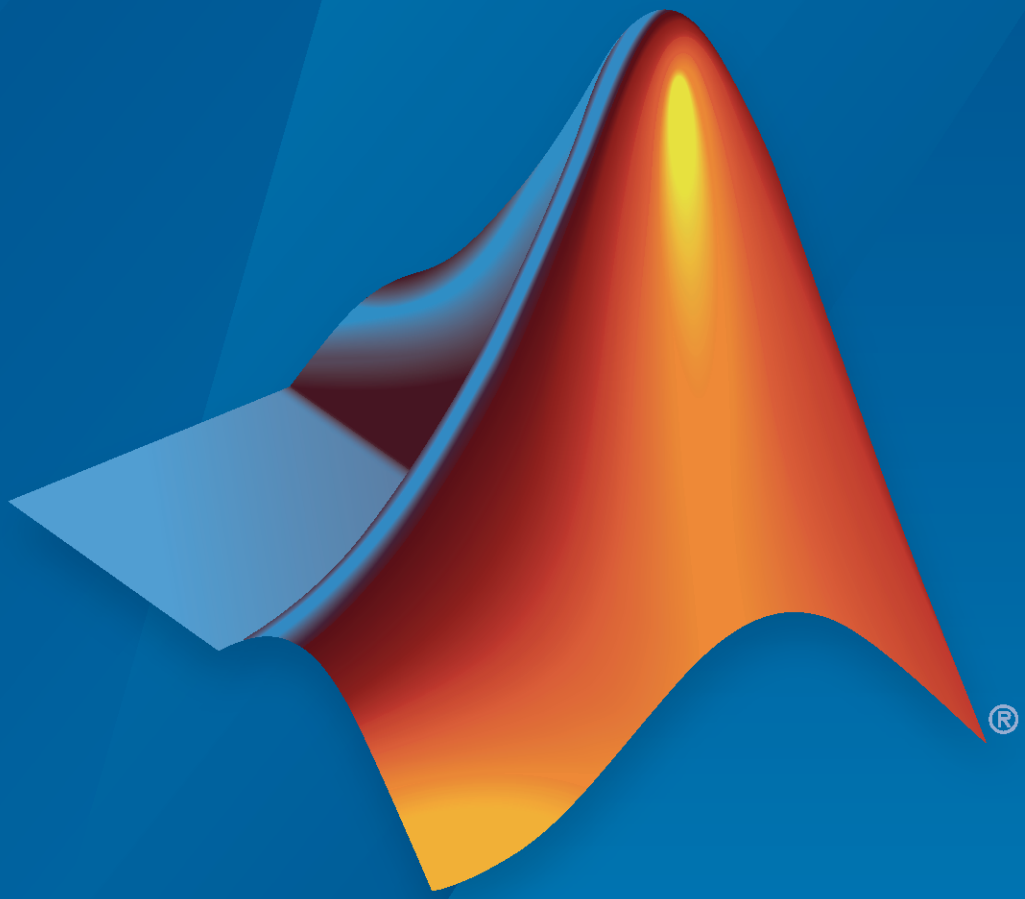
DSP HDL Toolbox™

Reference



MATLAB® & SIMULINK®

R2022a

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™ Reference

Revision History

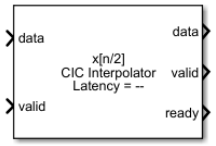
March 2022	Online only	New for Version 1.0 (Release 2022a)

Contents

Blocks

CIC Interpolator

Interpolate signal using CIC filter
Library: DSP HDL Toolbox / Filtering



Description

The CIC Interpolator block interpolates an input signal by using a cascaded integrator-comb (CIC) interpolation filter. CIC interpolation filters are a class of linear phase finite impulse response (FIR) filters consisting of a comb part and an integrator part. The CIC interpolation filter structure consists of *N* sections of cascaded comb filters, a rate change factor of *R*, and *N* sections of cascaded integrators. For more information about CIC interpolation filters, see “Algorithms” on page 1-7.

The block supports these combinations of input and output data.

- Scalar input and scalar output — Support for fixed and variable interpolation rates
- Scalar input and vector output — Support for fixed interpolation rates only
- Vector input and vector output — Support for fixed interpolation rates only

The block provides an architecture suitable for HDL code generation and hardware deployment.

Ports

Input

`data` — Input data
scalar | column vector

Input data, specified as a scalar or a column vector with a length from 1 to 64.

The input data must be a signed integer or signed fixed point with a word length less than or equal to 32.

Data Types: `int8` | `int16` | `int32` | `signed fixed point`
Complex Number Support: Yes

`valid` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

`R` — Variable interpolation rate
scalar

Use this port to dynamically specify the variable interpolation rate during run time.

This value must have the data type `fixdt(0,12,0)` and must be an integer in the range from 1 to the **Interpolation factor (Rmax)** parameter value.

Dependencies

To enable this port, on the **Main** tab, set the **Interpolation factor source** parameter to `Input port`.

Data Types: `fixdt(0,12,0)`

`reset` — Clears internal states

scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — CIC-interpolated output data

scalar | column vector

CIC-interpolated output data, returned as a scalar or a column vector with a length from 1 to 64. You can define the data type of this output by setting the **Output data type** parameter on the **Data Types** tab.

Data Types: `int8` | `int16` | `int32` | `signed fixed point`
Complex Number Support: Yes

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

`ready` — Indicates block is ready for new input data

scalar

Control signal that indicates that the block is ready for new input data sample on the next cycle. When **ready** is `1` (`true`), you can specify the **data** and **valid** inputs for the next time step. When **ready** is `0` (`false`), the block ignores any input data in the next time step.

Data Types: `Boolean`

Parameters

Main

`Interpolation factor source` — Source of interpolation factor
`Property` (default) | `Input port`

Select whether the block operates with a fixed or variable interpolation rate.

- `Property` — Use a fixed interpolation rate specified from the **Interpolation factor (R)** parameter.
- `Input port` — Use a variable interpolation rate specified from the **R** input port.

Note The block does not support variable interpolation for these two combinations of input and output:

- Scalar input and vector output
- Vector input and vector output

`Interpolation factor (R)` — Interpolation factor
2 (default) | integer from 1 to 2048

Specify the interpolation factor rate at which the block interpolates the input. This value must be an integer. The range of available values depends on the type of input and output data.

Input Data	Output Data	Interpolation factor (R) Valid Values
Scalar	Scalar	Integer from 1 to 2048
Scalar	Vector	Integer from 1 to 64
Vector	Vector	Integer from 1 to 64

Note For vector inputs, select the interpolation factor rate and input vector length such that their multiplication value does not exceed 64.

Dependencies

To enable this parameter, set the **Interpolation factor source** parameter to `Property`.

`Interpolation factor (Rmax)` — Upper bound of variable interpolation factor
2 (default) | integer from 1 to 2048

Specify the upper bound of the range of valid values for the **R** input port.

Note The block does not support variable interpolation for these two combinations of input and output:

- Scalar input and vector output
- Vector input and vector output

Dependencies

To enable this parameter, set the **Interpolation factor source** parameter to `Input port`.

`Differential delay (M)` — Differential delay
1 (default) | 2

Specify the differential delay of the comb part of the block.

`Number of sections (N)` — Number of integrator and comb sections
2 (default) | 1 | 3 | 4 | 5 | 6

Specify the number of sections in either the comb part or the integrator part of the block.

`Minimum number of cycles between valid input samples` — Minimum number of cycles between valid input samples
1 (default) | factors or multiples of *R*

Specify the minimum number of cycles between the valid input samples as 1, factors of *R*, or multiples of *R* based on the type of input and output data, where *R* is the interpolation factor.

Input Data	Output Data	Minimum Number of Cycles Between Valid Input Samples
Scalar	Scalar	greater than or equal to *R*
Scalar	Vector	factors less than R
Vector	Vector	1

Dependencies

To enable this parameter, set the **Interpolation factor source** parameter to `Property`.

`Gain correction` — Output gain compensation
`off` (default) | `on`

Select this parameter to compensate for the output gain of the block.

The latency of the block changes depending on the type of input, the interpolation you specify, the number of sections, and the value of this parameter. For more information on the latency of the block, see “Latency” on page 1-9.

Data Types

`Output data type` — Data type of output
`Full precision` (default) | `Same word length as input` | `Minimum section word lengths`

Select the data type for the output data.

- `Full precision` — The output data type has a word length equal to the input word length plus gain bits.
- `Same word length as input` — The output data type has a word length equal to the input word length.
- `Minimum section word lengths` — The output data type uses the word length you specify in the **Output word length** parameter.

`Output word length` — Word length of output

16 (default) | integer from 2 to 104

Specify the word length of the output as an integer from 2 to 104.

Dependencies

To enable this parameter, set the **Output data type** parameter to `Minimum section word lengths`.

Control Ports

`Enable reset input port` — Option to enable reset input port

`off` (default) | `on`

Select this parameter to enable the **reset** input port. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see “Tips” on page 1-6.

`Use HDL global reset` — Option to connect data path registers to generated HDL global reset signal

`off` (default) | `on`

Select this parameter to connect 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®. When you clear this parameter, the generated HDL global reset clears only the control path registers. 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.

For more reset considerations, see “Tips” on page 1-6.

Tips

Reset Behavior

- By default, the CIC Interpolator block connects 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 (FIL) 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 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.

Algorithms

CIC Interpolation Filter

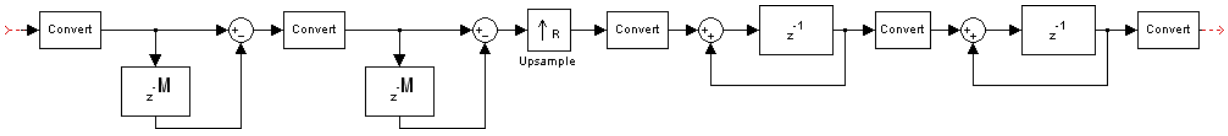
The transfer function of a CIC interpolation filter is

$$H(z) = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N = \frac{(1-z^{-RM})^N}{(1-z^{-1})^N} = \frac{(1-z^{-RM})^N}{1} \cdot \frac{1}{(1-z^{-1})^N} = H_C{}^N(z) \cdot H_I{}^N(z)\,.$$

- H_C is the transfer function of the comb part of the CIC filter.
- H_I is the transfer function of the integrator part of the CIC filter.
- N is the number of sections in either the comb part or integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- R is the interpolation factor.
- M is the differential delay.

CIC Filter Structure

The CIC Interpolator block has the CIC filter structure shown in this figure. The structure consists of N sections of cascaded comb filters, a rate change factor of R, and N sections of cascaded integrators [1].



You can locate the unit delay in the integrator part of the CIC filter in either the feedforward or feedback path. These two configurations yield an identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency of the block. Because this configuration is preferred for HDL implementation, this block puts the unit delay in the feedforward path of the integrator.

Fixed and Variable Interpolation

The block upsamples the comb stage output using R, either using the fixed interpolation rate provided using the **Interpolation factor (R)** parameter or the variable interpolation rate provided using the **R** input port. At the upsampling stage, the block uses a counter to count the valid input samples, which depend on the interpolation rate. Whenever the interpolation rate changes, the block resets and starts a new calculation from the next sample. This mechanism prevents the block from accumulating false values. Then, the block provides the interpolated output to the integrator part of the CIC filter.

Gain Correction

The gain of the CIC interpolation filter at each stage is given by

$$G_i = \left\{ \begin{array}{ll} 2^i & i = 1, 2, ..., N \\ \dfrac{2^{2N-i}(RM)^{i-N}}{R} & i = N+1, ..., 2N \end{array} \right\}.$$

- G_i is the gain at ith stage.
- R is the **Interpolation factor (R)** parameter value.
- M is the **Differential delay (M)** parameter value.
- N is the **Number of sections (N)** parameter value.

The output of the block is amplified by a specific gain value. This gain equals the gain of the $2N$th stage of the CIC interpolation filter and is given by $Gain = \dfrac{(R \text{ x } M)^N}{R}$.

The block implements gain correction in two parts: coarse gain and fine gain. In coarse gain correction, the block calculates the shift value, adds the shift value to the fractional bits to create a numeric type, and performs a bit-shift left and reinterpretcast. In fine gain correction, the block divides the remaining gain with the coarse gain if the gain is not a power of 2. Then, the block multiplies the corrected coarse gain value by the inverse value of the fine gain. Before the block starts processing, all possible shift and fine gain values are precalculated and stored in an array.

You can modify this equation to $Gain = 2^{cGain} \text{ x } fGain$. In this equation, $cGain$ is the coarse gain and $fGain$ is the fine gain. These gains are given by these equations.

- $cGain = floor(\log_2 Gain)$
- $fGain = Gain/2^{cGain} = Gain \text{ x } 2^{-cGain}$

To perform gain correction when the **Interpolation factor source** parameter is set to `Input port`, the block sets the output data type configured with the maximum interpolation rate and bit-shifts left for all of the values under the maximum interpolation rate. The bit-shift value is equal to $Maximum\ gain\ -\ \log_2(current\ gain)$.

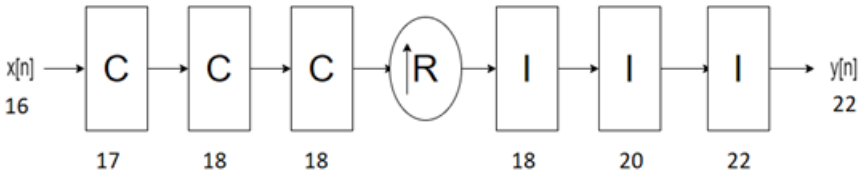
Output Data Type

The block outputs data based on the output data type selection. Consider a block with R, M, and N values of 8, 1, and 3, respectively, and an input width of 16. The word length at the ith internal stage is calculated as $B_i = B_{\text{In}} + [\log_2(G_i)]$, where:

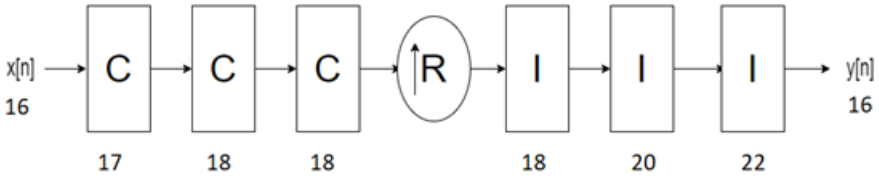
- G_i is the gain at ith stage.
- B_{In} is the input word length.
- B_i is the word length at ith stage.

The output word length is calculated as $B_{Out} = \ B_{In} + \ N - 1$, where B_{Out} is the output word length.

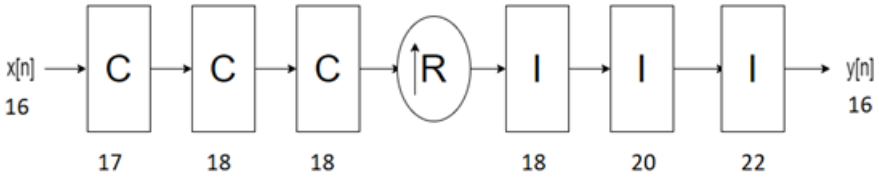
When you set the **Output data type** parameter to `Full precision`, the block outputs data with a word length of 22 by adding 6 gain bits to the input word length of 16. The word lengths of the internal comb and integrator stages are set to accommodate the bit growth.



When you set the **Output data type** parameter to `Same word length as input`, the block outputs data with a word length of 16, which is the same length as the input word length. The word lengths of the internal comb and integrator stages are set in the same way as in `Full precision` mode.



When you set the **Output data type** parameter to `Minimum section word lengths` and the **Output word length** parameter to `16`, the block outputs data with a word length of 16. The word lengths of the internal comb and integrator stages are set in the same way as in `Full precision` mode.



Latency

The latency of the block changes depending on the type of input, the interpolation you specify, the number of sections, the value of the **Gain correction** parameter, and the value of the **Minimum number of cycles between valid input samples** parameter. This table shows the latency of the block. N is the number of sections, *vecLen* is the length of the vector, and R is the interpolation factor.

Common latency is equal to 2 + (*N* x (*vecLen* x *R*)) + 3 x *N*, when *R* is equal to 1 and is equal to 3 + (*N* x (*vecLen* x *R*)) + 3 x *N*, when *R* is greater than 1.

Input Data	Output Data	Interpolation Type	Gain Correction	Minimum number of cycles between valid input samples (NumCycles)	Latency in Clock Cycles
Scalar	Scalar	Fixed	`off`	*NumCycles* = R and > R	3 + *N* 2 + *N*, when *R* = 1.
			`on`	*NumCycles* = R and > R	3 + *N* + 9 2 + *N* + 9, when *R* = 1.
Scalar	Scalar	Variable	`off`	NA	4 + *N* 3 + *N*, when R_{max} = 1.
			`on`	NA	4 + *N* + 9 3 + *N* + 9, when R_{max} = 1.
Scalar	Vector	Fixed	`off`	*NumCycles* = 1	*Common latency* + 1, when *R* is greater than *N*. *Common latency*, when *R* is less than or equal to *N*. *Common latency* – (1 + floor(*N*/(3 x *R*))), when *R* is less than *N* and (*vecLen* == 2 && (*R* == 2 && (*N* == 4 || *N* == 5 || *N* == 6)) || (*R*== 3 && *N* == 6))
				NumCycles < R	3 + *N* + ((*R* + 1) x *N* + 2) + 1 + (*N* – 1) x *NumCycles*.
			`on`	*NumCycles* = 1	*Common latency* + 1 + 9, when *R* is greater than *N*. *Common latency* + 9, when *R* is less than or equal to *N*. *Common latency* – (1 + floor(*N*/(3 x *R*))) + 9, when *R* is less than *N* and (*vecLen* == 2 && (*R* == 2 && (*N* == 4 || *N* == 5 || *N* == 6)) || (*R* == 3 && *N* == 6)).
				NumCycles < R	3 + *N* + ((*R* + 1) x *N* + 2) + 1 + (*N* – 1) x *NumCycles* + 9

Input Data	Output Data	Interpolation Type	Gain Correction	Minimum number of cycles between valid input samples (NumCycles)	Latency in Clock Cycles
Vector	Vector	Fixed	`off`	*NumCycles* = 1	*Common latency* *Common latency* – 1, when (*vecLen* == 2 && (*R* == 2 && (*N* == 4 || *N* == 5 || *N* == 6)) || (*R*== 3 && *N* == 6)) || (*vecLen* == 3 && (*R* == 2 && *N* == 6)) *Common latency* – ((*N* >1) + (*N* > 4)), when R = 1 and *vecLen* == 2. *Common latency* – ((*N* > (*vecLen* – 1)), when *R* = 1 and *vecLen* > 2.
			`on`	*NumCycles* = 1	*Common latency* + 9 *Common latency* – 1 + 9, when (*vecLen* == 2 && (*R* == 2 && (*N* == 4 || *N* == 5 || *N* == 6)) || (*R*== 3 && *N* == 6)) || (*vecLen* == 3 && (*R* == 2 && *N* == 6)) *Common latency* – ((*N* >1) + (*N* > 4)) + 9, when *R* = 1 and *vecLen* == 2. *Common latency* – ((*N* > (*vecLen* – 1)) + 9, when *R* = 1 and *vecLen* > 2.

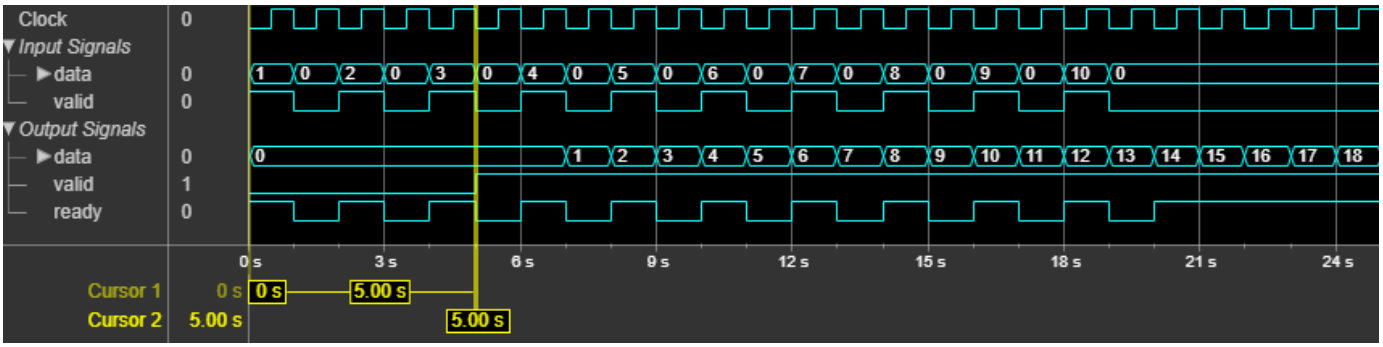
Note The block does not support variable interpolation for these two combinations of input and output:

- Scalar input and vector output
- Vector input and vector output

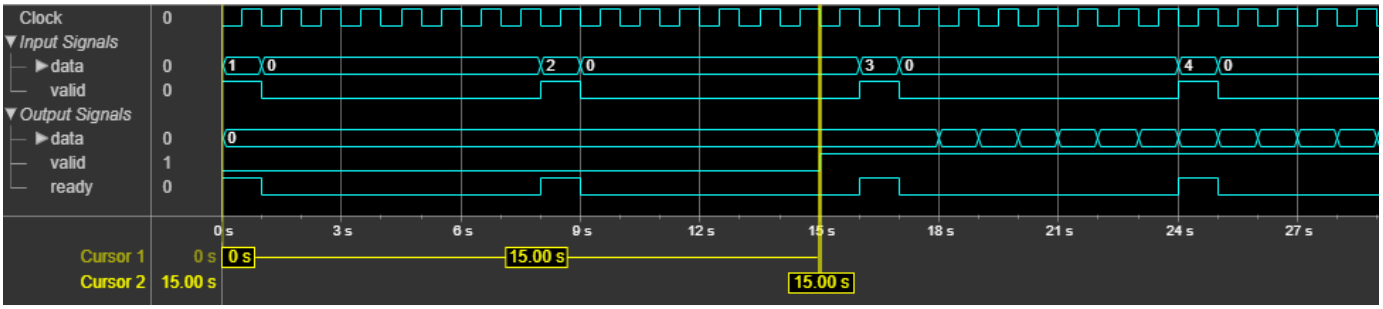
Scalar Input

This section shows the output of the block for a scalar input with different *R*, *M*, and *N* values.

