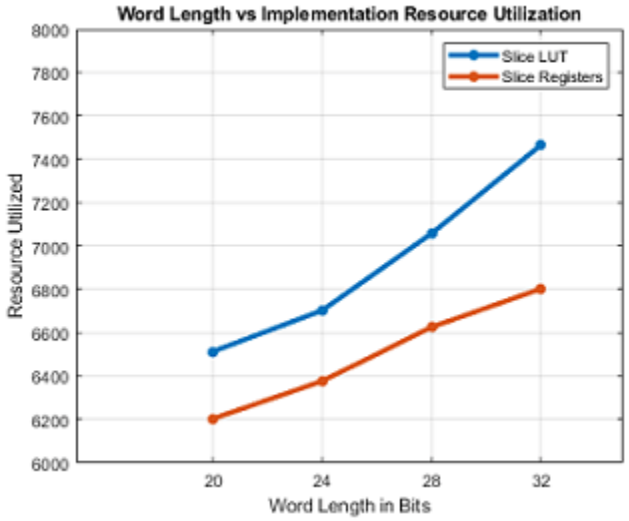
The following plot shows the Precision with respect to chosen word length.

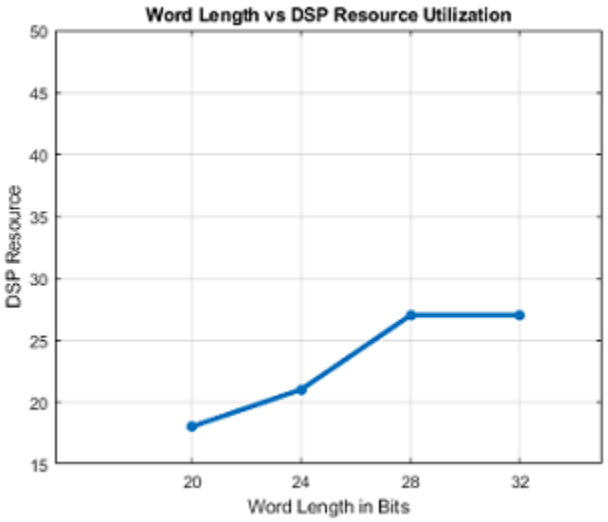


The following plot shows the Error with respect to chosen word length (Precision)



The following plots show the LUT/Registers/DSP Utilization with respect to chosen Word Length.





Summary

This example demonstrated how to design a Simulink model for a Cell Averaging Constant False Alarm Rate(CA CFAR) Detector, verify the results with an equivalent behavioral setup from the Phased Array System Toolbox. This example demonstrates how to automatically generate HDL code for a fixed-point equivalent algorithm and verify the generated code in Simulink. The generated HDL code as well as a co-simulation test bench for the Simulink subsystem was created with blocks that support HDL code generation. This example showed how to setup and launch ModelSim to cosimulate the HDL code and compare its output to the output generated by the HDL implementation model. The cosimulation is performed via ModelSim for the HDL code and compare results to the output generated by the HDL model.

FPGA-Based Minimum-Variance Distortionless-Response (MVDR) Beamformer

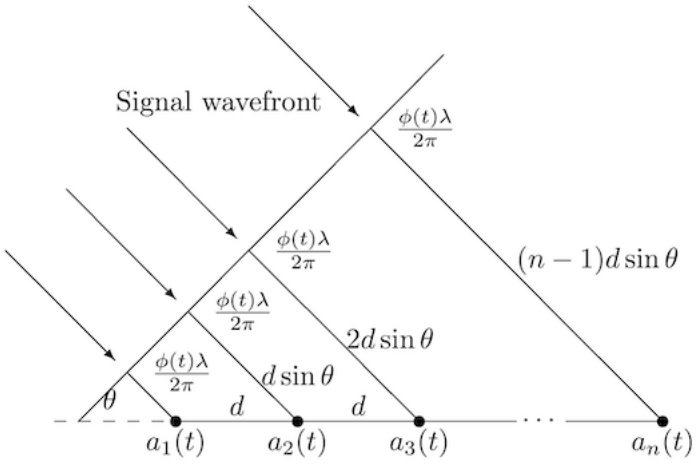
This example shows how to implement a minimum-variance distortionless-response (MVDR) beamformer suitable for hardware.

For more information on beamformers, see “Conventional and Adaptive Beamformers” (Phased Array System Toolbox).

MVDR Objective

The MVDR beamformer preserves the gain in the direction of arrival of a desired signal and attenuates interference from other directions [1], [2].

Given readings from a sensor array, such as the uniform linear array (ULA) in the following diagram, form data matrix A from samples of the array, where $a(t)$ is an n-by-1 column vector of readings from the array sampled at time t, and $a(t)^H$ is one row of matrix A. Many more samples are taken than there are elements in the array. This results in the number of rows in A being much greater than the number of columns. An estimate of the covariance matrix is $A^H A$, where A^H is the Hermitian or complex-conjugate transpose of A.