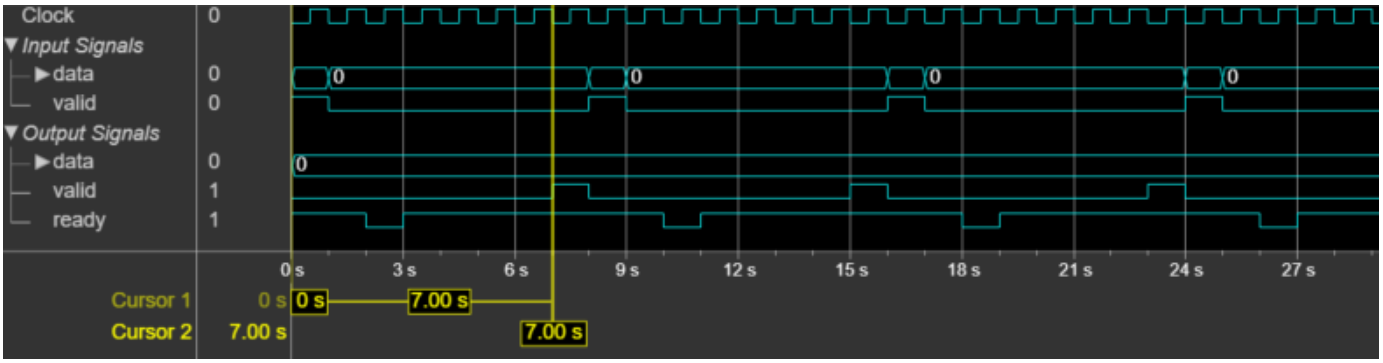
This figure shows the output of the block with the default configuration (that is, with a fixed interpolation rate and *R*, *M*, and *N* values of 2, 1, and 2, respectively). The latency of the block is 5 clock cycles and is calculated as 3 + *N*, where *N* is the number of sections.



This figure shows the output of the block with a fixed interpolation rate, *R*, *M*, and *N* values of 8, 1, and 3, respectively, and the **Gain correction** parameter selected. The latency of the block is 15 clock cycles and is calculated as 3 + *N* + 9, where *N* is the number of sections.



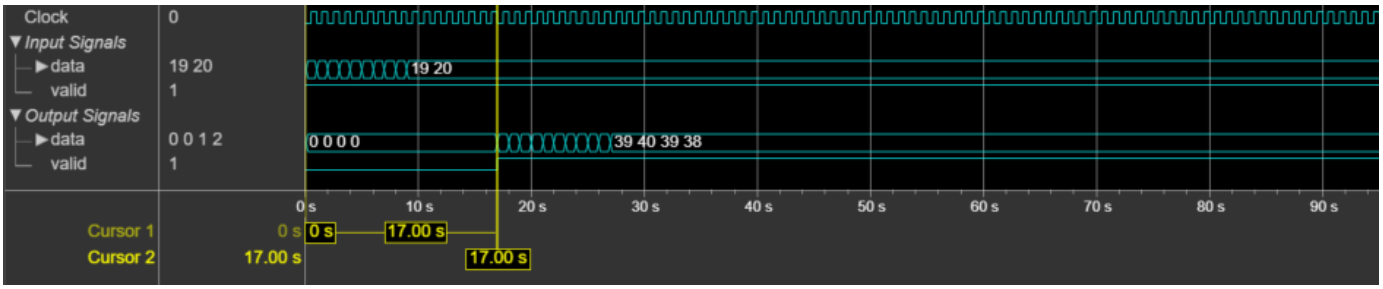
This figure shows the output of the block with variable interpolation rate (**R** input port) values of 2, 4, and 8 and with *M* and *N* values of 1 and 3, respectively. In this case, the **Gain correction** parameter is cleared. The block accepts **R** port value changes only when the **valid** input port is 1. The latency of the block is 7 clock cycles and is calculated as 4 + *N*, where *N* is the number of sections.



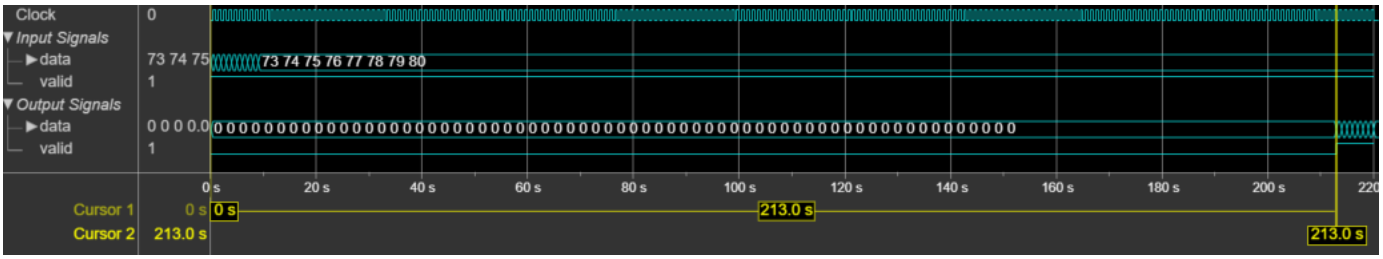
Vector Input

This section shows the output of the block for a vector input with different *R*, *M*, and *N* values.

This figure shows the output of the block for a two-element column vector input with the default configuration (that is, with a fixed interpolation rate and *R*, *M*, and *N* values of 2, 1, and 2, respectively). The latency of the block is 17 clock cycles.



This figure shows the output of the block for an eight-element column vector input with a fixed interpolation rate, *R*, *M*, and *N* values of 8, 1, and 3, respectively, and the **Gain correction** parameter selected. The latency of the block is 213 clock cycles.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. It also varies based on the input data type.

This table shows the resource and performance data synthesis results of the block for a scalar input with fixed and variable interpolation rates and for a two-element column vector of type `fixdt(1,16,0)` with a fixed interpolation rate when *R*, *M*, and *N* are 2, 1, and 2, respectively. The generated HDL code is targeted to the Xilinx® Zynq®- 7000 ZC706 Evaluation Board.

Input Data	Interpolation Type	Slice LUTs	Slice Registers	Maximum Frequency in MHz
Scalar	Fixed rate	68	90	844.12
	Variable rate	143	115	451.83
Vector	Fixed rate	480	921	376.51

The resources and frequencies vary based on the type of input data, *R*, *M*, and *N* values, and other parameter values selected in the block mask. Using a vector input can increase the throughput, however, doing so also increases the number of hardware resources that the block uses.

References

[1] Hogenauer, E. “An Economical Class of Digital Filters for Decimation and Interpolation.” *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29, no. 2 (April 1981): 155–62. https://doi.org/10.1109/TASSP.1981.1163535.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder™ provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

See Also

Objects

`dsphdl.CICInterpolator` | `dsphdl.CICDecimator`

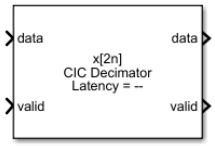
Blocks

CIC Decimator

Introduced in R2022a

CIC Decimator

Decimate signal using CIC filter
Library: DSP HDL Toolbox / Filtering



Description

The CIC Decimator block decimates an input signal by using a cascaded integrator-comb (CIC) decimation filter. CIC decimation filters are a class of linear phase finite impulse response (FIR) filters consisting of a comb part and an integrator part. The CIC decimation filter structure consists of *N* sections of cascaded integrators, a rate change factor of *R*, and *N* sections of cascaded comb filters. For more information about CIC decimation filters, see “Algorithms” on page 1-19.

The block supports these combinations of input and output data.

- Scalar input and scalar output — Support for fixed and variable decimation rates
- Vector input and scalar output — Support for fixed decimation rates only
- Vector input and vector output — Support for fixed decimation rates only

The block provides an architecture suitable for HDL code generation and hardware deployment.

Ports

Input

`data` — Input data
scalar | column vector

Input data, specified as a scalar or a column vector with a length from 1 to 64. The input data must be a signed integer or a signed fixed point with a word length less than or equal to 32. The **Decimation factor (R)** parameter must be an integer multiple of the input frame size.

Data Types: `int8` | `int16` | `int32` | `signed fixed point`
Complex Number Support: Yes

`valid` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

`R` — Variable decimation rate
scalar

Use this port to dynamically specify the variable decimation rate during run time.

This value must have the data type `fixdt(0,12,0)` and it must be an integer in the range from 1 to the **Decimation factor (Rmax)** parameter value.

Dependencies

To enable this port, on the **Main** tab, set the **Decimation factor source** parameter to `Input port`.

Data Types: `fixdt(0,12,0)`

`reset` — Clears internal states

scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — CIC-decimated output data

scalar | column vector

The block returns filtered output data as a scalar or a column vector with a length from 1 to 64. You can define the data type of this output by setting the **Output data type** parameter on the **Data Types** tab.

Data Types: `int8` | `int16` | `int32` | `signed fixed point`
Complex Number Support: Yes

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

Parameters

Main

`Decimation factor source` — Source of decimation factor

`Property` (default) | `Input port`

Select whether the block operates with a fixed or variable decimation rate.

- `Property` — Use a fixed decimation rate specified from the **Decimation factor (R)** parameter.
- `Input port` — Use a variable decimation rate specified from the **R** input port.

Note For vector inputs, the block does not support variable decimation.

`Decimation factor (R)` — Decimation factor
2 (default) | integer from 1 to 2048

Specify the decimation factor rate at which the block decimates the input.

Dependencies

To enable this parameter, set the **Decimation factor source** parameter to `Property`.

`Decimation factor (Rmax)` — Upper bound of variable decimation factor
2 (default) | integer from 1 to 2048

Specify the upper bound of the range of valid values for the **R** input port.

Note For vector inputs, the block does not support variable decimation.

Dependencies

To enable this parameter, set the **Decimation factor source** parameter to `Input port`.

`Differential delay (M)` — Differential delay
1 (default) | 2

Specify the differential delay of the comb part of the block.

`Number of sections (N)` — Number of integrator and comb sections
2 (default) | 1 | 3 | 4 | 5 | 6

Specify the number of sections in either the comb part or the integrator part of the block.

`Gain correction` — Output gain compensation
`off` (default) | `on`

Select this parameter to compensate for the output gain of the block.

The latency of the block changes depending on the type of input, the decimation you specify, the number of sections, and the value of this parameter. For more information on the latency of the block, see “Latency” on page 1-22.

Data Types

`Output data type` — Data type of output
`Full precision` (default) | `Same word length as input` | `Minimum section word lengths`

Select the data type for the output data.

- `Full precision` — The output data type has a word length equal to the input word length plus gain bits.

- `Same word length as input` — The output data type has a word length equal to the input word length.
- `Minimum section word lengths` — The output data type uses the word length you specify in the **Output word length** parameter. When you select this option, the block applies the pruning algorithm. For more information about the pruning algorithm, see [1].

`Output word length` — Word length of output
16 (default) | integer from 2 to 104

Specify the word length of the output.

Note When this value is 2, 3, 4, 5, or 6, the block can overflow the output data.

Dependencies

To enable this parameter, set the **Output data type** parameter to `Minimum section word lengths`.

Control Ports

`Enable reset input port` — Reset signal
`off` (default) | `on`

Select this parameter to enable the **reset** input port. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see “Tips” on page 1-18.

`Use HDL global reset` — Option to connect data path registers to generated HDL global reset signal
`off` (default) | `on`

Select this parameter to connect 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. When you clear this parameter, the generated HDL global reset clears only the control path registers. 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.

For more reset considerations, see “Tips” on page 1-18.

Tips

Reset Behavior

- By default, the CIC Decimator block connects 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 (FIL) 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 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.

Algorithms

CIC Decimation Filter

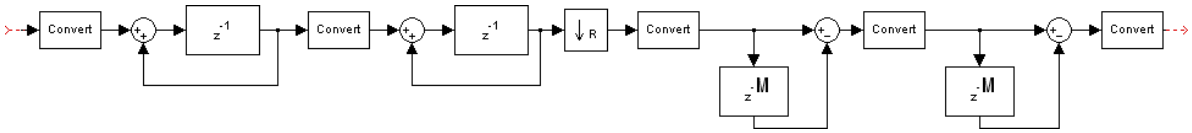
The transfer function of a CIC decimation filter is

$$H(z) = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N = \frac{\left(1-z^{-RM}\right)^N}{\left(1-z^{-1}\right)^N} = \frac{1}{\left(1-z^{-1}\right)^N} \cdot \frac{\left(1-z^{-RM}\right)^N}{1} = H_I{}^N(z) \cdot H_C{}^N(z)\,.$$

- H_I is the transfer function of the integrator part of the CIC filter.
- H_C is the transfer function of the comb part of the CIC filter.
- N is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part or integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- R is the decimation factor.
- M is the differential delay.

CIC Filter Structure

The CIC Decimator block has the CIC filter structure shown in this figure. The structure consists of N sections of cascaded integrators, a rate change factor of R, and N sections of cascaded comb filters [1].



You can locate the unit delay in the integrator part of the CIC filter in either the feedforward or feedback path. These two configurations yield an identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency of the block. Because this configuration is preferred for HDL implementation, this block puts the unit delay in the feedforward path of the integrator.

Fixed and Variable Decimation

The block downsamples the integrator stage output using R, either based on the fixed decimation rate provided using the **Decimation factor (R)** parameter or the variable decimation rate provided using the **R** input port. At the downsampler stage, the block uses a counter to count the valid input samples, which depend on the decimation rate. Whenever the decimation rate changes, the block resets and starts a new calculation from the next sample. This mechanism prevents the block from accumulating false values. Then, the block provides the decimated output to the comb part of the CIC filter.

Gain Correction

The gain of the block is given by $Gain = (R \times M)^N$.

- R is the **Decimation factor (R)** parameter value.
- M is the **Differential delay (M)** parameter value.
- N is the **Number of sections (N)** parameter value.

The block implements gain correction in two parts: coarse gain and fine gain. In coarse gain correction, the block calculates the shift value, adds the shift value to the fractional bits to create a numeric type, and performs a bit-shift left and reinterpretcast. In fine gain correction, the block divides the remaining gain with the coarse gain if the gain is not a power of 2. Then, the block multiplies the corrected coarse gain corrected value with the inverse value of the fine gain. Before the block starts processing, all possible shift and fine gain values are precalculated and stored in an array.

You can modify this equation as $Gain = 2^{cGain} \times fGain$. In this equation, $cGain$ is the coarse gain, and $fGain$ is the fine gain. These gains are given by these equations.

- $cGain = floor(\log_2 Gain)$
- $fGain = Gain/2^{cGain} = Gain \times 2^{-cGain}$

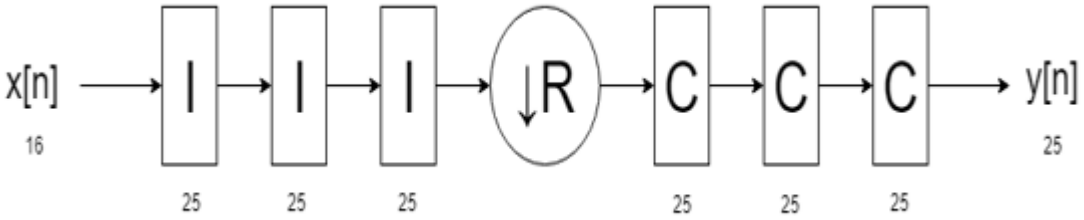
To perform gain correction when the **Decimation factor source** parameter is set to `Input port`, the block sets the output data type configured with the maximum decimation rate and bit-shifts left for all of the values under the maximum decimation rate. The bit-shift value is equal to $Maximum\ gain - \log_2(current\ gain)$.

Output Data Type

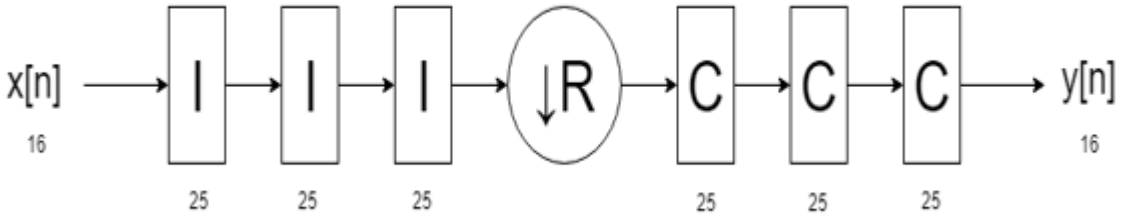
The block outputs data based on the output data type selection. Consider a block with R, M, and N values of 8, 1, and 3, respectively, and an input width of 16. The output word length is calculated as $B_{\text{Out}} = B_{\text{In}} + [\log_2(Gain)]$.

- $Gain = (R \times M)^N$
- B_{In} is the input word length.
- B_{Out} is the output word length.

When you set the **Output data type** parameter to `Full precision`, the block outputs data with a word length of 25 by adding 9 gain bits to the input word length of 16.

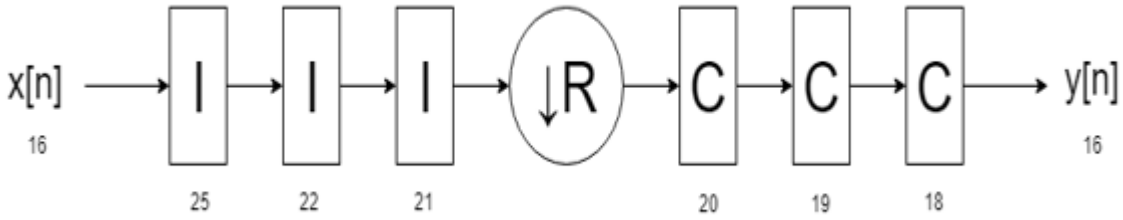


When you set the **Output data type** parameter to `Same word length as input`, the block outputs data with a word length of 16, which is the same length as the input word length. The internal integrator and comb stages use the full-precision data type with 25 bits.



When you set the **Output data type** parameter to `Minimum section word lengths` and the **Output word length** parameter to `16`, the block outputs data with a word length of 16. In this case, the block changes the bit width at each stage, based on the pruning algorithm.

If the **Output word length** parameter value is less than the number of bits required at the block output, the least significant bits (LSBs) at the earlier stages are pruned. The Hogenauer algorithm [1] provides the number of LSBs to discard at each stage. This algorithm minimizes the loss of information in the output data.



Latency

The latency of the block changes depending on the type of input, the decimation you specify, the number of sections, and the value of the **Gain correction** parameter. This table shows the latency of the block. *N* is the number of sections and *vecLen* is the length of the vector.

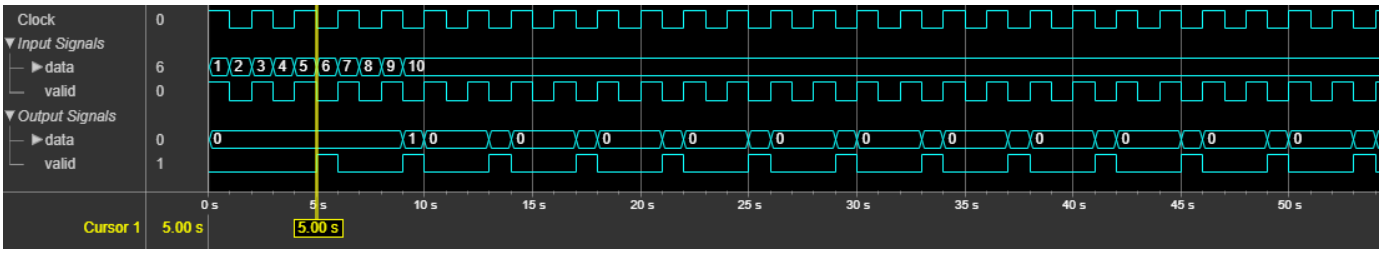
Input Data	Output Data	Decimation Type	Gain Correction	Latency in Clock Cycles
Scalar	Scalar	Fixed	`off`	3 + *N*. When *R* = 1, 2 + *N*.
			`on`	3 + *N* + 9. When *R* = 1, 2 + *N* + 9.
Scalar	Scalar	Variable	`off`	4 + *N*. When R_{max} = 1, 3 + *N*.
			`on`	4 + *N* + 9. When R_{max} = 1, 3 + *N* + 9.
Vector	Scalar	Fixed	`off`	`floor`((*vecLen* – 1) x (*N*/*vecLen*)) + 1 + *N* + (2 + (*vecLen* + 1) x *N*
			`on`	`floor`((*vecLen* – 1) x (*N*/*vecLen*)) + 1 + *N* + (2 + (*vecLen* + 1) x *N*) + 9
Vector	Vector	Fixed	`off`	`floor`((*vecLen* – 1) x (*N*/*vecLen*)) + 1 + *N* + (2 + (*vecLen* + 1) x *N*
			`on`	`floor`((*vecLen* – 1) x (*N*/*vecLen*)) + 1 + *N* + (2 + (*vecLen* + 1) x *N*) + 9

Note For vector inputs, the block does not support variable decimation.

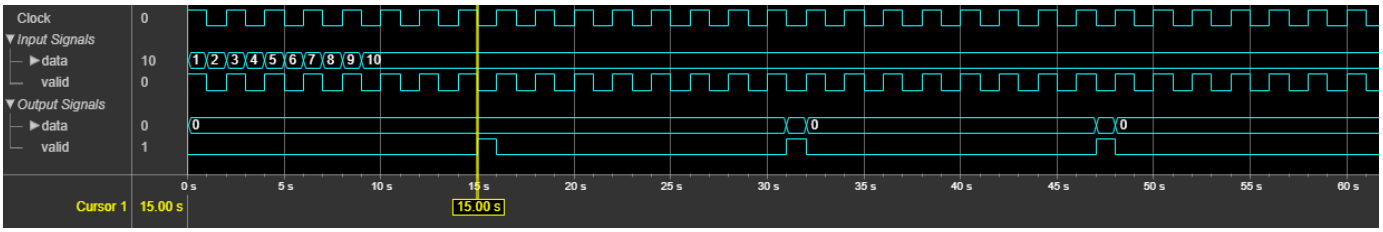
Scalar Input

This section shows the output of the block for a scalar input with different *R*, *M*, and *N* values.

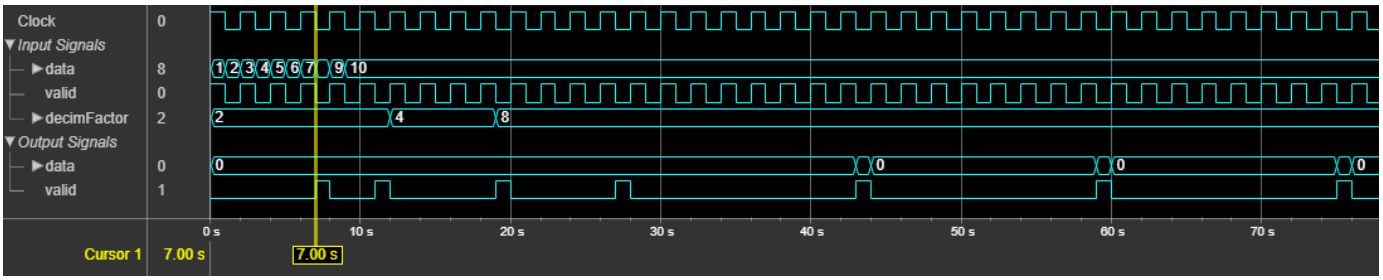
This figure shows the output of the block with the default configuration (that is, with a fixed decimation rate and *R*, *M*, and *N* values of 2, 1, and 2, respectively). The block returns valid output data at every second cycle based on the fixed **Decimation factor (R)** parameter value of 2. The latency of the block is 5 clock cycles and is calculated as 3 + *N*, where *N* is the number of sections.



This figure shows the output of the block with a fixed decimation rate, *R*, *M*, and *N* values of 8, 1, and 3, respectively, and the **Gain correction** parameter selected. The block returns valid output data at every eighth cycle based on the fixed **Decimation factor (R)** parameter value of 8. The latency of the block is 15 clock cycles and is calculated as 3 + *N* + 9, where *N* is the number of sections.



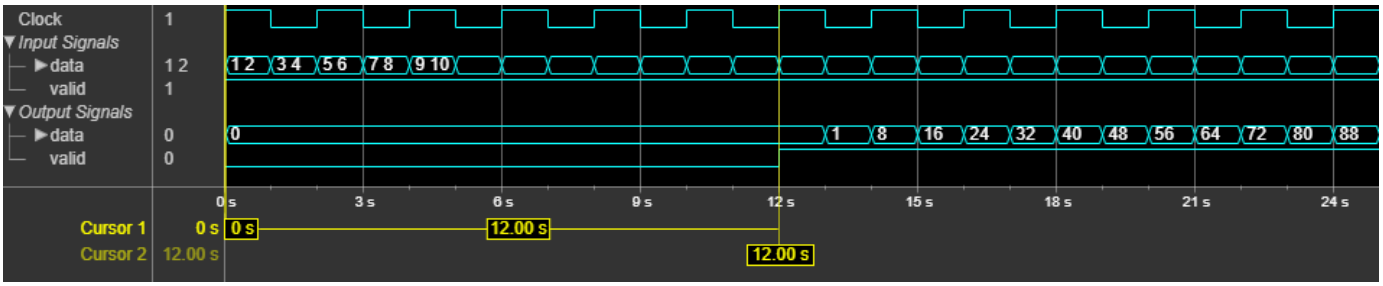
This figure shows the output of the block with variable decimation rate (**R** input port) values of 2, 4, and 8 and with *M* and *N* values of 1 and 3, respectively. In this case, the **Gain correction** parameter is cleared. The block returns valid output data at the second, fourth, and eighth cycles corresponding to the **R** port values 2, 4, and 8, respectively. The block accepts **R** port value changes only when the **valid** input port is 1. The latency of the block is 7 clock cycles and is calculated as 4 + *N*, where *N* is the number of sections.



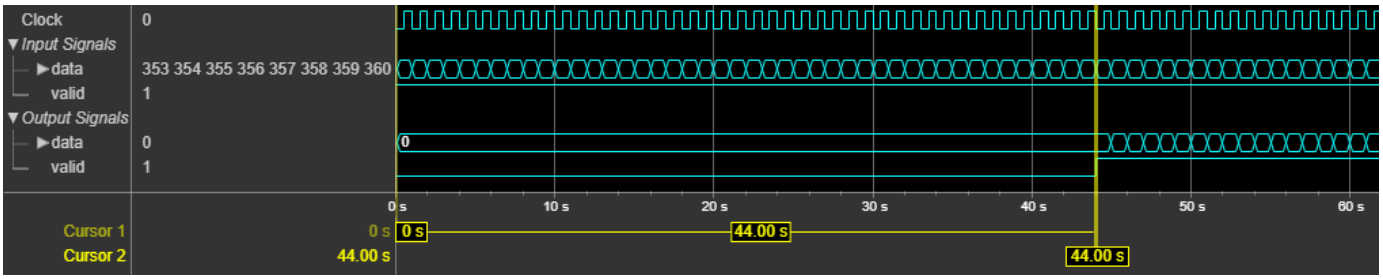
Vector Input

This section shows the output of the block for a vector input with different *R*, *M*, and *N* values.

This figure shows the output of the block for a two-element column vector input with the default configuration, (that is, with a fixed decimation rate and *R*, *M*, and *N* values of 2, 1, and 2, respectively). The latency of the block is 12 clock cycles.



This figure shows the output of the block for an eight-element column vector input with a fixed decimation rate, *R*, *M*, and *N* values of 8, 1, and 3, respectively, and the **Gain correction** parameter selected. The latency of the block is 44 clock cycles.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. It also varies based on the input data type.

This table shows the resource and performance data synthesis results of the block for a scalar input of type `fixdt(1,16,0)` with fixed and variable decimation rates and for a two-element column vector input with a fixed decimation rate when *R*, *M*, and *N* are 2, 1, and 2, respectively. The generated HDL is targeted to the Xilinx Zynq- 7000 ZC706 Evaluation Board.

Input Data	Decimation Type	Slice LUTs	Slice Registers	Maximum Frequency in MHz
Scalar	Fixed rate	101	166	711.74
	Variable rate	206	186	441.70
Vector	Fixed rate	218	627	624.61

The resources and frequencies vary based on the type of input data and the values of *R*, *M*, and *N*, as well as other parameter values selected in the block mask. Using a vector input can increase the throughput, however, doing so also increases the number of hardware resources that the block uses.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox

Behavior changed in R2022a

Before R2022a, this block was named CIC Decimation HDL Optimized, and was included in the DSP System Toolbox™ **DSP System Toolbox HDL Support** library.

Changes to decimation factor parameters

Behavior changed in R2022a

In previous releases, a decimation factor of 1 was invalid. You can now set the decimation factor to 1.

Configuration	Before R2022a	After 2022a
Variable decimation factor	Select the **Variable decimation** parameter and set the **Decimation factor (R)** parameter to the maximum expected decimation factor.	Set the **Decimation factor source** parameter to `Input port` and set the **Decimation factor (Rmax)** parameter to the maximum expected decimation factor. The **decimFactor** port is renamed to **R**.
Fixed decimation factor	Clear the **Variable decimation** parameter and set the **Decimation factor (R)** parameter to the desired decimation factor.	Set the **Decimation factor source** parameter to `Property` and set the **Decimation factor (R)** to the desired decimation factor.

References

[1] Hogenauer, E. “An Economical Class of Digital Filters for Decimation and Interpolation.” *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29, no. 2 (April 1981): 155–62. https://doi.org/10.1109/TASSP.1981.1163535.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).

InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

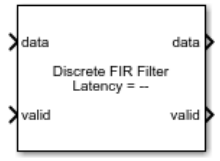
Objects
`dsphdl.CICDecimator` | `dsphdl.CICInterpolator`

Blocks
CIC Interpolator

Introduced in R2019b

Discrete FIR Filter

Finite impulse response filter
Library: DSP HDL Toolbox / Filtering



Description

The Discrete FIR Filter block models finite-impulse response filter architectures optimized for HDL code generation. The block accepts scalar or frame-based input, and provides an option for programmable coefficients. It provides a hardware-friendly interface with input and output control signals. To provide a cycle-accurate simulation of the generated HDL code, the block models architectural latency including pipeline registers and resource sharing.

The block provides three filter structures. The direct form systolic architecture provides a fully parallel implementation that makes efficient use of Intel® and Xilinx DSP blocks. The direct form transposed architecture is a fully parallel implementation and is suitable for FPGA and ASIC applications. The partly serial systolic architecture provides a configurable serial implementation that makes efficient use of FPGA DSP blocks. 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.

All three structures optimize hardware resources by sharing multipliers for symmetric or antisymmetric filters and by removing the multipliers for zero-valued coefficients such as in half-band filters and Hilbert transforms.

The latency between valid input data and the corresponding valid output data depends on the filter structure, serialization options, the number of coefficients, and whether the coefficient values provide optimization opportunities. For details of structure and latency, see the “Algorithm” on page 1-33 section.

For a FIR filter with multichannel support, use the Discrete FIR Filter block instead.

Ports

Input

`data` — Input data

scalar or column vector of real or complex values

Input data, specified as a scalar or column vector of real or complex values. The vector size must be a power of 2 in the range from 1 to 64. When the input data type is an integer type or a fixed-point type, the block uses fixed-point arithmetic for internal calculations.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fixed point` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`valid` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

`coeff` — Filter coefficients

real or complex row vector

Filter coefficients, specified as a row vector of real or complex values. You can change the input coefficients at any time. The size of the vector depends on the size and symmetry of the sample coefficients specified in the **Coefficients prototype** parameter. The prototype specifies a sample coefficient vector that is representative of the symmetry and zero-valued locations of the expected input coefficients. The block uses the prototype to optimize the filter by sharing multipliers for symmetric or antisymmetric coefficients, and removing multipliers for zero-valued coefficients. Therefore, provide only the nonduplicate coefficients at the port. For example, if you set the **Coefficients prototype** parameter to a symmetric 14-tap filter, the block expects a vector of 7 values on the **coeff** input port. You must still provide zeros in the input **coeff** vector for the nonduplicate zero-valued coefficients.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Dependencies

To enable this port, set **Coefficients source** to `Input port (Parallel interface)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

`reset` — Clears internal states

scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select **Enable reset input port**.

Data Types: `Boolean`

Output

`data` — Filtered output data

scalar or column vector of real or complex values

Filtered output data, returned as a scalar or column vector of real or complex values. The dimensions of the output match the dimensions of the input. When the input data type is a floating-point type, the output data inherits the data type of the input data. When the input data type is an integer type or a fixed-point type, the **Output** parameter on the **Data Types** tab controls the output data type.

Data Types: `fixed point` | `single` | `double`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

`ready` — Indicates block is ready for new input data

scalar

Control signal that indicates that the block is ready for new input data sample on the next cycle. When **ready** is `1` (`true`), you can specify the **data** and **valid** inputs for the next time step. When **ready** is `0` (`false`), the block ignores any input data in the next time step.

When using the partly-serial architecture, the block processes one sample at a time. If your design waits for this block to return ready set to `0` before setting the input **valid** to `0` (`false`), then one additional cycle of input data arrives at the port. The block stores this additional data while processing the current data, and then does not set **ready** to `1` (`true`), until your model processes the additional input data.

Dependencies

To enable this port, set **Filter structure** to `Partly serial systolic`.

Data Types: `Boolean`

Parameters

Main

`Coefficient source` — Source of filter coefficients

`Property` (default) | `Input port (Parallel interface)`

You can enter constant filter coefficients as a parameter or provide time-varying filter coefficients using an input port.

Selecting `Input port (Parallel interface)` enables the **coeff** port on the block and the **Coefficients prototype** parameter. Specify a prototype to enable the block to optimize the filter implementation according to the symmetry of your coefficients. To use `Input port (Parallel interface)`, set the **Filter structure** parameter to `Direct form systolic` or `Direct form transposed`.

When you use programmable coefficients with frame-based input, the output after a change of coefficient values may not exactly match the output in the scalar case. This behavior is because, when the filter is decomposed into subfilters, the filter calculations are done at different times relative to the input coefficient values, compared with the scalar implementation.

`Coefficients` — Discrete FIR filter coefficients

`[0.5, 0.5]` (default) | real or complex vector

Discrete FIR filter coefficients, specified as a vector of real or complex values. You can also specify the vector as a workspace variable or as a call to a filter design function. When the input data type is

a floating-point type, the block casts the coefficients to the same data type as the input. When the input data type is an integer type or a fixed-point type, you can set the data type of the coefficients on the **Data Types** tab.

Example: `firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0])`

Dependencies

To enable this parameter, set **Coefficients source** to `Property`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`Coefficients prototype` — Prototype filter coefficients

`[]` (default) | real or complex vector

Prototype filter coefficients, specified as a vector of real or complex values. The prototype specifies a sample coefficient vector that is representative of the symmetry and zero-valued locations of the expected input coefficients. If all of your input coefficient vectors have the same symmetry and zero-valued coefficient locations, set **Coefficients prototype** to one of those vectors. If your coefficients are unknown or not expected to share symmetry or zero-valued locations, set **Coefficients prototype** to `[]`. The block uses the prototype to optimize the filter by sharing multipliers for symmetric or antisymmetric coefficients, and removing multipliers for zero-valued coefficients.

Coefficient optimizations affect the expected size of the vector on the **coeff** port. Provide only the nonduplicate coefficients at the port. For example, if you set the **Coefficients prototype** parameter to a symmetric 14-tap filter, the block shares one multiplier between each pair of duplicate coefficients, so the block expects a vector of 7 values on the **coeff** port. You must still provide zeros in the input **coeff** vector for the nonduplicate zero-valued coefficients.

Dependencies

To enable this parameter, set **Coefficients source** to `Input port (Parallel interface)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`Filter structure` — HDL filter architecture

`Direct form systolic` (default) | `Direct form transposed` | `Partly serial systolic`

Specify the HDL filter architecture as one of these structures:

- `Direct form systolic` — This architecture provides a fully parallel filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For architecture details, see “Fully Parallel Systolic Architecture”.
- `Direct form transposed` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. For architecture details, see “Fully Parallel Transposed Architecture”.
- `Partly serial systolic` — This architecture provides a serial filter implementation and options for tradeoffs between throughput and resource utilization. It makes efficient use of Intel and Xilinx DSP blocks. The block implements a serial L-coefficient filter with M multipliers and requires input samples that are at least N cycles apart, such that $L = N \times M$. You can specify either M or N. For this implementation, the block provides an output port, **ready**, that indicates when the block is ready for new input data. For architecture details, see “Partly Serial Systolic Architecture ($1 < N < L$)” and “Fully Serial Systolic Architecture ($N \geq L$)”. You cannot use frame-based input with the partly-serial architecture.

All implementations share multipliers for symmetric and antisymmetric coefficients and remove multipliers for zero-valued coefficients.

`Specify serialization factor as` — Rule to define serial implementation

`Minimum number of cycles between valid input samples` (default) | `Maximum number of multipliers`

You can specify the rule that the block uses to serialize the filter as either:

- `Minimum number of cycles between valid input samples` – Specify a requirement for input data timing using the **Number of cycles** parameter.
- `Maximum number of multipliers` – Specify a requirement for resource usage using the **Number of multipliers** parameter.

For a filter with L coefficients, the block implements a serial filter with not more than M multipliers and requires input samples that are at least N cycles apart, such that $L = N \times M$. The block might remove multipliers when it applies coefficient optimizations, so the actual M or N value of the filter implementation can be lower than the value that you specified.

Dependencies

To enable this parameter, set the **Filter structure** parameter to `Partly serial systolic`.

`Number of cycles` — Serialization requirement for input timing

2 (default) | positive integer

Serialization requirement for input timing, specified as a positive integer. This parameter represents N, the minimum number of cycles between valid input samples. In this case, the block calculates $M = L/N$. To implement a fully-serial architecture, set **Number of cycles** greater than the filter length, L, or to `Inf`.

The block might remove multipliers when it applies coefficient optimizations, so the actual M and N values of the filter can be lower than the value you specified.

Dependencies

To enable this parameter, set **Filter structure** to `Partly serial systolic` and set **Specify serialization factor as** to `Minimum number of cycles between valid input samples`.

`Number of multipliers` — Serialization requirement for resource usage

2 (default) | positive integer

Serialization requirement for resource usage, specified as a positive integer. This parameter represents M, the maximum number of multipliers in the filter implementation. In this case, the block calculates $N = L/M$. If the input data is complex, the block allocates `floor(`M`/2)` multipliers for the real part of the filter and `floor(`M`/2)` multipliers for the imaginary part of the filter. To implement a fully-serial architecture, set **Number of multipliers** to `1` for real input with real coefficients, `2` for complex input and real coefficients or real coefficients with complex input, or `3` for complex input and complex coefficients.

The block might remove multipliers when it applies coefficient optimizations, so the actual M and N values of the filter can be lower than the value you specified.

Dependencies

To enable this parameter, set the **Filter structure** to `Partly serial systolic`, and set **Specify serialization factor as** to `Maximum number of multipliers`.

Data Types

`Rounding mode` — Rounding mode for type-casting the output

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Rounding mode for type-casting the output to the data type specified by the **Output** parameter. When the input data type is floating point, the block ignores this parameter. For more details, see "Rounding Modes".

`Saturate on integer overflow` — Overflow handling for type-casting the output

`off` (default) | `on`

Overflow handling for type-casting the output to the data type specified by the **Output** parameter. When the input data type is floating point, the block ignores this parameter. For more details, see "Overflow Handling".

`Coefficients` — Data type of discrete FIR filter coefficients

`Inherit: Same word length as input` (default) | `<data type expression>`

The block casts the filter coefficients to this data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the block ignores this parameter.

The recommended data type for this parameter is `Inherit: Same word length as input`.

The block returns a warning or error if:

- The coefficients data type does not have enough fractional length to represent the coefficients accurately.
- The coefficients data type is unsigned while the coefficients include negative values.

Dependencies

To enable this parameter, set **Coefficients source** to `Property`.

`Output` — Data type of filter output

`Inherit: Inherit via internal rule` (default) | `Inherit: Same word length as input` | `<data type expression>`

The block casts the output of the filter to this data type. The quantization uses the settings of the **Rounding mode** and **Overflow mode** parameters. When the input data type is floating point, the block ignores this parameter.

The block increases the word length for full precision inside each filter tap and casts the final output to the specified type. The maximum final internal data type (*WF*) depends on the input data type (*WI*), the coefficient data type (*WC*), and the number of coefficients (*L*) and is given by

$WF = WI + WC + \text{ceil}(\log2(L))$.

When you specify a fixed set of coefficients, because the coefficient values limit the potential growth, usually the actual full-precision internal word length is smaller than *WF*.

When you use programmable coefficients, the block cannot calculate the dynamic range, and the internal data type is always *WF*.

Control Ports

`Enable reset input port` — Option to enable reset input port
`off` (default) | `on`

Select this parameter to enable the **reset** input port. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

`Use HDL global reset` — Option to connect data path registers to generated HDL global reset signal
`off` (default) | `on`

Select this parameter to connect 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. When you clear this parameter, the generated HDL global reset clears only the control path registers. 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.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Algorithms

The filter architectures for the Discrete FIR Filter block are shared with other blocks and described in detail on the “FIR Filter Architectures for FPGAs and ASICs” page. The sections here show the hardware resources and synthesized clock speed for the Discrete FIR Filter block configured with each filter architecture.

Performance — Fully Parallel Systolic

This table shows post-synthesis resource utilization for the HDL code generated for a symmetric 26-tap FIR filter with 16-bit input and 16-bit coefficients. The synthesis targets a Xilinx ZC-706 (XC7Z045ffg900-2) FPGA. The **Global HDL reset type** parameter is `Synchronous` and **Minimize clock enables** is selected. The **reset** port is not enabled, so only control path registers are connected to the generated global HDL reset.

Resource	Uses
LUT	36
Slice Reg	487
Slice	45
Xilinx LogiCORE DSP48	13

After place and route, the maximum clock frequency of the design is 630 MHz.

Performance — Fully Parallel Transposed

This table shows post-synthesis resource utilization for the HDL code generated for a symmetric 26-tap FIR filter with 16-bit input and 16-bit coefficients. The synthesis targets a Xilinx ZC-706 (XC7Z045ffg900-2) FPGA. The **Global HDL reset type** parameter is `Synchronous` and **Minimize clock enables** is selected. The **reset** port is not enabled, so only control path registers are connected to the generated global HDL reset.

Resource	Uses
LUT	32
Slice Reg	108
Xilinx LogiCORE DSP48	26

After place and route, the maximum clock frequency of the design is 541 MHz.

Performance — Partly Serial Systolic (1 < *N* < *L*)

This table shows post-synthesis resource utilization for the HDL code generated from the “Partly Serial Systolic FIR Filter Implementation” example. The implementation is for a 32-tap FIR filter with 16-bit input, 16-bit coefficients, and a serialization factor of 8 cycles between valid input samples. The synthesis targets a Xilinx Virtex-6 (XC6VLX240T-1FF1156) FPGA. The **Global HDL reset type** parameter is `Synchronous` and **Minimize clock enables** is selected.

Resource	Uses
LUT	181
FFS	428
Xilinx LogiCORE DSP48	2

After place and route, the maximum clock frequency of the design is 561 MHz.

Performance — Fully Serial Systolic (*N* ≥ *L*)

Resource	Uses
LUT	122
Slice Reg	225
Xilinx LogiCORE DSP48	1

After place and route, the maximum clock frequency of the design is 590 MHz.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox

Behavior changed in R2022a

Before R2022a, this block was named Discrete FIR Filter HDL Optimized, and was included in the DSP System Toolbox **DSP System Toolbox HDL Support** library.