Compute the MVDR beamformer response by solving the following equation for x, where b is a steering vector pointing in the direction of the desired signal.

$$(A^H A)x = b$$

The MVDR weight vector w is computed from x and b using the following equation, which normalizes x to preserve the gain in the direction of arrival of the desired signal.

$$w = \frac{x}{b^H x}$$

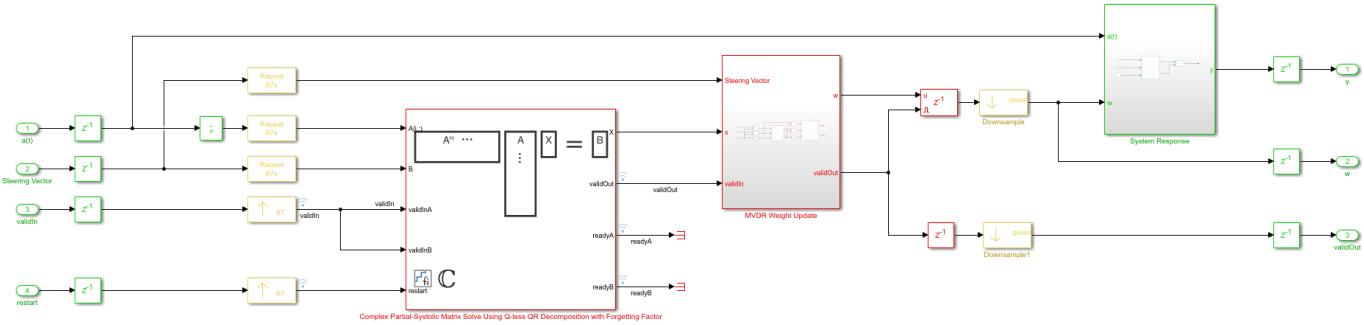
The MVDR system response is the inner product between the MVDR weight vector w and a current sample from the sensor array $a(t)$.

$$y = w^H a(t)$$

HDL-Optimized MVDR

The three equations in the previous section are implemented by the three primary blocks in the following model. The rate changes give the matrix solve additional clock cycles to update before the next input sample. The number of clock cycles between a valid input and when the complex matrix solve block is ready is twice its input wordlength to allow time for CORDIC iterations, plus 15 cycles for internal delays.

```
load_system('MVDRBeamformerHDLOptimizedModel');
open_system('MVDRBeamformerHDLOptimizedModel/MVDR - HDL Optimized')
```



Instead of forming data matrix A and computing the Cholesky factorization of covariance matrix $A^H A$, the upper-triangular matrix of the QR decomposition of A is computed directly and updated as each data vector $a(t)$ streams in from the sensor array. Because the data is updated indefinitely, a forgetting factor is applied after each factorization. To integrate with an equivalent of a matrix of m rows, the forgetting factor α should be set to

$$\alpha = \exp(-1/(2m)).$$

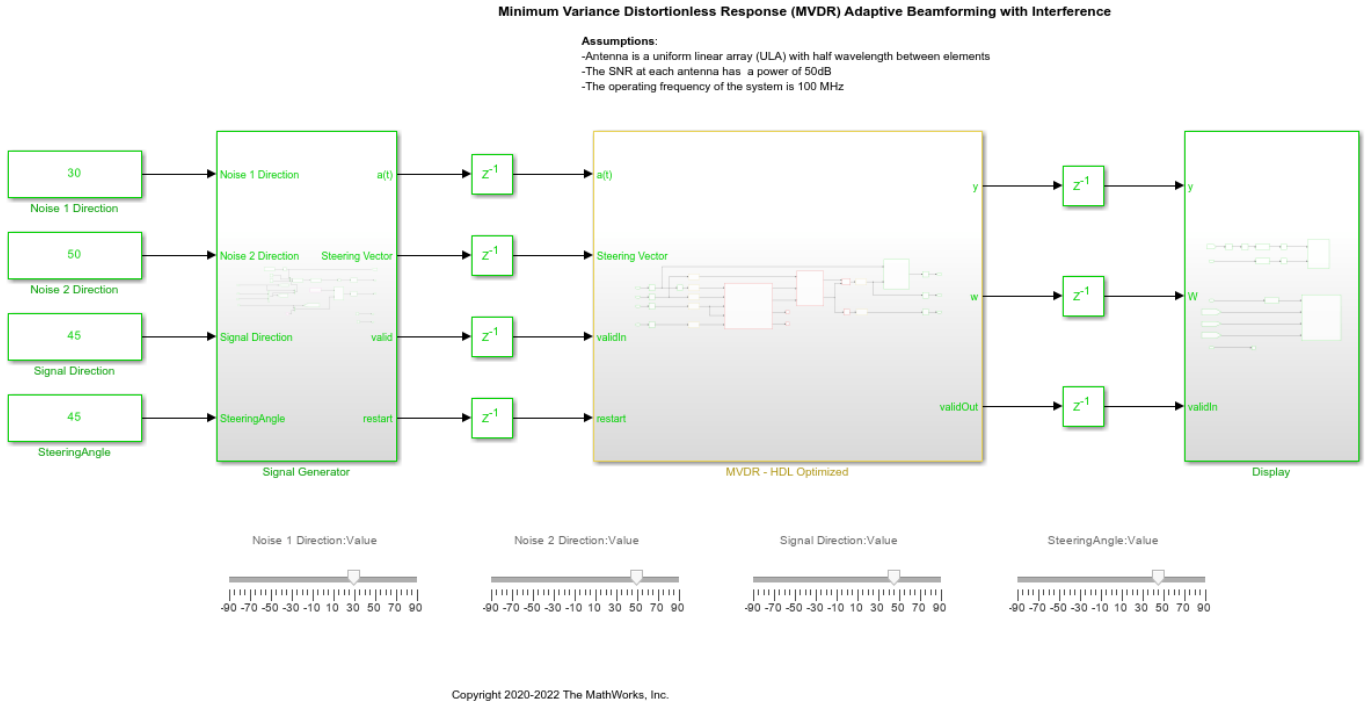
This example simulates the equivalent of a matrix with $m = 300$ rows, so the forgetting factor is set to 0.9983.