High-throughput interface

This block supports high-throughput data. You can apply input data as a *N*-by-1 vector, where *N* can be up to 64 values. You cannot use frame-based input with the partly-serial architecture.

Input coefficients must be a row vector
Behavior changed in R2022a

When you use programmable coefficients with this block, you must supply the coefficients as a row vector (1-by-*N* matrix). Before R2022a, the block accepted a one-dimensional array (for example, `ones(5)`), a column vector(*M*-by-1 matrix), or a row vector of coefficients.

RAM-based party-serial architecture

This block uses a RAM-based partly-serial architecture which uses fewer resources than the former register-based architecture. Uninitialized RAM locations can result in `X` values at the start of your HDL simulation. You can avoid `X` values by having your test initialize the RAM, or by enabling the **Initialize all RAM blocks** Configuration Parameter. This parameter sets the RAM locations to `0` for simulation and is ignored by synthesis tools.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

For a FIR filter with multichannel, use the Discrete FIR Filter block instead.

HDL Architecture

The block provides three filter structures. The direct form systolic architecture provides a fully parallel implementation that makes efficient use of Intel and Xilinx DSP blocks. The direct form transposed architecture is a fully parallel implementation and is suitable for FPGA and ASIC applications. The partly serial systolic architecture provides a configurable serial implementation that also makes efficient use of FPGA DSP blocks. 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.

All three structures optimize hardware resources by sharing multipliers for symmetric or antisymmetric filters and by removing the multipliers for zero-valued coefficients such as in half-band filters and Hilbert transforms.

You can set block parameters to make tradeoffs between throughput and resource utilization.

- For highest throughput, choose a fully parallel systolic or transposed architecture. The generated code can accept input data and provides filtered output data on every cycle.
- For reduced area, choose partly serial systolic architecture. Then specify a rule that the block uses to serialize the filter based on either input timing or resource usage. To specify a serial filter using

an input timing rule, set **Specify serialization factor as** to `Minimum number of cycles between valid input samples`, and choose **Number of cycles** to be greater than or equal to 2. In this case, the filter accepts only input samples that are at least **Number of cycles** cycles apart. To specify a serial filter using a resource rule, set **Specify serialization factor as** to `Maximum number of multipliers`, and set **Number of multipliers** to be less than the number of filter coefficients. In this case, the filter accepts input samples that are at least `NumCoeffs/NumMults` apart.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

The Discrete FIR Filter block does not support resource sharing optimization through HDL Coder settings. Instead, set the **Filter structure** parameter to `Partly serial systolic`, and configure a serialization factor based on either input timing or resource usage.

See Also

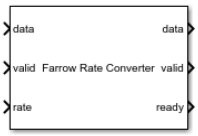
Objects
`dsphdl.FIRFilter`

Blocks
FIR Decimator | FIR Rate Converter | FIR Interpolator

Introduced in R2017a

Farrow Rate Converter

Polynomial sample-rate converter
Library: DSP HDL Toolbox / Signal Operations



Description

The Farrow Rate Converter block converts the sample rate of a signal by using FIR filters to implement a polynomial sinc approximation. A Farrow filter is an efficient rate converter when the rate conversion factor is a ratio of large integer decimation and interpolation factors. Specify the rate conversion factor by providing the input sample rate and the desired output sample rate. You can provide the rate conversion factor as a fixed parameter or as a time-varying input signal.

You can use this block with the default coefficients for most rate conversions. The default coefficients are a LaGrange interpolation that matches the Farrow Rate Converter block in DSP System Toolbox. Or, you can specify a custom set of coefficients if the default does not meet your specifications.

The block provides a hardware-friendly interface with input and output control signals. To provide a cycle-accurate simulation of the generated HDL code, the block models architectural latency including pipeline registers and multiplier optimizations.

Ports

Input

`data` — Input data
real or complex scalar

Input data, specified as a real or complex scalar. When the input data type is an integer type or a fixed-point type, the block uses fixed-point arithmetic for internal calculations.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fixed point` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`valid` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

`rate` — Rate change factor
scalar

Specify the rate change factor as a positive rational value that is the ratio of the input sample rate and the output sample rate, F_{in}/F_{out}. There are no limits on the rate change factor.

When this input value changes, the block resets the internal phase accumulator. This reset means you can change the rate change factor from decimation to interpolation. For example, you can use this block to align data streams that have similar but varying sample clocks.

The block derives the data type of the internal accumulator from the data type of this signal. The data type of the rate change must have at least one integer bit and one fractional bit. The accumulator data type is `fixdt(1,fractionalWL+1,fractionalWL)`, where *fractionalWL* is the fraction length of the rate change data type. The *fractionalWL* determines the accuracy of the phase timing, but also increases the critical path. When the rate change word length is large, you can limit hardware resource use by fitting the multiplicand data type to the DSP blocks on the FPGA. .

Dependencies

To enable this port, set the **Rate change source** parameter to `Input port`.

Data Types: `fixed point`

`reset` — Clears internal states
scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select **Enable reset input port**.

Data Types: `Boolean`

Output

`data` — Filtered output data
real or complex scalar

Filtered output data, returned as a real or complex scalar. When the input data type is a floating-point data type, the output data inherits the data type of the input data. When the input data type is an integer type or a fixed-point type, the **Output** parameter on the **Data Types** tab controls the output data type.

Data Types: `fixed point` | `single` | `double`

`valid` — Indicates valid output data
scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

`ready` — Indicates block is ready for new input data

scalar

Control signal that indicates that the block is ready for new input data sample on the next cycle. When **ready** is `1` (`true`), you can specify the **data** and **valid** inputs for the next time step. When **ready** is `0` (`false`), the block ignores any input data in the next time step.

Data Types: `Boolean`

Parameters

Main

`Rate change source` — Source of rate change

`Property` (default) | `Input port`

You can enter a constant rate change as a parameter or provide a time-varying rate change by using an input port.

Selecting `Input port` enables the **rate** port on the block.

`Rate change (fsin/fsout)` — Rate change factor

`147/160` (default) | positive real scalar

Specify the rate change factor as a ratio of the input sample rate and the output sample rate, F_{in}/F_{out}, or provide a positive rational value. There are no limits on the rate change factor. Specify the data type for this value by using the **RateChange** parameter on the **Data Types** tab.

Dependencies

To enable this parameter, set **Rate change source** to `Property`.

Data Types: `double`

`Coefficients matrix` — FIR filter coefficients

`[-1/6 1/2 -1/3 0;1/2 -1 -1/2 1;-1/2 1/2 1 0;1/6 0 -1/6 0]` (default) | matrix of real values

Specify FIR filter coefficients as an *M*-by-*N* matrix of real values, where *N* is the number of filters and *M* is the number of coefficients in each filter. *N* must be less than six. The block implements a polynomial of order *N* – 1. The default value is a special closed-form LaGrange solution that accomplishes most rate changes.

Data Types: `double`

`Filter structure` — HDL filter architecture

`Direct form systolic` (default) | `Direct form transposed`

This block implements the FIR filter stages by using the same architectures as the Discrete FIR Filter block. Specify the HDL filter architecture as one of these structures:

- `Direct form systolic` — This architecture provides a fully parallel filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For architecture details, see “Fully Parallel Systolic Architecture”.

- `Direct form transposed` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. For architecture and performance details, see “Fully Parallel Transposed Architecture”.

All implementations share multipliers for symmetric and antisymmetric coefficients and remove multipliers for zero-valued coefficients.

Data Types

`Rounding mode` — Rounding mode for type-casting the output

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Rounding mode for type-casting the output to the data type specified by the **Output** parameter. When the input data type is a floating-point data type, the block ignores this parameter. For more details, see “Rounding Modes”.

`Saturate on integer overflow` — Overflow handling for type-casting the output

`off` (default) | `on`

Overflow handling for type-casting the output to the data type specified by the **Output** parameter. When the input data type is a floating-point data type, the block ignores this parameter. For more details, see “Overflow Handling”.

`Coefficients` — Data type of filter coefficients

`Inherit: Same word length as input` (default) | `<data type expression>`

The block casts the filter coefficients to this data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is a floating-point data type, the block ignores this parameter.

The recommended data type for this parameter is `Inherit: Same word length as input`. When selecting this data type, consider the size supported by the DSP blocks on your target FPGA.

`RateChange` — Data type of rate change factor

`fixdt(1,16)` (default) | `<data type expression>`

The block casts the **Rate change (fsin/fsout)** parameter value to this data type and uses this data type to derive the data type for the internal accumulator. The accumulator data type is `fixdt(1,`*`fractionalWL`*`+1,`*`fractionalWL`*`)`, where *fractionalWL* is the fraction length of the rate change data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is a floating-point data type, the block ignores this parameter.

This data type must have enough integer bits to represent the `fsIn/fsOut` value. If the data type specified does not have enough integer bits, the block returns an error. The default setting does not specify a number of fractional bits, so the block can compute the necessary integer bits. This data type must have at least one integer bit and one fractional bit. The fractional part of this data type determines the accuracy of the phase timing, but also increases the critical path. When the rate change word length is large, you can limit hardware resources by fitting the multiplicand data type to the DSP blocks on the FPGA.

Dependencies

To enable this parameter, set **Rate change source** to `Property`.

`Multiplicand` — Data type of multiplicand

`Inherit: Inherit via internal rule` (default) | `<data type expression>`

The block casts the output of the accumulator to this data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is a floating-point data type, the block ignores this parameter. When the rate change word length is large, you can limit hardware resource use by controlling the multiplicand data type. When selecting this data type, consider the size supported by the DSP blocks on your target FPGA.

`Output` — Data type of filter output
`Inherit: same as first input` (default) | `<data type expression>`

The block casts the output of each filter stage to this data type. The quantization uses the settings of the **Rounding mode** and **Overflow mode** parameters. When the input data type is a floating-point data type, the block ignores this parameter.

Control Ports

`Enable reset input port` — Option to enable reset input port
`off` (default) | `on`

Select this parameter to enable the **reset** input port. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

`Use HDL global reset` — Option to connect data path registers to generated HDL global reset signal
`off` (default) | `on`

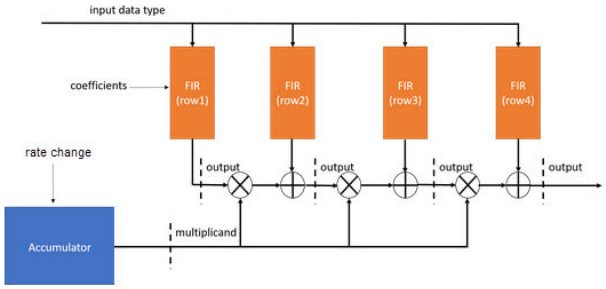
Select this parameter to connect the generated HDL global reset signal to the data path registers. This parameter does not change the appearance of the block or modify simulation behavior in Simulink. When you clear this parameter, the generated HDL global reset clears only the control path registers. 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.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Algorithms

The Farrow Rate Converter block uses Horner’s rule to compute samples from the polynomial. The polynomial is implemented with several FIR filters. Each FIR filter is an instance of the Discrete FIR Filter block. For details of filter architectures and resource optimization, see “FIR Filter Architectures for FPGAs and ASICs”.

This diagram shows the Farrow architecture for a four-stage filter. There are four FIR filters. The output of each filter is added to the result of the previous stage, cast to the output data type, then multiplied by the current value of the accumulator. The accumulator operates on the fractional part of the rate change, and uses a data type derived from the rate change data type. The accumulator data type is `fixdt(1,`*`fractionalWL`*`+1,`*`fractionalWL`*`)`, where *fractionalWL* is the fraction length of the rate change data type.



When you set the coefficient data type to `Same as input wordlength`, each subfilter computes the best precision data type based on its coefficients. As a result, each filter can have a different output data type. To maintain full precision when you set the output data type to `Full precision`, the block casts the output of each subfilter to the highest precision data type of all the subfilter data types.

The table shows the coefficient data type and output data type for each subfilter when using the default Farrow coefficients, and an input data type of `fixdt(1,18,14)`.

Coefficient Value	Coefficient Data Type	Subfilter Output Data Type
`-1/6 1/2 -1/2 1/6`	`fixdt(1,18,17)`	`fixdt(1,36,31)`
`1/2 -1 1/2 0`	`fixdt(1,18,17)`	`fixdt(1,36,31)`
`-1/3 -1/2 1 -1/6`	`fixdt(1,18,16)`	`fixdt(1,35,30)`
`0 1 0 0`	`fixdt(1,18,16)`	`fixdt(1,34,30)`

In this case, the full precision data type is `fixdt(1,36,31)`.

Performance

This table shows post-synthesis resource utilization for the HDL code generated for the default coefficients and rate change settings, with 16-bit input and 16-bit coefficients. The synthesis targets a Xilinx ZC-706 (XC7Z045ffg900-2) FPGA. The **Global HDL reset type** parameter is `Synchronous` and **Minimize clock enables** is selected. The **reset** port is not enabled, so only control path registers are connected to the generated global HDL reset.

Resource	Uses
LUT	394
FF	604
BRAM	0
Xilinx LogiCORE DSP48	13

After place and route, the maximum clock frequency of the design is 495 MHz.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

See Also

Objects

`dsphdl.FarrowRateConverter` | `dsphdl.FIRFilter`

Blocks

FIR Rate Converter | Discrete FIR Filter

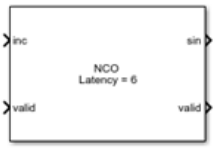
Topics

"Sample Rate Conversion for an LTE Receiver" (Wireless HDL Toolbox)

Introduced in R2022a

NCO

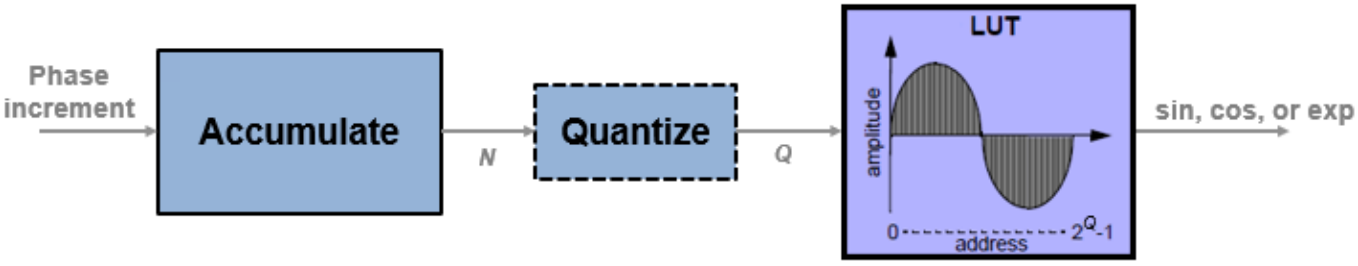
Generate real or complex sinusoidal signals
Library: DSP HDL Toolbox / Signal Operations
DSP HDL Toolbox / Sources



Description

The NCO block generates real or complex sinusoidal signals, while providing hardware-friendly control signals.

A numerically-controlled oscillator (NCO) accumulates a phase increment and uses the quantized output of the accumulator as the index to a lookup table that contains the sine wave values. The wrap around of the fixed-point accumulator and quantizer data types provide periodicity of the sine wave, and quantization reduces the necessary size of the table for a given frequency resolution.



For an example of how to generate a sine wave using the NCO block, see “Generate Sine Wave”. For more information on configuration and implementation, refer to the “Algorithms” on page 1-51 section.

The block also provides these features:

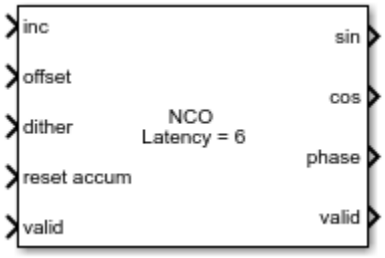
- Optional frame-based output.
- A lookup table compression option to reduce the lookup table size. This compression results in less than one LSB loss in precision. See “Lookup Table Compression” on page 1-52 for more information.
- An optional input port for external dither.
- An optional reset port that resets the phase accumulator to its initial value.
- An optional output port for the current NCO phase.

Ports

Note

- This block appears in the **Sources** libraries with **Phase increment source** parameter set to `Property`. The only input port is **valid**.
- This block appears in the **Signal Operations** libraries with **Phase increment source** parameter set to `Input port`. This configuration shows the optional input port **inc**.

This icon shows the optional ports of the NCO block.



Input

`inc` — Phase increment

scalar integer

Phase increment, specified as a scalar integer. The block casts this value to match the accumulator word length.

`double` and `single` data types are supported for simulation but not for HDL code generation.

Dependencies

To enable this port, set the **Phase increment source** parameter to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt([],N,0)`

`offset` — Phase offset

scalar integer

Phase offset, specified as a scalar integer.

`double` and `single` data types are supported for simulation but not for HDL code generation.

Dependencies

To enable this port, set the **Phase offset source** parameter to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt([],N,0)`

`dither` — Dither

integer | column vector of integers

Dither, specified as an integer or a column vector of integers. The length of the vector must equal the **Samples per frame** parameter value.

`double` and `single` data types are supported for simulation but not for HDL code generation.

Dependencies

To enable this port, set the **Dither source** parameter to `Input port`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt([],N,0)`

`valid` — Control signal that enables NCO operation

scalar

Control signal that enables NCO operation, specified as a Boolean scalar. When this signal is `1`, the block increments the phase and captures any input values. When this signal is `0`, the block holds the phase accumulator and ignores any input values.

When the **Samples per frame** parameter is greater than `1`, this value enables processing of **Samples per frame** samples.

Data Types: `Boolean`

`reset accum` — Resets the accumulator

scalar

Control signal that resets the accumulator, specified as a Boolean scalar. When this signal is `1`, the block resets the accumulator to its initial value. This signal does not reset the output samples in the pipeline.

Dependencies

To enable this port, select the **Enable accumulator reset input port** parameter.

Data Types: `Boolean`

Output

`sin, cos, exp` — Generated waveform

scalar | column vector

Generated waveform, returned as a scalar or as a column vector with length equal to the **Samples per frame** parameter value. The output can be a single port that returns **sin** or **cos** values, a single port that returns **exp** values representing `cosine + j*sine`, or two ports that return **sin** and **cos** values, respectively.

When all input values are fixed-point type or all input ports are disabled, the block determines the output type using the **Output data type** parameter. When any input value is floating-point type, the block ignores the **Output data type** parameter. In this case, the block returns the waveform as floating-point values. Floating-point data types are supported for simulation but not for HDL code generation.

Dependencies

By default, this output port is a sine wave, **sin**. The port label and format changes based on the **Type of output signal** parameter.

`phase` — Current phase of NCO

scalar | column vector

Current phase of NCO, returned as a scalar or as a column vector with length equal to the **Samples per frame** parameter value. The phase is the output of the quantized accumulator with offset and increment applied. If quantization is disabled, this port returns the output of the accumulator with offset and increment applied. The values are of type `fixdt(1,N,0)`, where `N` is the **Number of quantizer accumulator bits** parameter value. If quantization is disabled, then `N` is the **Accumulator Word length** parameter value.

If any input value is floating-point type, the block returns the **phase** as a floating-point value. Floating-point data types are supported for simulation but not for HDL code generation.

Dependencies

To enable this port, select the **Enable phase port** parameter.

Data Types: `single` | `double` | `fixdt(1,N,0)`

`valid` — Indicates validity of output data

scalar

Control signal that indicates validity of output data, returned as a Boolean scalar. When output **valid** is `1`, the values on the **sin**, **cos**, **exp**, and **phase** ports are valid. When output **valid** is `0`, the values on the output ports are not valid.

When the **Samples per frame** parameter is greater than `1`, this signal indicates the validity of all elements in the output vector.

Data Types: `Boolean`

Parameters

Main

Note This block supports `double` and `single` input for simulation but not for HDL code generation. When all input values are fixed-point type or all input ports are disabled, the block determines the output type using the **Output data type** parameter. When any input value is floating-point type, the block ignores the **Output data type** parameter. In this case, the block returns the waveform and optional **phase** as floating-point values.

To use the Fixed-Point Designer™ data type override feature, you can obtain a `double` output value by applying `double` input data to one of the optional ports.

`Phase increment source` — Source of phase increment

`Input port` (default) | `Property`

You can set the phase increment with an input port or by entering a value for the parameter. If you select `Property`, the **Phase increment** parameter appears for you to enter a value. If you select `Input port`, the **inc** port appears on the block.

`Phase increment` — Phase increment for generated waveform

`100` (default) | integer

Phase increment for the generated the waveform, specified as an integer. The block casts this value to match the accumulator word length.

Dependencies

To enable this parameter, set the **Phase increment source** parameter to `Property`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixdt([],N,0)`

`Phase offset source` — Source of phase offset
`Input port` (default) | `Property`

You can set the phase offset with an input port or by entering a value for the parameter. If you select `Property`, the **Phase offset** parameter appears for you to enter a value. If you select `Input port`, the **offset** port appears on the block.

`Phase offset` — Phase offset for generated waveform
`0` (default) | integer

Phase offset for the generated waveform, specified as an integer.

Dependencies

To enable this parameter, set the **Phase offset source** parameter to `Property`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixdt([],N,0)`

`Dither source` — Source of number of dither bits
`Property` (default) | `Input port` | `None`

You can set the dither from an input port or from a parameter. If you select `Property`, the **Number of dither bits** parameter appears. If you select `Input port`, a port appears on the block. If you select `None`, the block does not add dither.

`Number of dither bits` — Bits used to express dither
`4` (default) | positive integer

Number of dither bits, specified as a positive integer.

Dependencies

To enable this parameter, set the **Dither source** parameter to `Property`.

`Samples per frame` — Vector size for frame-based input and output
`1` (default) | positive integer

When you set this value to `1`, the block has scalar input and output. When this value is greater than `1`, the **dither** port expects a column vector of length **Samples per frame** and the **sin**, **cos**, **exp**, and **phase** ports return column vectors of length **Samples per frame**.

`Enable look up table compression method` — Compress the lookup table
`off` (default) | `on`

By default, the block implements a noncompressed lookup table, and the output of this block matches the output of the NCO block. When you enable this option, the block implements a compressed lookup table. The Sunderland compression method reduces the size of the lookup table, losing less than one LSB of precision. The spurious free dynamic range (SFDR) is empirically 1–3 dB lower than

the noncompressed case. The hardware savings of the compressed lookup table allow room to improve performance by increasing the word length of the accumulator and the number of quantize bits. For detail of the compression method, see “Algorithms” on page 1-51.

`Enable accumulator reset input port` — Enable reset control signal
`off` (default) | `on`

Select this parameter to enable the **reset accum** port. When **reset accum** is `1`, the block resets the accumulator to its initial value.

`Type of output signal` — Format of output waveform
`Sine` (default) | `Cosine` | `Complex exponential` | `Sine and cosine`

If you select `Sine` or `Cosine`, the block shows the applicable port, **sin** or **cos**. If you select `Complex exponential`, the output is of the form `cosine + j*sine` and the port is labeled **exp**. If you select `Sine and cosine`, the block shows two ports, **sin** and **cos**.

When you set the **Type of output signal** parameter to `Complex exponential` or `Sine and cosine`, the block implements a 1/8 sine wave lookup table for each of the sine and cosine parts of the waveform, and uses control logic to select and invert the values to generate both sine and cosine waveforms. This optimization means that dual output mode uses similar hardware resources compared to single output mode.

`Enable phase port` — Output current phase
`off` (default) | `on`

Select this parameter to return the current NCO phase on the **phase** port. The phase is the output of the quantized accumulator, with offset and increment applied. If quantization is disabled, this port returns the output of the accumulator, with offset and increment applied.

Data Types

`Rounding Mode` — Rounding mode for fixed-point operations
`Floor` (default)

Rounding mode for fixed-point operations. **Rounding Mode** is a read-only parameter with value `Floor`.

`Overflow mode` — Overflow mode for fixed-point operations
`Wrap` (default)

Overflow mode for fixed-point operations. **Overflow mode** is a read-only parameter. Fixed-point numbers wrap around on overflow.

`Accumulator Data Type` — Accumulator data type
`Binary point scaling` (default)

Accumulator data type description. This parameter is read-only, with value `Binary point scaling`. The block defines the fixed-point data type using the **Accumulator Signed**, **Accumulator Word length**, and **Accumulator Fraction length** parameters.

`Accumulator Signed` — Signed or unsigned accumulator data format
`Signed` (default)

This parameter is read-only. All output is signed format.

`Accumulator Word length` — Accumulator word length
`16` (default) | integer

Units are in bits. This value must include the sign bit.

If you clear the **Quantize phase** parameter, then **Accumulator word length** determines the LUT size. For HDL code generation, the LUT size must be between 2 and 2^{17} entries. When you select **Enable look up table compression method**, this parameter must be an integer in the range [5,21]. When you clear **Enable look up table compression method**, this parameter must be an integer in the range [3,19]. For more information on how this parameter affects the LUT size, see the “Algorithms” on page 1-51 section.

When you select the **Quantize phase** parameter, there is no limit to the **Accumulator word length** parameter value.

`Accumulator Fraction length` — Accumulator fraction length
`0` (default) | integer

This parameter is read-only. The accumulator fraction length is zero bits.

The accumulator operates on integers. If the phase increment is `fixdt` type with a fractional part, the block returns an error.

`Quantize phase` — Quantize accumulated phase
`off` (default) | `on`

When you select **Quantize phase**, the block quantizes the result of the phase accumulator to a fixed bit-width. The block uses this quantized value to select a waveform value from the lookup table. Quantizing the output of the phase accumulator enables you to reduce the lookup table size without lowering the frequency resolution. Select the size of the lookup table by using the **Number of quantizer accumulator bits** parameter.

When you clear **Quantize phase**, the block uses the full accumulator value as the address of the lookup table.

`Number of quantizer accumulator bits` — Number of quantizer accumulator bits
`12` (default) | integer

Number of quantizer accumulator bits, specified as an integer scalar less than the accumulator word length. For HDL code generation, this parameter value must result in a LUT size between 2 and 2^{17} entries. When you select **Enable look up table compression method**, this parameter must be an integer in the range [5,21]. When you clear **Enable look up table compression method**, this parameter must be an integer in the range [3,19]. For more information on how this parameter affects the LUT size, see the “Algorithms” on page 1-51 section.

Dependencies

To enable this parameter, select the **Quantize phase** parameter.

`Output Data Type` — Output data type
`Binary point scaling` (default) | `double` | `single`

Specify the data type for the **sin**, **cos**, and **exp** ports. This parameter is ignored if any input is of floating-point type. In that case, the output data type is floating-point.

If you select `Binary point scaling`, the block defines the fixed-point data type using the **Output Signed**, **Output Word length**, and **Output Fraction length** parameters.

`Output Signed` — Signed or unsigned output data format
`Signed` (default)

This parameter is read-only. All output is signed format.

`Output Word length` — Output word length
16 (default) | integer

Units are in bits. This value must include the sign bit.

`Output Fraction length` — Output fraction length
14 (default) | integer

Units are in bits.

Algorithms

The frequency resolution of the sine wave depends on the size of the accumulator. Given a sample time, T_s, and the desired output frequency resolution Δf, calculate the necessary accumulator word length, N.

$$N = \text{ceil}\left(\log_2\left(\frac{1}{T_s \cdot \Delta f}\right)\right)$$

For a desired output frequency F_o, calculate the *phase increment*.

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

Quantizing the output of the phase accumulator enables you to reduce the lookup table size without lowering the frequency resolution. Calculate the quantized word length to achieve a desired spurious free dynamic range (*SFDR*).

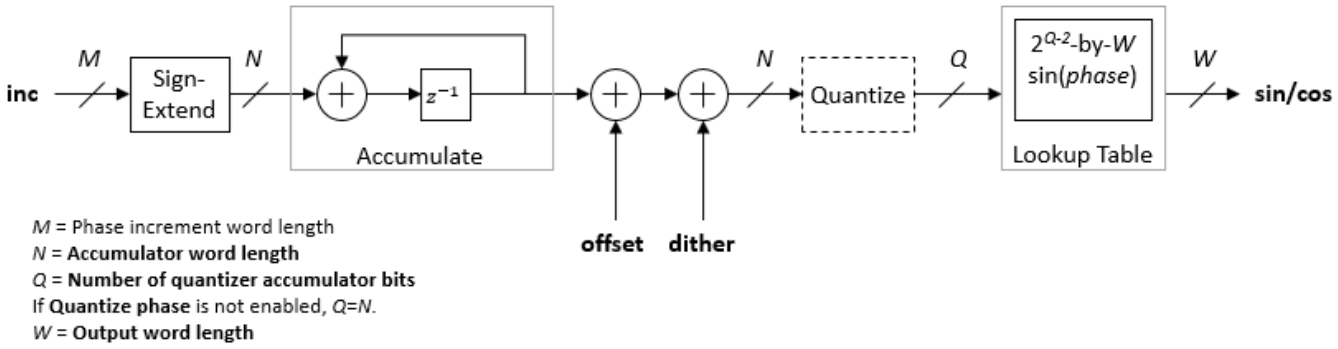
$$Q = \text{ceil}\left(\frac{SFDR - 12}{6}\right)$$

Phase offset and dither are optionally added at the accumulator stage. For a desired phase offset (in radians) of the output waveform, calculate the *phase offset* value that the block adds in the accumulator.

$$phaseoffset = \frac{2^N \cdot desiredphaseoffset}{2\pi}$$

The NCO implementation depends on whether you select **Enable look up table compression method**.

Without lookup table compression, the block uses the same quarter-sine lookup table as the NCO block. The size of the LUT is $2^{Q-2} \times W$ bits, where Q is **Number of quantizer accumulator bits** and W is **Output word length**.



The block casts the phase increment value to match the accumulator word length.

If you do not enable **Quantize phase**, then $Q = N$, where N is **Accumulator Word length**. Consider the impact on simulator memory and hardware resources when you select these parameters.

When you set the **Type of output signal** parameter to `Complex exponential` or `Sine and cosine`, the block implements a 1/8 sine wave lookup table for each of the sine and cosine parts of the waveform, and uses control logic to select and invert the values to generate both sine and cosine waveforms. This optimization means that dual output mode uses similar hardware resources compared to single output mode.

For an example of how to generate a sine wave using the NCO block, see “Generate Sine Wave”.

Lookup Table Compression

When you select lookup table (LUT) compression, the NCO block applies the Sunderland compression method. Sunderland techniques use trigonometric identities to divide each phase of the quarter sine wave into three components and express it as:

$$\sin(A + B + C) = \sin(A + B)\cos(C) + \cos(A)\cos(B)\sin(C) - \sin(A)\sin(B)\sin(C)$$

If the quarter-sine phase has `Q-2` bits, then the phase components A and B have a word length of LA=LB=`ceil((Q-2)/3)`. Phase component C contains the remaining phase bits. If the phase has 12 bits, then the quarter sine phase has 10 bits, and the components are defined as:

- A, the four most significant bits

  $(0 \le A \le \frac{\pi}{2})$
- B, the next four bits

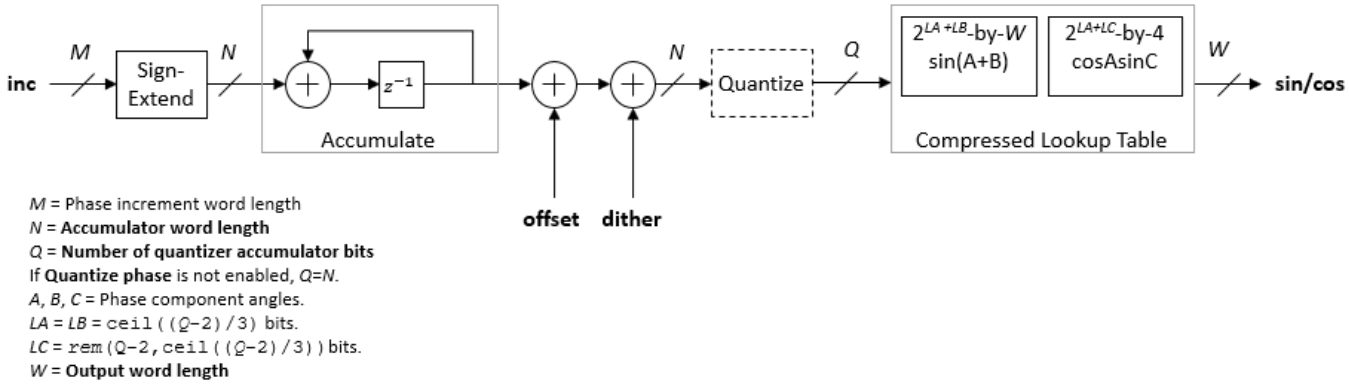
  $(0 \le B \le \frac{\pi}{2} \times 2^{-4})$
- C, the remaining two least significant bits

  $(0 \le C \le \frac{\pi}{2} \times 2^{-8})$

Given the relative sizes of A, B, and C, the equation can be approximated by:

$$\sin(A + B + C) \approx \sin(A + B) + \cos A \sin C$$

The NCO block implements this equation with one LUT for sin($A + B$) and one LUT for cos(A)sin(C). The second term is a fine correction factor that you can truncate to fewer bits without losing precision. Therefore, the second LUT returns a four-bit result.



With the default accumulator size of 16 bits, and the default quantized phase width of 12 bits, the LUTs use 2^8×16 plus 2^6×4 bits (4.5 kb). For comparison, a quarter-sine lookup table without compression uses 2^{10}×16 bits (16 kb). The compression approximation is accurate within one LSB, resulting in an SNR of at least 60 dB on the output. See [1].

When you set the **Type of output** parameter to `Complex exponential` or `Sine and cosine`, the block implements a compressed lookup table for each of the sine and cosine parts of the waveform. The hardware resource use is still smaller than dual output mode with an uncompressed table.

Control Signals

The block has two input control signals, **reset accum** (optional) and **valid**, and one output control signal, **valid**. When **reset accum** is `1`, the block sets the phase accumulator to its initial value. When the input **valid** is `1`, the block increments the phase and captures any input values. When this signal is `0`, the block holds the phase accumulator and ignores any input values. When the output **valid** signal is `1`, the values on the other output ports are valid.

Latency

The latency of the NCO block is six cycles.

Performance

This table shows post-synthesis resource utilization for the HDL code generated for the NCO block in the “Generate Sine Wave” example. The synthesis targets a Xilinx ZC-706 (XC7Z045ffg900-2) FPGA.

Resource	Uses
LUT	744
Slice Reg	156
Xilinx LogiCORE DSP48	0

After place and route, the maximum clock frequency of the design is 477 MHz.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this block was named NCO HDL Optimized, and was included in the DSP System Toolbox **DSP System Toolbox HDL Support** library.

Resource optimization for dual output mode

When you set the **Type of output signal** parameter to `Complex exponential` or `Sine and cosine`, this block implements a 1/8 sine wave lookup table for each of the sine and cosine parts of the waveform, and uses control logic to select and invert the values to generate both sine and cosine waveforms. This optimization means that dual output mode uses similar hardware resources compared to single output mode. In previous releases, the block implemented one lookup table for each output waveform.

HDL-optimized NCO requires valid input port
Behavior changed in R2020a

In previous releases, the input **validIn** port of the NCO HDL Optimized block was optional. It is now required, and renamed **valid**. If you are using no other input ports, the block uses the **valid** signal as an enable signal.

HDL-optimized NCO with floating-point inputs applies phase quantization
Behavior changed in R2020a

The output waveform returned from floating-point input values has changed. The output waveform now matches that returned from the same input values specified in fixed-point types.

Prior to R2020a, when using floating-point input types, the NCO HDL Optimized block did not quantize the phase internally. The block expected floating-point phase increment and phase offset inputs specified in radians. Now, the block quantizes the phase internally, and you must specify the input phase increment and offset in terms of the quantized size, for both floating-point and fixed-point input types.

For example, prior to R2020a, for a floating-point HDL NCO to generate output samples with a desired output frequency of F_0 and sample frequency of F_s, you had to specify the phase increment as $2\pi(F_0/F_s)$ and phase offset as $\pi/2$.

Starting in R2020a, you must specify the phase increment and phase offset in terms of the quantized size, N. These input values are the same as the input values you use with fixed-point types. Specify the phase increment as $(F_0 \times 2^N)/F_s$, and the phase offset as $(\pi/2) \times 2^N/2\pi$, or $2^N/4$.

NCO HDL Optimized block now ignores LUTRegisterResetType parameter
Behavior changed in R2020a

In previous releases, you could choose from two options for the **LUTRegisterResetType** parameter on the **HDL Block Properties** dialog of the NCO HDL Optimized block. The two options were `default`, and `none`. Starting in R2020a, the block ignores the parameter setting and uses `none` for this parameter value. This option does not connect a reset signal to the LUT registers. This configuration enables the synthesis tool to determine whether to implement the lookup tables with LUTs or BRAM.

References

[1] Cordesses, L., "Direct Digital Synthesis: A Tool for Periodic Wave Generation (Part 1)." *IEEE Signal Processing Magazine*. Volume 21, Issue 4, July 2004, pp. 50–54.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

- When you set **Dither source** to `Property`, the block adds random dither every cycle. If you generate a validation model with these settings, a warning is displayed. Random generation of the internal dither can cause mismatches between the models. You can increase the error margin for the validation comparison to account for the difference. You can also disable dither or set **Dither source** to `Input port` to avoid this issue.
- You cannot use the NCO block inside a Resettable Synchronous Subsystem.

See Also

Blocks

NCO

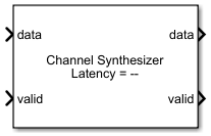
Objects
`dsphdl.NCO`

Topics
“HDL QAM Transmitter and Receiver” (Communications Toolbox)

Introduced in R2013a

Channel Synthesizer

Combine narrowband signals into multichannel signal
Library: DSP HDL Toolbox / Filtering



Description

The Channel Synthesizer block combines narrowband signals into a multi-channel signal using the polyphase filter bank technique.

The block accepts a real- or complex-valued row-vector input data and control signals, and outputs synthesized column-vector and a control signal. You can achieve gigasamples-per-second (GSPS) throughput by using this block. The block implements a polyphase filter, with one subfilter per input vector element. The block supports HDL code generation and hardware deployment.

Ports

Input

`data` — Input data

real-valued row vector | complex-valued row vector

Input data, specified as a real- or complex-valued row vector.

The vector length must be a power of 2 and in the range [4, 64].

You can use `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `fixed point` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`valid` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

`reset` — Clears internal states

scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Synthesized output data

complex-valued column vector

Synthesized output data, returned as a complex-valued column vector.

When the input data type is a floating-point type, the output data inherits the data type of the input data. When the input data type is an integer type or a fixed-point type, the **Output** parameter on the **Data Types** tab controls the output data type.

The output size is same as the input size and is equal to the number of frequency bands or IFFT length. The output order is bit natural. The output data type depends on the IFFT bit growth, required to avoid overflow, and the data type set in the **Output** parameter.

Data Types: `fixed point` | `single` | `double`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

Parameters

Main

`Filter coefficients` — Polyphase filter coefficients

`[-0.0329 0.1218 0.3183 0.4829 0.5469 0.4829 0.5469 0.4829 0.3183 0.1218 -0.0329]` (default) | complex-valued vector

Polyphase filter coefficients, specified as a vector of numeric values. If the number of coefficients is not a multiple of the number of frequency bands or the IFFT length, the block pads this vector with zeros. The default filter specification is a raised-cosine FIR filter, `rcosdesign(0.25,2,4,'sqrt')`. You can specify a vector of coefficients or a call to a filter design function that returns the coefficient values. By default, the block casts the coefficients to the same word length as the input.

`Filter structure` — HDL filter architecture

`Direct form transposed` (default) | `Direct form systolic`

Specify the HDL filter architecture as one of these values:

- `Direct form transposed` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. For architecture and performance details, see “Fully Parallel Transposed Architecture”.

- `Direct form systolic` — This architecture provides a fully parallel filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For architecture and performance details, see "Fully Parallel Systolic Architecture".

All implementations share multipliers for symmetric and antisymmetric coefficients and remove multipliers for zero-valued coefficients.

`Complex multiplication` — HDL implementation of complex multipliers
`Use 4 multipliers and 2 adders` (default) | `Use 3 multipliers and 5 adders`

Specify the HDL implementation of complex multipliers as either `Use 4 multipliers and 2 adders` or `Use 3 multipliers and 5 adders`. The speed of the multipliers depends on your synthesis tool and target device.

`Divide butterfly outputs by two` — IFFT scaling
`on` (default) | `off`

When you select this parameter, the IFFT implements an overall 1/*N* scale factor by scaling the result of each pipeline stage by 2, where *N* is the IFFT length. This adjustment keeps the output of the IFFT in the same amplitude range as its input. If you disable scaling, the IFFT avoids overflow by increasing the word length by one bit at each stage.

Data Types

`Rounding mode` — Rounding mode for typecasting output
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Use fixed-point arithmetic for internal calculations when the input is an integer or fixed-point data type. This option does not apply when the input is `single` or `double`. For more details, see "Rounding Modes".

`Saturate on integer overflow` — Overflow handling for typecasting output
`off` (default) | `on`

Use fixed-point arithmetic for internal calculations when the input is an integer or fixed-point data type. This option does not apply when the input is `single` or `double`. Cast the coefficients and output of the polyphase filter to the data types you specify. For more information, see "Overflow Handling".

`Coefficients` — Data type of the coefficients
`Inherit: Same word length as input` (default) | `<data type expression>`

Cast the polyphase filter coefficients to this data type using the rounding and overflow settings you specify. When you select `Inherit: Same word length as input` (default), the block selects the binary point using `fixed point` best-precision rules.

`Output` — Data type of block output
`Inherit: via internal rule` (default) | `Inherit: Same as input` | `<data type expression>`

When you select `Inherit: via internal rule`, the block selects a best-precision binary point by considering filter coefficients values and the input data type range. When you select `Inherit: Same as input`, the block casts the output of the polyphase filter to the input data type, using the rounding and overflow settings you specify.

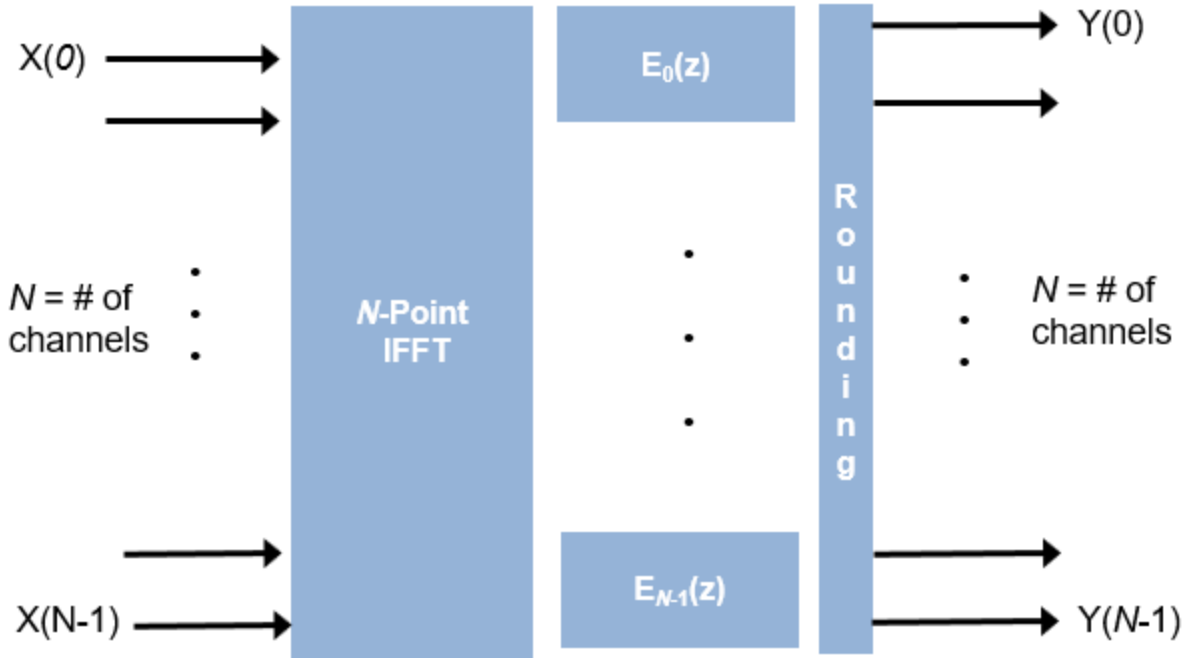
Control Ports

`Enable reset input port` — Option to enable reset input port
`off` (default) | `on`

Select this parameter to enable the **reset** input port. The reset signal implements a local synchronous reset of the data path registers.