The **Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor** block is implemented using the method found in [3]. The upper-triangular matrix R from the QR decomposition of A is identical to the Cholesky factorization of $A^H A$ except the signs of values on the diagonal. Solving the matrix equation $(A^H A)x = b$ by computing the Cholesky factorization of $A^H A$ is not as efficient or as numerically sound as computing the QR decomposition of A directly [4].

Run Model

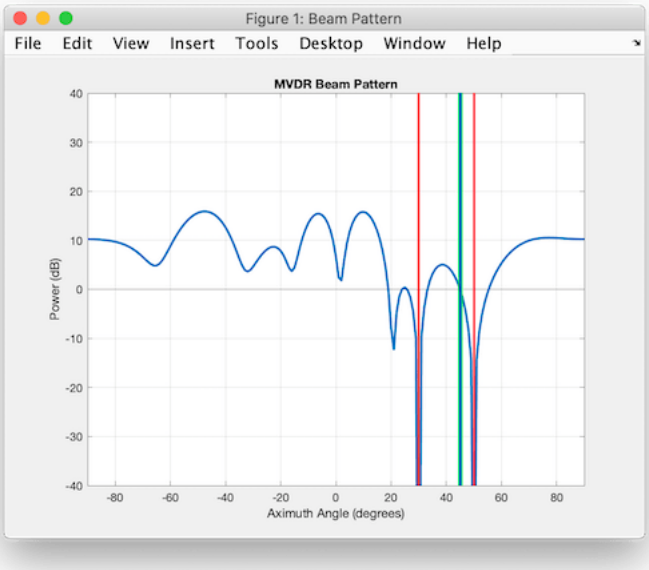
Open and simulate the model.

```
open_system('MVDRBeamformerHDLOptimizedModel')
scopes = find_system('MVDRBeamformerHDLOptimizedModel','BlockType','Scope');
close_system(scopes);
```

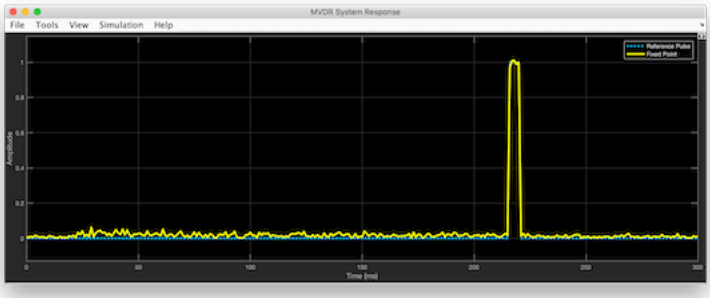


As the model is simulating, you can adjust the signal direction, steering angle and noise directions by dragging the sliders, or by editing the constant values.

When the signal direction and steering angle are aligned as indicated by the blue and green lines, you can see that the beam pattern has a gain of 0 dB. The noise sources are nulled as indicated by the red lines.



The desired pulse appears when the noise sources are nulled. This example simulates with the same latency as the hardware, so you can see the signal settle over time as the simulation starts and when the directions are changed.



Set Parameters

The parameters for the beamformer are set in the model workspace. You can modify the parameters by editing and running the `setMVDRExampleModelWorkspace` function.

References

[1] V. Behar et al. "Parameter Optimization of the adaptive MVDR QR-based beamformer for jamming and multipath supression in GPS/GLONASS receivers". In: Proc. 16th Saint Petersburg International Conference on Integrated Navigation systems. Saint Petersburg, Russia, May 2009, pp. 325--334.

[2] Jack Capon. "High-resolution frequency-wavenumber spectrum analysis". In: vol. 57. 1969, pp. 1408--1418.

[3] C.M. Rader. "VLSI Systolic Arrays for Adaptive Nulling". In: IEEE Signal Processing Magazine (July 1996), pp. 29--49.

[4] Charles F. Van Loan. Introduction to Scientific Computing: A Matrix-Vector Approach Using Matlab. Second edition. Prentice-Hall, 2000. isbn: 0-13-949157-0.