Algorithms

The polyphase filter algorithm requires a subfilter for each FFT channel. For more information about the polyphase filter architecture, see the Channelizer (DSP System Toolbox) block reference page.

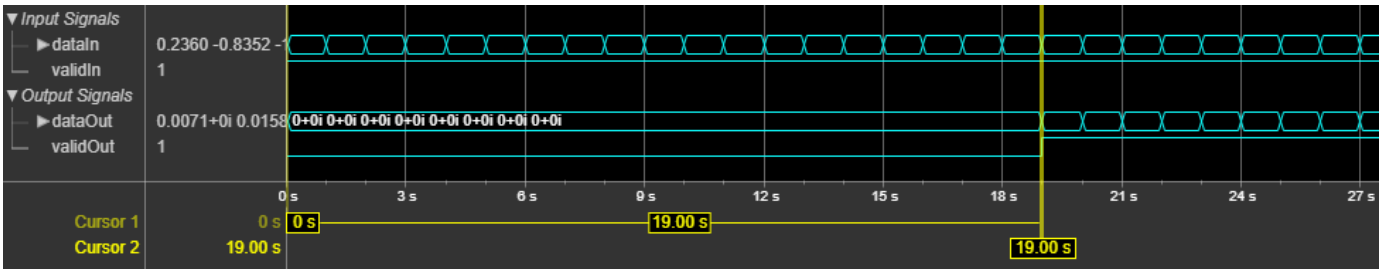


If the FFT length is N, the block implements N subfilters in the hardware. Each subfilter is an FIR filter direct form transposed or direct form systolic with Num_{Coeffs}/N taps. The block casts the output of the subfilters to the specified **Output** data type by using the rounding and overflow settings you select and then pipelines filter tap in the subfilter to target the DSP sections of an FPGA.

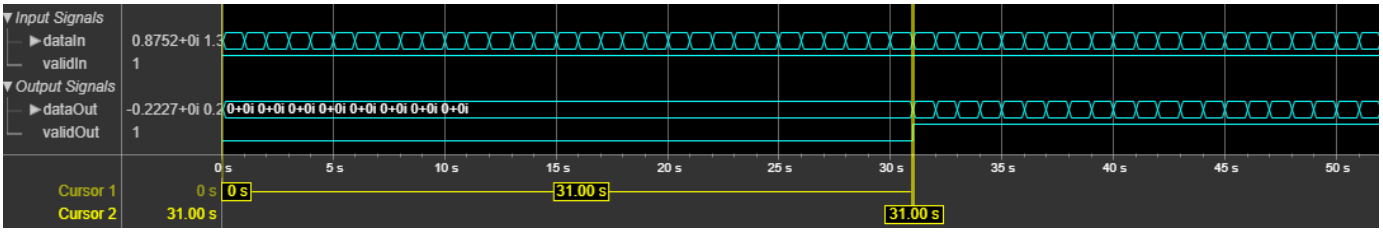
Latency

The latency varies with the input size and filter structure. After you update the model, the block displays the latency on the block icon. The displayed latency is the number of cycles between the first valid input and the first valid output, assuming that the input is continuous. The filter coefficients and complex multiplication do not affect the latency.

This figure shows the output of the block for a vector input length 8 when you set the **Filter structure** parameter to `Direct form transposed` and all other parameters to their default values. The latency of the block is 19 clock cycles.



This figure shows the output of the block for a vector input of length 8 when you set the **Filter structure** parameter to `Direct form systolic` and all other parameters to their default values. The latency of the block is 31 clock cycles.



Performance

These resource and performance data are the place-and-route results from the generated HDL targeted to the Xilinx Zynq- 7000 ZC706 evaluation board. The two examples in the tables use this common configuration:

- 1-by-8 vector
- 16-bit complex input data
- Filter structure — Direct form transposed
- Filter length — 96 coefficients
- Coefficient data type — Same word length as input
- Output data type — Same as input
- Complex multiplication (default) — Use 4 multipliers and 2 adders
- Output scaling — Enabled

The performance of the synthesized HDL code varies with your target and synthesis options.

When you set the **Filter structure** parameter to `Direct form transposed`, the block achieves a clock frequency of 382 MHz. The design uses these resources.

Resource	Number Used
LUT	1953
FFS	3833
Xilinx LogiCORE® DSP48	208

When you set the **Filter structure** to `Direct form systolic`, the block achieves a clock frequency of 381 MHz. The design uses these resources.

Resource	Number Used
LUT	2026
FFS	3519
Xilinx LogiCORE DSP48	208

References

[1] Harris, Fredric J. *Multirate Signal Processing for Communication Systems*. Upper Saddle River, N.J: Prentice Hall PTR, 2004.

[2] Harris, Frederic J., Chris Dick, and Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." *IEEE® Transactions on Microwave Theory and Techniques*. 51, no 4, (April 2003): 1395–1412. https://doi.org/10.1109/TMTT.2003.809176.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

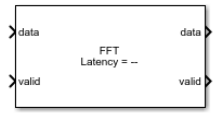
Blocks
Channelizer

Objects
`dsphdl.ChannelSynthesizer`

Introduced in R2022a

FFT

Compute fast Fourier transform (FFT)
Library: DSP HDL Toolbox / Transforms



Description

The FFT block provides two architectures that implement the algorithm for FPGA and ASIC applications. You can select an architecture that optimizes for either throughput or area.

- `Streaming Radix 2^2` — Use this architecture for high-throughput applications. This architecture supports scalar or vector input data. You can achieve gigasamples-per-second (GSPS) throughput using vector input.
- `Burst Radix 2` — Use this architecture for a minimum resource implementation, especially with large fast Fourier transform (FFT) sizes. Your system must be able to tolerate bursty data and higher latency. This architecture supports only scalar input data.

The FFT block accepts real or complex data, provides hardware-friendly control signals, and optional output frame control signals.

Ports

Input

`data` — Input data

scalar or column vector of real or complex values

Input data, specified as a scalar or column vector of real or complex values. Only the `Streaming Radix 2^2` architecture supports a vector input. The vector size must be a power of 2, in the range from 1 to 64, and less than or equal to FFT length.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

`valid` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

`reset` — Clears internal states

scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Frequency channel output data

scalar or column vector of real or complex values

When input is fixed-point data type and scaling is enabled, the output data type is the same as the input data type. When the input is integer type and scaling is enabled, the output is fixed-point type with the same word length as the input integer. The output order is bit-reversed by default. If scaling is disabled, the output word length increases to avoid overflow. Only the `Streaming Radix 2^2` architecture supports vector input and output. For more information, see the **Divide butterfly outputs by two** parameter.

Data Types: `double` | `single` | `fixed point`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

`ready` — Indicates block is ready for new input data

scalar

Control signal that indicates that the block is ready for new input data sample on the next cycle. When **ready** is `1` (`true`), you can specify the **data** and **valid** inputs for the next time step. When **ready** is `0` (`false`), the block ignores any input data in the next time step.

For a waveform that shows this protocol, see the third diagram in the Timing Diagram section.

Dependencies

To enable this port, set the **Architecture** parameter to `Burst Radix 2`.

Data Types: `Boolean`

`start` — Indicates first valid cycle of output frame

scalar

Control signal that indicates the first valid cycle of the output frame. When **start** is `1` (`true`), the block returns the first valid sample of the frame on the output **data** port.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable start output port** parameter.

Data Types: `Boolean`

end — Indicates last valid cycle of output frame

scalar

Control signal that indicates the last valid cycle of the output frame. When **end** is `1` (`true`), the block returns the last valid sample of the frame on the output **data** port.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable end output port** parameter.

Data Types: `Boolean`

Parameters

Main

FFT length — Number of data points for one FFT calculation

`1024` (default)

This parameter specifies the number of data points used for one FFT calculation. For HDL code generation, the FFT length must be a power of 2 between 2^2 to 2^{16}.

Architecture — Architecture type

`Streaming Radix 2^2` (default) | `Burst Radix 2`

This parameter specifies the type of architecture.

- `Streaming Radix 2^2` — Select this value to specify low-latency architecture. This architecture type supports GSPS throughput when using vector input.
- `Burst Radix 2` — Select this value to specify minimum resource architecture. This architecture type does not support vector input. When you use this architecture, your input data must comply with the **ready** backpressure signal.

For more details about these architectures, see “Algorithms” on page 1-68.

Complex multiplication — HDL implementation

`Use 4 multipliers and 2 adders` (default) | `Use 3 multipliers and 5 adders`

This parameter specifies the complex multiplier type for HDL implementation. Each multiplication is implemented either with `Use 4 multipliers and 2 adders` or with `Use 3 multipliers and 5 adders`. The implementation speed depends on the synthesis tool and target device that you use.

Output in bit-reversed order — Order of output data

`on` (default) | `off`

This parameter returns output elements in bit-reversed order.

When you select this parameter, the output elements are bit-reversed. To return output elements in linear order, clear this parameter.

The FFT algorithm calculates output in the reverse order to the input. If you specify the output to be in the same order as the input, the algorithm performs an extra reversal operation. For more information, see “Linear and Bit-Reversed Output Order”.

`Input in bit-reversed order` — Expected order of input data
`off` (default) | `on`

When you select this parameter, the block expects input data in bit-reversed order. By default, this parameter is disabled, and the block expects the input in linear order.

The FFT algorithm calculates output in the reverse order to the input. If you specify the output to be in the same order as the input, the algorithm performs an extra reversal operation. For more information, see “Linear and Bit-Reversed Output Order”.

`Divide butterfly outputs by two` — FFT scaling
`off` (default) | `on`

When you select this parameter, the FFT implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps the output of the FFT in the same amplitude range as its input. If you disable scaling, the FFT avoids overflow by increasing the word length by 1 bit after each butterfly multiplication. The bit increase is the same for both architectures.

Data Types

`Rounding mode` — Rounding mode for internal fixed-point calculations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

This parameter specifies the type of rounding mode for internal fixed-point calculations. For more information about rounding modes, see “Rounding Modes”. When the input is any integer or fixed-point data type, this block uses fixed-point arithmetic for internal calculations. This parameter does not apply when the input data is `single` or `double`. Rounding applies to twiddle-factor multiplication and scaling operations.

Control Ports

`Enable reset input port` — Optional reset signal
`off` (default) | `on`

This parameter enables a reset input port. When you select this parameter, the input **reset** port appears on the block icon.

`Enable start output port` — Optional control signal indicating start of data
`off` (default) | `on`

This parameter enables a port that indicates the start of output data. When you select this parameter, the output **start** port appears on the block icon.

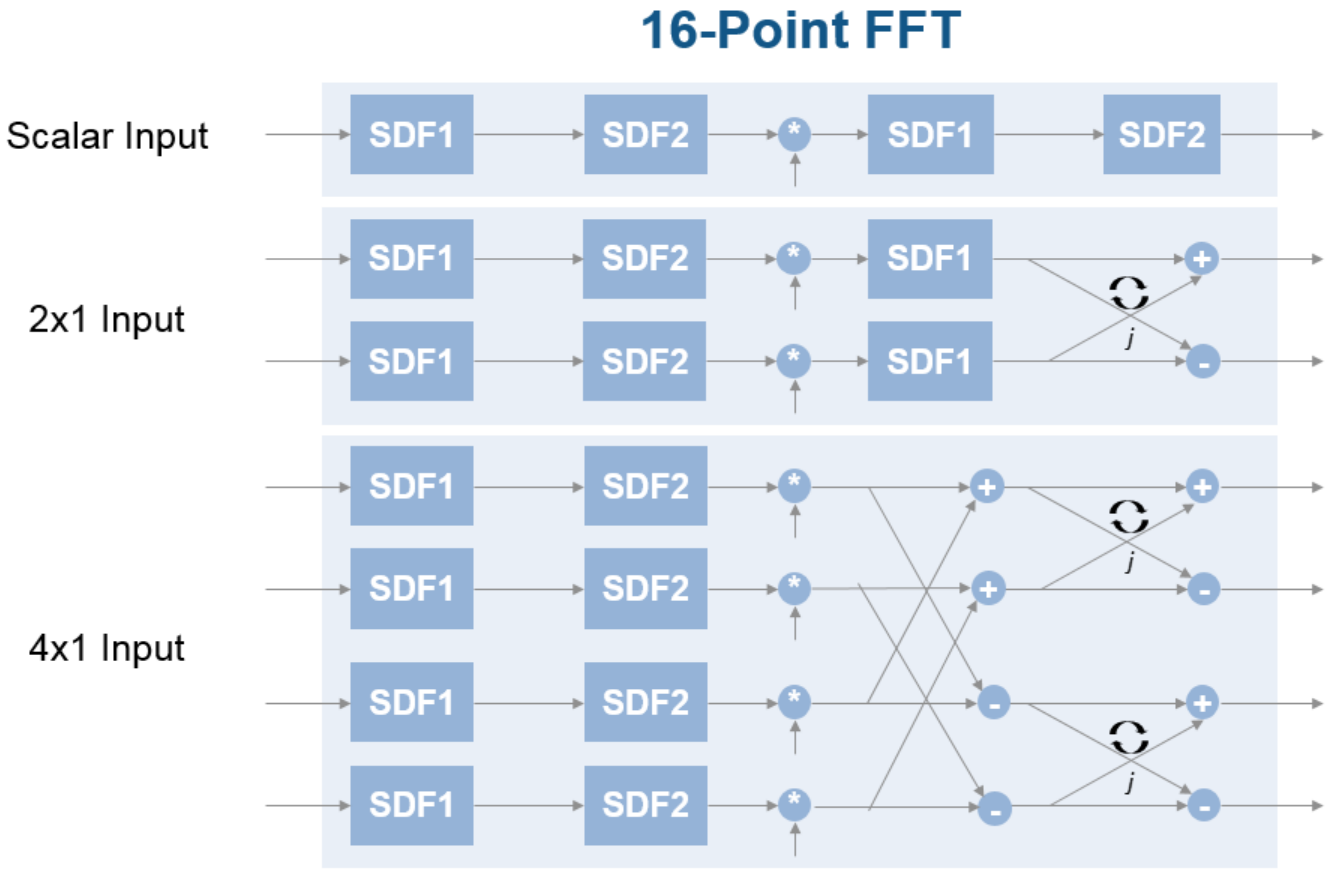
`Enable end output port` — Optional control signal indicating end of data
`off` (default) | `on`

This parameter enables a port that indicates the end of output data. When you select this parameter, the output **end** port appears on the block icon.

Algorithms

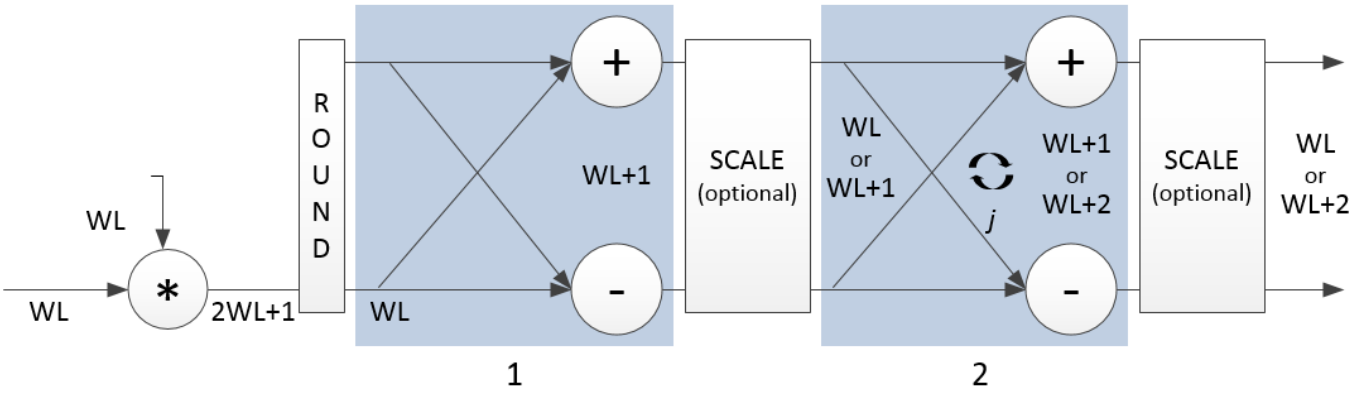
Streaming Radix 2^2

The streaming Radix 2^2 architecture implements a low-latency architecture. It saves resources compared to a streaming Radix 2 implementation by factoring and grouping the FFT equation. The architecture has $\log_4(N)$ stages. Each stage contains two single-path delay feedback (SDF) butterflies with memory controllers. When you use vector input, each stage operates on fewer input samples, so some stages reduce to a simple butterfly, without SDF.



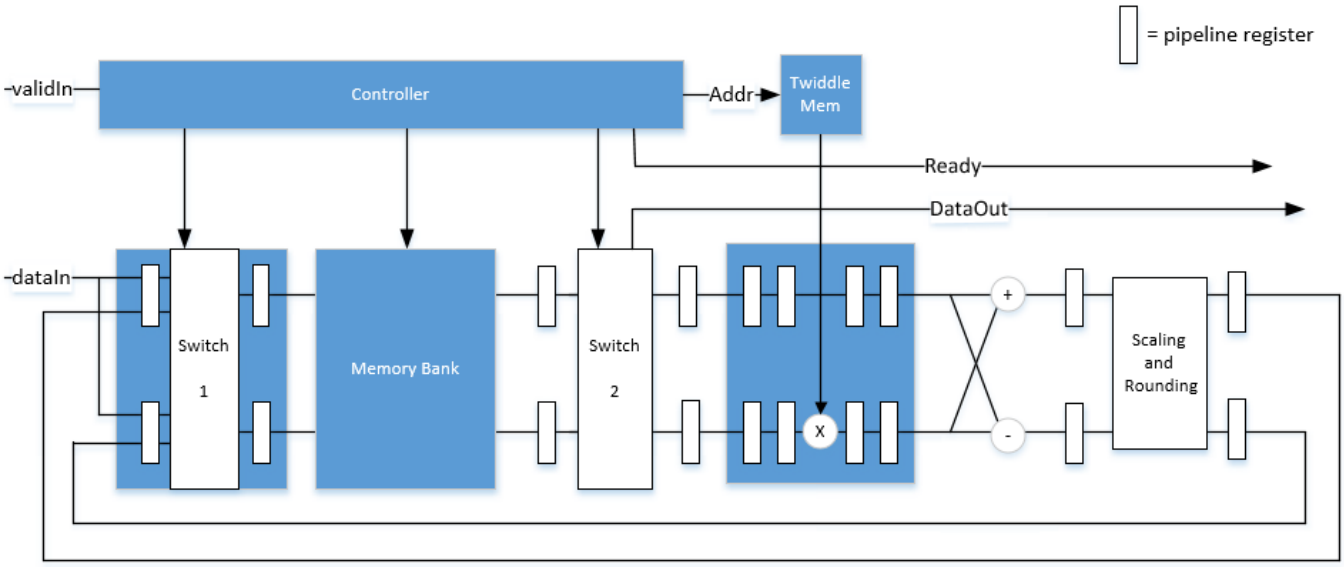
The first SDF stage is a regular butterfly. The second stage multiplies the outputs of the first stage by –*j*. To avoid a hardware multiplier, the block swaps the real and imaginary parts of the inputs, and again swaps the imaginary parts of the resulting outputs. Each stage rounds the result of the twiddle factor multiplication to the input word length. The twiddle factors have two integer bits, and the rest of the bits are used for fractional bits. The twiddle factors have the same bit width as the input data, *WL*. The twiddle factors have two integer bits, and *WL*-2 fractional bits.

If you enable scaling, the algorithm divides the result of each butterfly stage by 2. Scaling at each stage avoids overflow, keeps the word length the same as the input, and results in an overall scale factor of 1/*N*. If scaling is disabled, the algorithm avoids overflow by increasing the word length by 1 bit at each stage. The diagram shows the butterflies and internal word lengths of each stage, not including the memory.



Burst Radix 2

The burst Radix 2 architecture implements the FFT by using a single complex butterfly multiplier. The algorithm cannot start until it has stored the entire input frame, and it cannot accept the next frame until computations are complete. The output **ready** port indicates when the algorithm is ready for new data. The diagram shows the burst architecture, with pipeline registers.



When you use this architecture, your input data must comply with the **ready** backpressure signal.

Control Signals

The algorithm processes input data only when the input **valid** port is 1. Output data is valid only when the output **valid** port is 1.

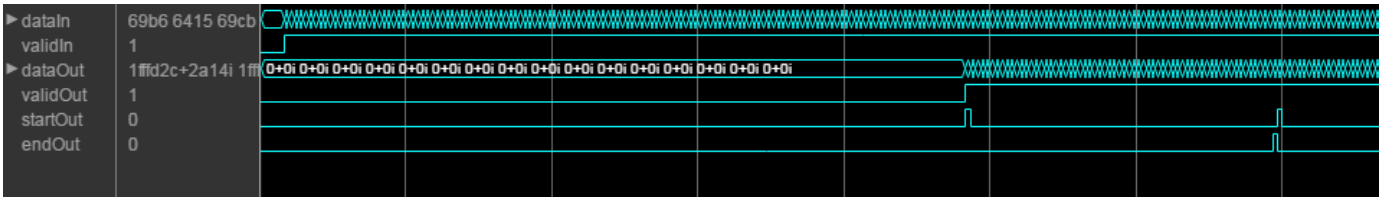
When the optional input **reset** port is 1, the algorithm stops the current calculation and clears all internal states. The algorithm begins new calculations when **reset** port is 0 and the input **valid** port starts a new frame.

Timing Diagram

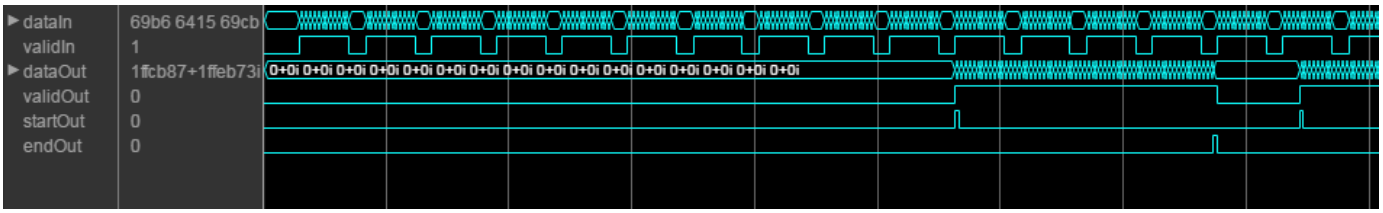
This diagram shows the input and output **valid** port values for contiguous scalar input data, streaming Radix 2^2 architecture, an FFT length of 1024, and a vector size of 16.

The diagram also shows the optional **start** and **end** port values that indicate frame boundaries. If you enable the **start** port, the **start** port value pulses for one cycle with the first valid output of the frame. If you enable the **end** port, the **start** port value pulses for one cycle with the last valid output of the frame.

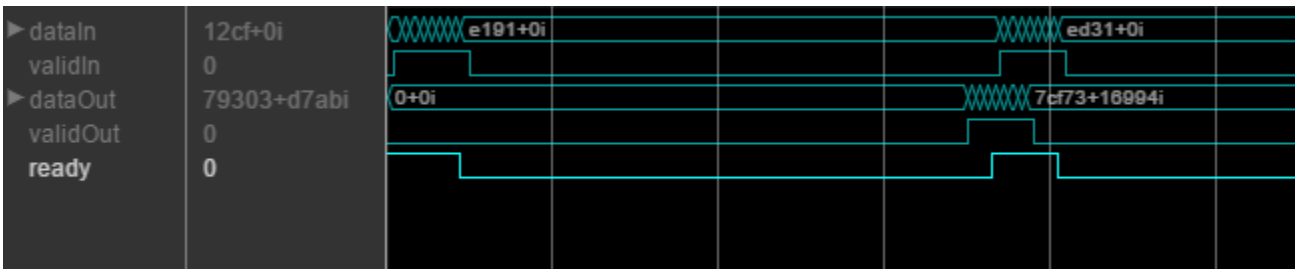
If you apply continuous input frames, the output will also be continuous after the initial latency.



The input **valid** port can be noncontiguous. Data accompanied by an input **valid** port is processed as it arrives, and the resulting data is stored until a frame is filled. Then the algorithm returns contiguous output samples in a frame of *N* (**FFT length**) cycles. This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16.



When you use the burst architecture, 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. You must apply input **data** and **valid** signals only when **ready** is `1` (true). The algorithm ignores any input **data** and **valid** signals when **ready** is `0` (false).

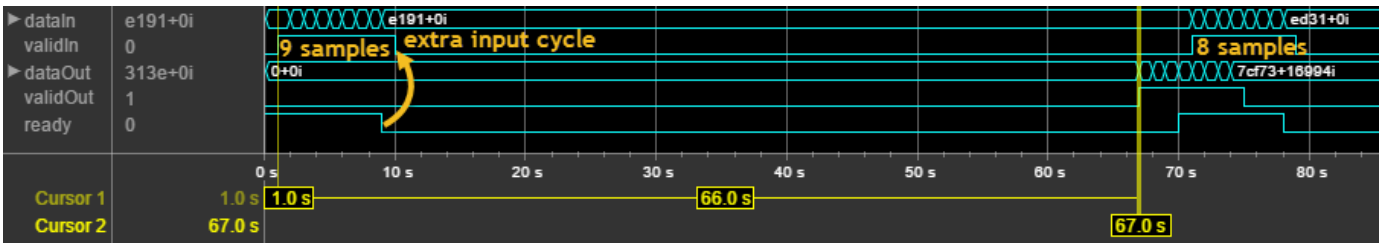


Latency

The latency varies with the **FFT length** and input vector size. After you update the model, the block icon displays the latency. The displayed latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. To obtain this latency programmatically, see “Automatic Delay Matching for the Latency of FFT Block”.

When using the burst architecture with a contiguous input, if your design waits for **ready** to output `0` before de-asserting the input **valid**, then one extra cycle of data arrives at the input. This data sample

is the first sample of the next frame. The algorithm can save one sample while processing the current frame. Due to this one sample advance, the observed latency of the later frames (from input **valid** to output **valid**) is one cycle shorter than the reported latency. The latency is measured from the first cycle, when input **valid** is 1 to the first cycle when output **valid** is 1. The number of cycles between when **ready** port is 0 and the output **valid** port is 1 is always *latency* – *FFTLength*.



Performance

This resource and performance data is the synthesis result from the generated HDL targeted to a Xilinx Virtex®-6 (XC6VLX75T-1FF484) FPGA. The examples in the tables have this configuration:

- 1024 FFT length (default)
- Complex multiplication using 4 multipliers, 2 adders
- Output scaling enabled
- Natural order input, Bit-reversed output
- 16-bit complex input data
- Clock enables minimized (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options. For instance, reordering for a natural-order output uses more RAM than the default bit-reversed output, and real input uses less RAM than complex input.

For a scalar input Radix 2^2 configuration, the design achieves 326 MHz clock frequency. The latency is 1116 cycles. The design uses these resources.

Resource	Number Used
LUT	4597
FFS	5353
Xilinx LogiCORE DSP48	12
Block RAM (16K)	6

When you vectorize the same Radix 2^2 implementation to process two 16-bit input samples in parallel, the design achieves 316 MHz clock frequency. The latency is 600 cycles. The design uses these resources.

Resource	Number Used
LUT	7653
FFS	9322
Xilinx LogiCORE DSP48	24

Resource	Number Used
Block RAM (16K)	8

The block supports scalar input data only when implementing burst Radix 2 architecture. The burst design achieves 309 MHz clock frequency. The latency is 5811 cycles. The design uses these resources.

Resource	Number Used
LUT	971
FFS	1254
Xilinx LogiCORE DSP48	3
Block RAM (16K)	6

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox

Behavior changed in R2022a

Before R2022a, this block was named FFT HDL Optimized, and was included in the DSP System Toolbox **DSP System Toolbox HDL Support** library.

FFT length of 4

Behavior changed in R2022a

This block now supports an FFT length of 4. In previous releases the FFT length had to be a power of 2 from 2^3 to 2^{16}.

References

[1] Algnabi, Y.S, F.A. Aldaamee, R. Teymourzadeh, M. Othman, and M.S. Islam. “Novel architecture of pipeline Radix 2^2 SDF FFT Based on digit-slicing technique.” *10th IEEE International Conference on Semiconductor Electronics (ICSE)*. 2012, pp. 470–474.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block supports HDL code generation using HDL Coder. HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

- You cannot generate HDL code for this block inside an Enabled Subsystem.

See Also

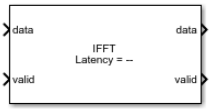
Blocks
IFFT | Channelizer

Objects
`dsphdl.FFT` | `dsphdl.IFFT`

Introduced in R2014a

IFFT

Compute inverse fast Fourier transform (IFFT)
Library: DSP HDL Toolbox / Transforms



Description

The IFFT block provides two architectures that implement the algorithm for FPGA and ASIC applications. You can select an architecture that optimizes for either throughput or area.

- `Streaming Radix 2^2` — Use this architecture for high-throughput applications. This architecture supports scalar or vector input data. You can achieve gigasamples-per-second (GSPS) throughput using vector input.
- `Burst Radix 2` — Use this architecture for a minimum resource implementation, especially with large fast-Fourier-transform (FFT) sizes. Your system must be able to tolerate bursty data and higher latency. This architecture supports only scalar input data.

The IFFT accepts real or complex data, provides hardware-friendly control signals, and optional output frame control signals.

Ports

Input

`data` — Input data

scalar or column vector of real or complex values

Input data, specified as a scalar or column vector of real or complex values. Only the `Streaming Radix 2^2` architecture supports a vector input. 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**.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

`valid` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

When you set the **Architecture** parameter to `Burst Radix 2`, you must apply input **data** and **valid** signals only when **ready** is `1` (true). The block ignores the input **data** and **valid** signals when **ready** is `0` (false).

Data Types: `Boolean`

`reset` — Clears internal states

scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Frequency channel output data

scalar or column vector of real or complex values

When input is fixed-point data type and scaling is enabled, the output data type is the same as the input data type. When the input is integer type and scaling is enabled, the output is fixed-point type with the same word length as the input integer. The output order is bit-reversed by default. If scaling is disabled, the output word length increases to avoid overflow. Only the `Streaming Radix 2^2` architecture supports vector input and output. For more information, see **Divide butterfly outputs by two** parameter.

Data Types: `fixed point` | `double` | `single`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

`ready` — Indicates block is ready for new input data

scalar

Control signal that indicates that the block is ready for new input data sample on the next cycle. When **ready** is `1` (`true`), you can specify the **data** and **valid** inputs for the next time step. When **ready** is `0` (`false`), the block ignores any input data in the next time step.

For a waveform that shows this protocol, see the third diagram in the Timing Diagram section.

Dependencies

To enable this port, set the **Architecture** parameter to `Burst Radix 2`.

Data Types: `Boolean`

`start` — Indicates first valid cycle of output frame

scalar

Control signal that indicates the first valid cycle of the output frame. When **start** is `1` (`true`), the block returns the first valid sample of the frame on the output **data** port.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable start output port** parameter.

Data Types: `Boolean`

end — Indicates last valid cycle of output frame

scalar

Control signal that indicates the last valid cycle of the output frame. When **start** is `1` (`true`), the block returns the last valid sample of the frame on the output **data** port.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable end output port** parameter.

Data Types: `Boolean`

Parameters

Main

`FFT length` — Number of data points used for one FFT calculation

`1024` (default)

This parameter specifies the number of data points used for one inverse-fast-Fourier-transform (IFFT) calculation. For HDL code generation, the FFT length must be a power of 2 between 2^2 and 2^{16}.

`Architecture` — Architecture type

`Streaming Radix 2^2` (default) | `Burst Radix 2`

This parameter specifies the type of architecture.

- `Streaming Radix 2^2` — Select this value to specify low-latency architecture. This architecture type supports GSPS throughput when using vector input.
- `Burst Radix 2` — Select this value to specify minimum resource architecture. This architecture type does not support vector input. When you use this architecture, your input data must comply with the **ready** backpressure signal.

For HDL code generation, the FFT length must be a power of 2 between 2^2 and 2^{16}.

For more details about these architectures, see “Algorithms” on page 1-78.

`Complex Multiplication` — HDL implementation

`Use 4 multipliers and 2 adders` (default) | `Use 3 multipliers and 5 adders`

This parameter specifies the complex multiplier type for HDL implementation. Each multiplication is implemented either with `Use 4 multipliers and 2 adders` or with `Use 3 multipliers and 5 adders`. The implementation speed depends on the synthesis tool and target device that you use.

`Output in bit-reversed order` — Order of output data

on (default) | off

This parameter returns output elements in bit-reversed order.

When you select this parameter, the output elements are bit-reversed. To return output elements in linear order, clear this parameter.

The IFFT algorithm calculates output in the reverse order to the input. If you specify the output to be in the same order as the input, the algorithm performs an extra reversal operation. For more information, see “Linear and Bit-Reversed Output Order”.

`Input in bit-reversed order` — Expected order of input data
off (default) | on

When you select this parameter, the block expects input data in bit-reversed order. By default, the check box is cleared and the input is expected in linear order.

The IFFT algorithm calculates output in the reverse order to the input. If you specify the output to be in the same order as the input, the algorithm performs an extra reversal operation. For more information, see “Linear and Bit-Reversed Output Order”.

`Divide butterfly outputs by two` — FFT scaling
on (default) | off

When you select this parameter, the block implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by two. This adjustment keeps the output of the IFFT in the same amplitude range as its input. If you disable scaling, the block avoids overflow by increasing the word length by 1 bit after each butterfly multiplication. The bit increase is the same for both architectures.

Data Types

`Rounding Method` — Rounding mode for internal fixed-point calculations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

This parameter allows you to select the type of rounding mode for internal fixed-point calculations. For more information about rounding modes, see “Rounding Modes”. When the input is any integer or fixed-point data type, the IFFT algorithm uses fixed-point arithmetic for internal calculations. This option does not apply when the input is `single` or `double` type. Rounding applies to twiddle factor multiplication and scaling operations.

Control Ports

`Enable reset input port` — Optional reset signal
off (default) | on

This parameter enables a reset input port. When you select this parameter, the input **reset** port appears on the block icon.

`Enable start output port` — Optional control signal indicating start of data
off (default) | on

This parameter enables a port that indicates the start of output data. When you select this parameter, the output **start** port appears on the block icon.

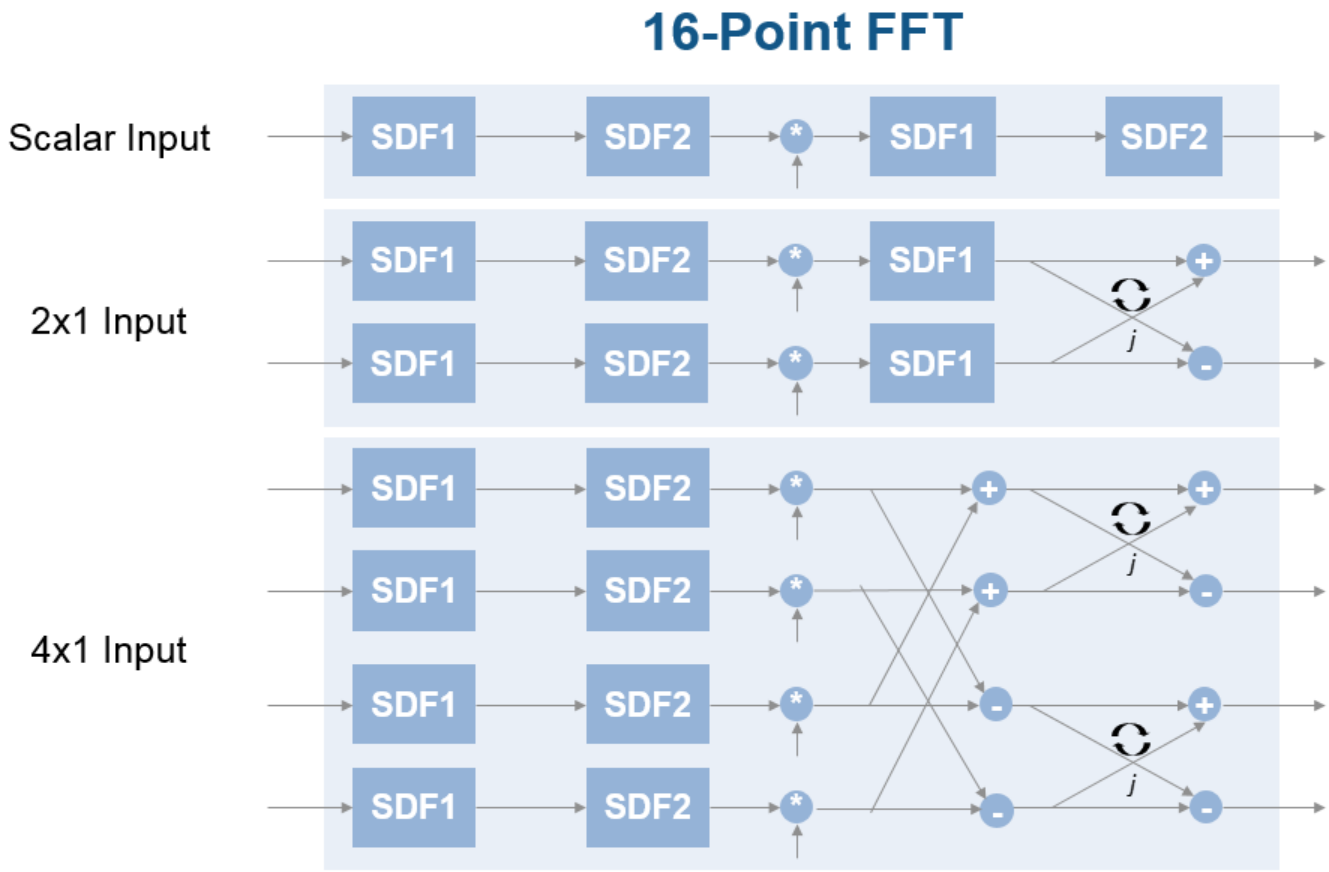
`Enable end output port` — Optional control signal indicating end of data
off (default) | on

This parameter enables a port that indicates the end of output data. When you select this parameter, the output **end** port appears on the block icon.

Algorithms

Streaming Radix 2^2

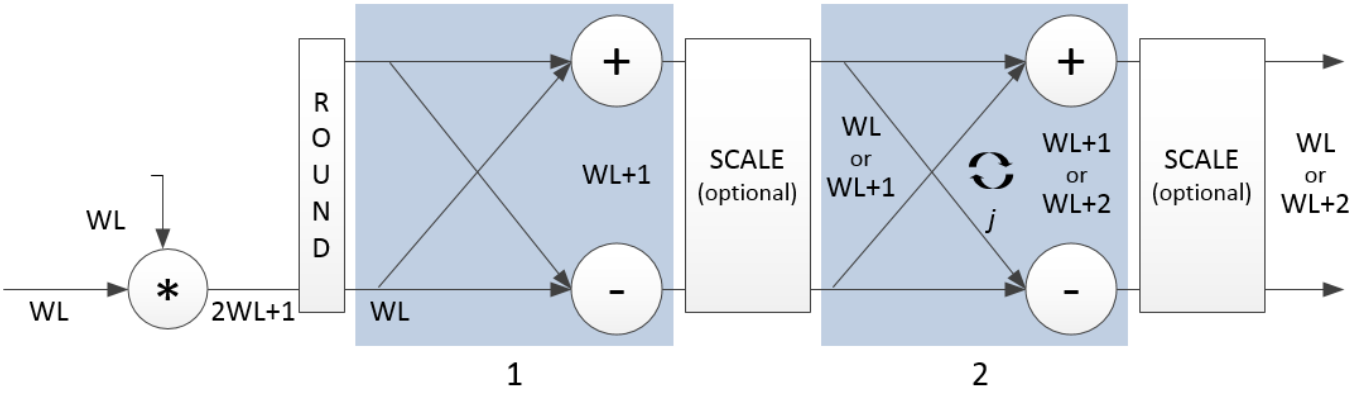
The streaming Radix 2^2 architecture implements a low-latency architecture. It saves resources compared to a streaming Radix 2 implementation by factoring and grouping the FFT equation. The architecture has $\log_4(N)$ stages. Each stage contains two single-path delay feedback (SDF) butterflies with memory controllers. When you use vector input, each stage operates on fewer input samples, so some stages reduce to a simple butterfly, without SDF.



The first SDF stage is a regular butterfly. The second stage multiplies the outputs of the first stage by –*j*. To avoid a hardware multiplier, the block swaps the real and imaginary parts of the inputs, and again swaps the imaginary parts of the resulting outputs. Each stage rounds the result of the twiddle factor multiplication to the input word length. The twiddle factors have two integer bits, and the rest of the bits are used for fractional bits. The twiddle factors have the same bit width as the input data, *WL*. The twiddle factors have two integer bits, and *WL*-2 fractional bits.

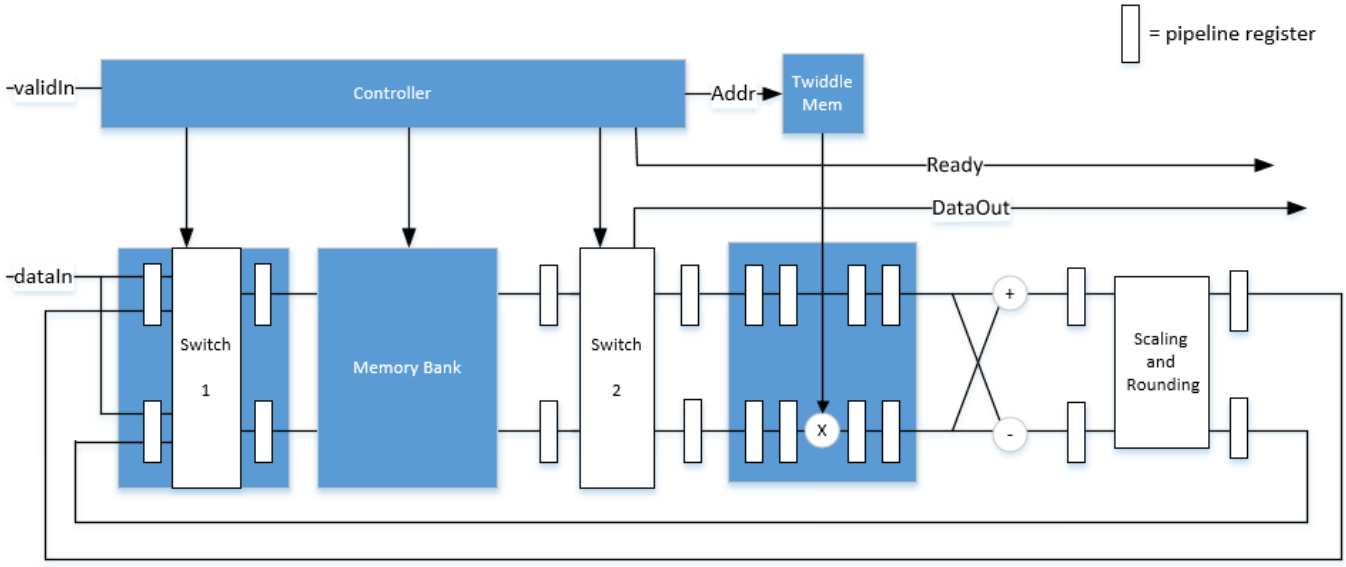
If you enable scaling, the algorithm divides the result of each butterfly stage by 2. Scaling at each stage avoids overflow, keeps the word length the same as the input, and results in an overall scale factor of 1/*N*. If scaling is disabled, the algorithm avoids overflow by increasing the word length by 1

bit at each stage. The diagram shows the butterflies and internal word lengths of each stage, not including the memory.



Burst Radix 2

The burst Radix 2 architecture implements the FFT by using a single complex butterfly multiplier. The algorithm cannot start until it has stored the entire input frame, and it cannot accept the next frame until computations are complete. The output **ready** port indicates when the algorithm is ready for new data. The diagram shows the burst architecture, with pipeline registers.



When you use this architecture, your input data must comply with the **ready** backpressure signal.

Control Signals

The algorithm processes input data only when the input **valid** port is 1. Output data is valid only when the output **valid** port is 1.

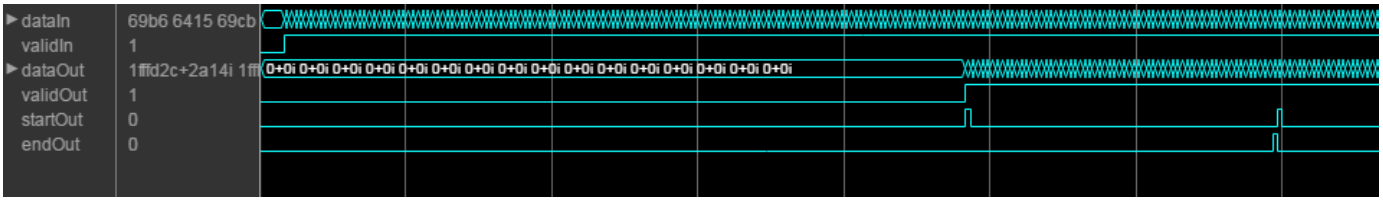
When the optional input **reset** port is 1, the algorithm stops the current calculation and clears all internal states. The algorithm begins new calculations when **reset** port is 0 and the input **valid** port starts a new frame.

Timing Diagram

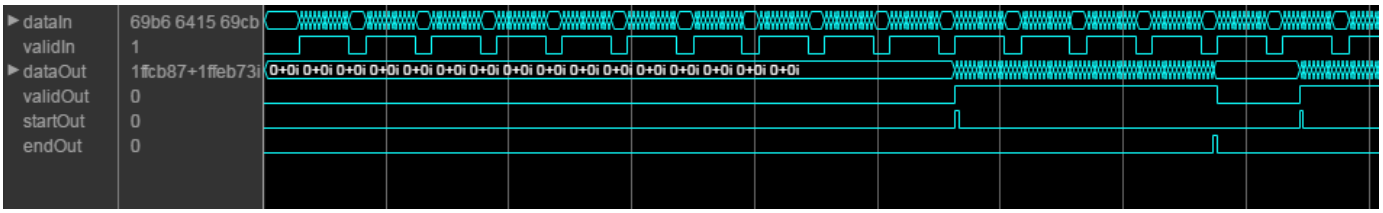
This diagram shows the input and output **valid** port values for contiguous scalar input data, streaming Radix 2^2 architecture, an FFT length of 1024, and a vector size of 16.

The diagram also shows the optional **start** and **end** port values that indicate frame boundaries. If you enable the **start** port, the **start** port value pulses for one cycle with the first valid output of the frame. If you enable the **end** port, the **start** port value pulses for one cycle with the last valid output of the frame.

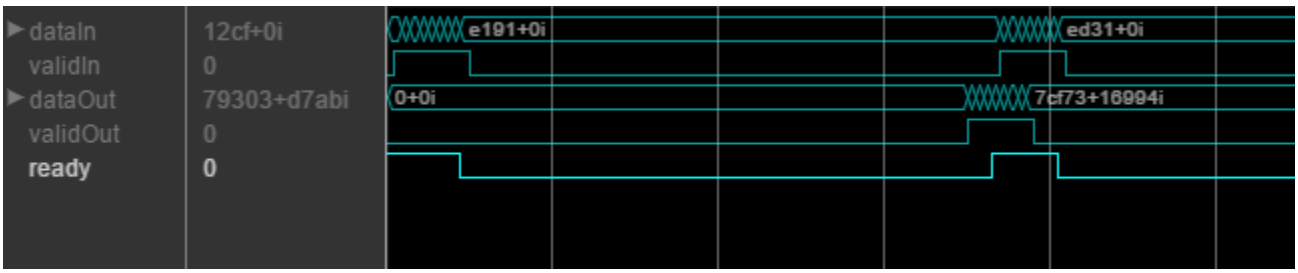
If you apply continuous input frames, the output will also be continuous after the initial latency.



The input **valid** port can be noncontiguous. Data accompanied by an input **valid** port is processed as it arrives, and the resulting data is stored until a frame is filled. Then the algorithm returns contiguous output samples in a frame of *N* (**FFT length**) cycles. This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16.



When you use the burst architecture, 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. You must apply input **data** and **valid** signals only when **ready** is `1` (true). The algorithm ignores any input **data** and **valid** signals when **ready** is `0` (false).

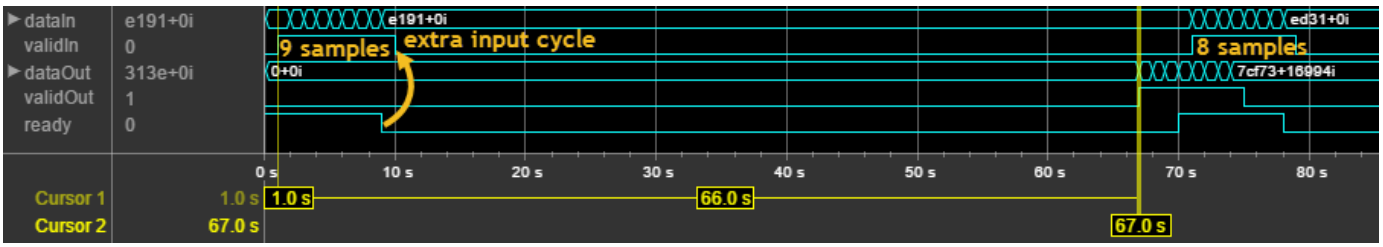


Latency

The latency varies with the **FFT length** and input vector size. After you update the model, the block icon displays the latency. The displayed latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. To obtain this latency programmatically, see “Automatic Delay Matching for the Latency of FFT Block”.

When using the burst architecture with a contiguous input, if your design waits for **ready** to output `0` before de-asserting the input **valid**, then one extra cycle of data arrives at the input. This data sample

is the first sample of the next frame. The algorithm can save one sample while processing the current frame. Due to this one sample advance, the observed latency of the later frames (from input **valid** to output **valid**) is one cycle shorter than the reported latency. The latency is measured from the first cycle, when input **valid** is 1 to the first cycle when output **valid** is 1. The number of cycles between when **ready** port is 0 and the output **valid** port is 1 is always *latency* – *FFTLength*.



Performance

This resource and performance data is the synthesis result from the generated HDL targeted to a Xilinx Virtex-6 (XC6VLX75T-1FF484) FPGA. The examples in the tables have this configuration:

- 1024 FFT length (default)
- Complex multiplication using 4 multipliers, 2 adders
- Output scaling enabled
- Natural order input, Bit-reversed output
- 16-bit complex input data
- Clock enables minimized (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options. For instance, reordering for a natural-order output uses more RAM than the default bit-reversed output, and real input uses less RAM than complex input.

For a scalar input Radix 2^2 configuration, the design achieves 326 MHz clock frequency. The latency is 1116 cycles. The design uses these resources.

Resource	Number Used
LUT	4597
FFS	5353
Xilinx LogiCORE DSP48	12
Block RAM (16K)	6

When you vectorize the same Radix 2^2 implementation to process two 16-bit input samples in parallel, the design achieves 316 MHz clock frequency. The latency is 600 cycles. The design uses these resources.

Resource	Number Used
LUT	7653
FFS	9322
Xilinx LogiCORE DSP48	24

Resource	Number Used
Block RAM (16K)	8

The block supports scalar input data only when implementing burst Radix 2 architecture. The burst design achieves 309 MHz clock frequency. The latency is 5811 cycles. The design uses these resources.

Resource	Number Used
LUT	971
FFS	1254
Xilinx LogiCORE DSP48	3
Block RAM (16K)	6

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox

Behavior changed in R2022a

Before R2022a, this block was named IFFT HDL Optimized, and was included in the DSP System Toolbox **DSP System Toolbox HDL Support** library.

FFT length of 4

Behavior changed in R2022a

This block now supports an FFT length of 4. In previous releases the FFT length had to be a power of 2 from 2^3 to 2^{16}.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).

InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

- You cannot generate HDL code for this block inside an Enabled Subsystem.

See Also

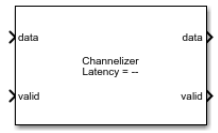
Blocks
FFT | Channelizer

Objects
`dsphdl.IFFT` | `dsphdl.FFT`

Introduced in R2014a

Channelizer

Polyphase filter bank and fast Fourier transform
Library: DSP HDL Toolbox / Filtering



Description

The Channelizer block separates a broadband input signal into multiple narrowband output signals. It provides hardware speed and area optimization for streaming data applications. The block accepts scalar or vector input of real or complex data, provides hardware-friendly control signals, and has optional output frame control signals. You can achieve gigasamples-per-second (GSPS) throughput using vector input. The block implements a polyphase filter, with one subfilter per input vector element. The hardware implementation interleaves the subfilters, which results in sharing each filter multiplier (*FFT Length* / *Input Size*) times. The FFT implementation uses the same pipelined Radix 2^2 FFT algorithm as the FFT block.

Ports

Input

`data` — Input data
scalar or column vector of real or complex values

Input data, specified as a scalar or a column vector of real or complex values.

The vector size must be a power of 2 and in the range [2, 64], and is not greater than the number of channels (FFT length).

`double` and `single` data types are supported for simulation, but not for HDL code generation.

The block does not accept `uint64` data.

Data Types: `fixed point` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `single` | `double`

`valid` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

`reset` — Clears internal states
scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select the **Enable reset input port** parameter.

Data Types: `Boolean`

Output

`data` — Frequency channel output data

vector

- If you set **Output vector size** to `Same as number of frequency bands` (default), the output data is a 1-by-*M* vector where *M* is the FFT length.
- If you set **Output vector size** to `Same as input size`, the output data is an *M*-by-1 vector where *M* is the input vector size.

The output order is bit natural for either output size. The output data type is a result of the **Filter output** and the bit growth in the FFT necessary to avoid overflow.

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

`start` — Indicates first valid cycle of output frame

scalar

Control signal that indicates the first valid cycle of the output frame.

When **start** is `1` (`true`), the block returns the first valid sample of the frame from the output **data** port.

Dependencies

To enable this port, on the **Control Ports** select the **Enable start output port** parameter.

Data Types: `Boolean`

end — Indicates last valid cycle of output frame

scalar

Control signal that indicates the last valid cycle of the output frame.

When **end** is `1` (`true`), the block returns the last valid sample of the frame from the output **data** port.

Dependencies

To enable this port, on the **Control Ports** select the **Enable end output port** parameter.

Data Types: `Boolean`

Parameters

Main

`Filter coefficients` — Polyphase filter coefficients

`[ -0.032, 0.121, 0.318, 0.482, 0.546, 0.482, 0.318, 0.121, -0.032 ]` (default) | vector of real or complex numeric values

If the number of coefficients is not a multiple of **Number of frequency bands (FFT length)**, the block pads this vector with zeros. The default filter specification is a raised-cosine FIR filter, `rcosdesign(0.25,2,4,'sqrt')`. You can specify a vector of coefficients or a call to a filter design function that returns the coefficient values. By default, the block casts the coefficients to the same data type as the input.

`Filter structure` — HDL filter architecture

`Direct form transposed` (default) | `Direct form systolic`

Specify the HDL filter architecture as one of these structures:

- `Direct form transposed` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. For architecture and performance details, see “Fully Parallel Transposed Architecture”.
- `Direct form systolic` — This architecture provides a fully parallel filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For architecture and performance details, see “Fully Parallel Systolic Architecture”.

All implementations share multipliers for symmetric and antisymmetric coefficients and remove multipliers for zero-valued coefficients.

`Number of frequency bands (FFT length)` — FFT length

8 (default) | integer power of two

For HDL code generation, the FFT length must be a power of 2 from 2^2 to 2^{16}.

`Complex multiplication` — HDL implementation of complex multipliers

`Use 4 multipliers and 2 adders` (default) | `Use 3 multipliers and 5 adders`

HDL implementation of complex multipliers, specified as either `'Use 4 multipliers and 2 adders'` or `'Use 3 multipliers and 5 adders'`. Depending on your synthesis tool and target device, one option may be faster or smaller.

`Output vector size` — Size of output data

`Same as number of frequency bands` (default) | `Same as input size`

The output data is a vector of *M* elements. The output order is bit natural for either output size.

- `Same as number of frequency bands` — Output data is a 1-by-*M* vector, where *M* is the FFT length.

- `Same as input size` — Output data is an *M*-by-1 vector, where *M* is the input vector size.

`Divide butterfly outputs by two` — FFT scaling
on (default) | off

When you select this parameter, the FFT implements an overall 1/*N* scale factor by scaling the result of each pipeline stage by 2. This adjustment keeps the output of the FFT in the same amplitude range as its input. If scaling is disabled, the FFT avoids overflow by increasing the word length by 1 bit at each stage.

Data Types

`Rounding mode` — Rounding method used for internal fixed-point calculations
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

See “Rounding Modes”. The block uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is `single` or `double`. Each FFT stage rounds after the twiddle factor multiplication but before the butterflies. Rounding can also occur when casting the coefficients and the output of the polyphase filter to the data types you specify.

`Saturate on integer overflow` — Overflow handling for internal fixed-point calculations
off (default) | on

See “Overflow Handling”. The block uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is `single` or `double`. This option applies to casting the coefficients and the output of the polyphase filter to the data types you specify.

The FFT algorithm avoids overflow by either scaling the output of each stage (`Normalize` enabled), or by increasing the word length by 1 bit at each stage (`Normalize` disabled).

`Coefficients` — Data type of the filter coefficients
`Inherit: Same word length as input` (default) | data type expression

The block casts the polyphase filter coefficients to this data type, using the rounding and overflow settings you specify. When you select `Inherit: Same word length as input` (default), the block selects the binary point using `fi()` best-precision rules.

`Filter output` — Data type of the output of the polyphase filter
`Inherit: Same word length as input` (default) | `Inherit: via internal rule` | data type expression

The block casts the output of the polyphase filter (the input to the FFT) to this data type, using the rounding and overflow settings you specify. When you select `Inherit: via internal rule`, the block selects a best-precision binary point by considering the values of your filter coefficients and the range of your input data type.

By default, the FFT logic does not modify the data type. When you disable **Divide butterfly outputs by two**, the FFT increases the word length by 1 bit at each stage to avoid overflow.

Control Ports

`Enable reset input port` — Optional reset signal
off (default) | on

When you select this parameter, the **reset** port shows on the block icon. When the **reset** input is `true`, the block stops calculation and clears all internal state.

`Enable start output port` — Optional control signal indicating start of data
off (default) | on

When you select this parameter, the **start** port shows on the block icon. The **start** signal is true for the first cycle of output data in a frame.

`Enable end output port` — Optional control signal indicating end of data
off (default) | on

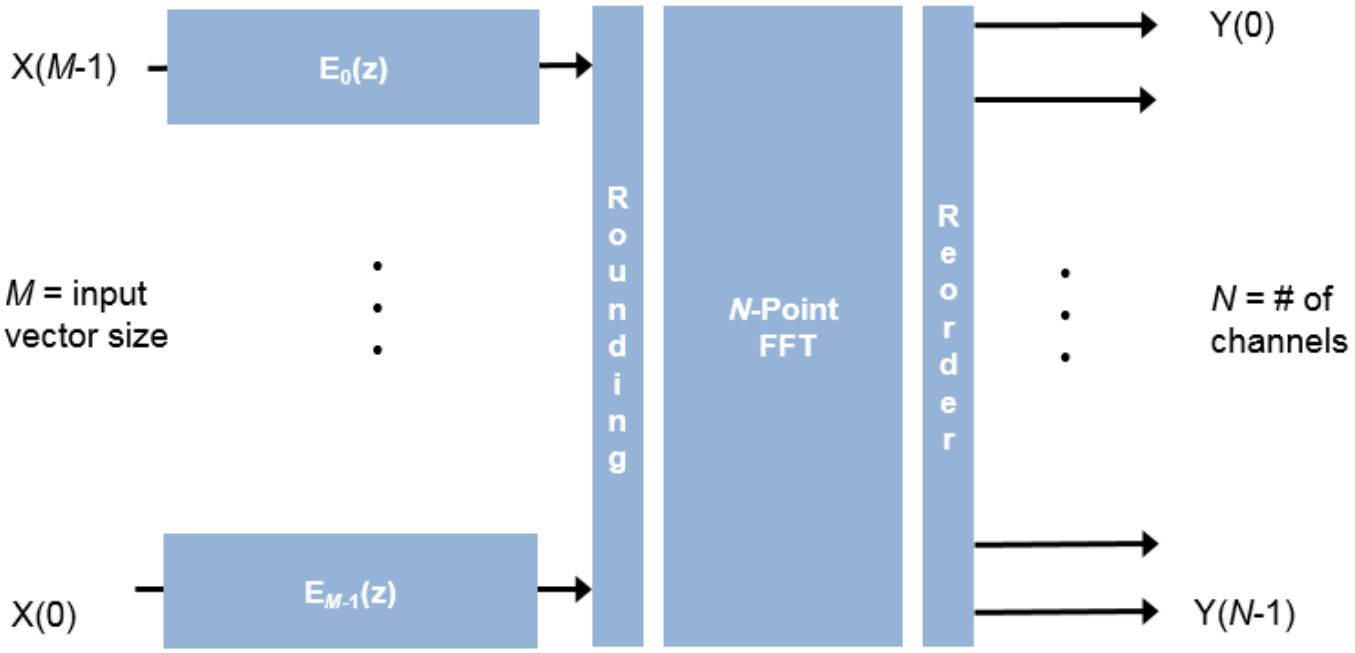
When you select this parameter, the **end** port shows on the block icon. The **end** signal is true for the last cycle of output data in a frame.

Algorithms

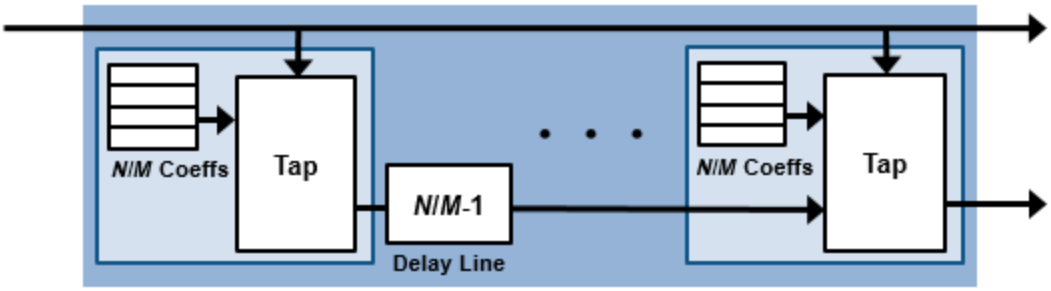
The polyphase filter algorithm requires a subfilter for each FFT channel. For more detail on the polyphase filter architecture, refer to [1], and to the Channelizer (DSP System Toolbox) block reference page.

Note The output of this block does not match the output from the Channelizer (DSP System Toolbox) block sample-for-sample. This mismatch is because the blocks apply the input samples to the subfilters in different orders. The Channelizer (DSP System Toolbox) block applies input X(0) to subfilter $E_{M-1}(z)$, X(1) to subfilter $E_{M-2}(z)$, ..., X(M-1) to subfilter $E_0(z)$. The channels detected by both blocks match, when analyzed over multiple frames.

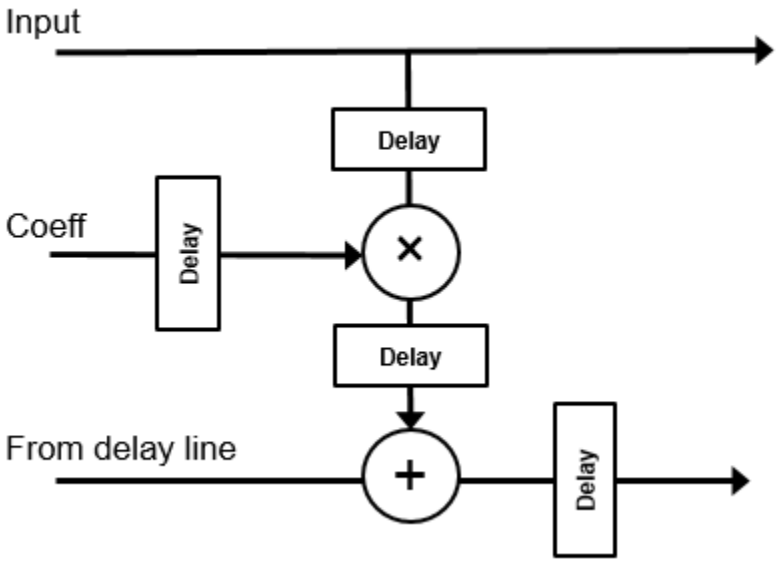
If the input vector size, M, is the same as the FFT length, N, then the block implements N subfilters in the hardware. Each subfilter is an FIR filter (`Direct form transposed` or `Direct form systolic`) with *NumCoeffs*/N taps.



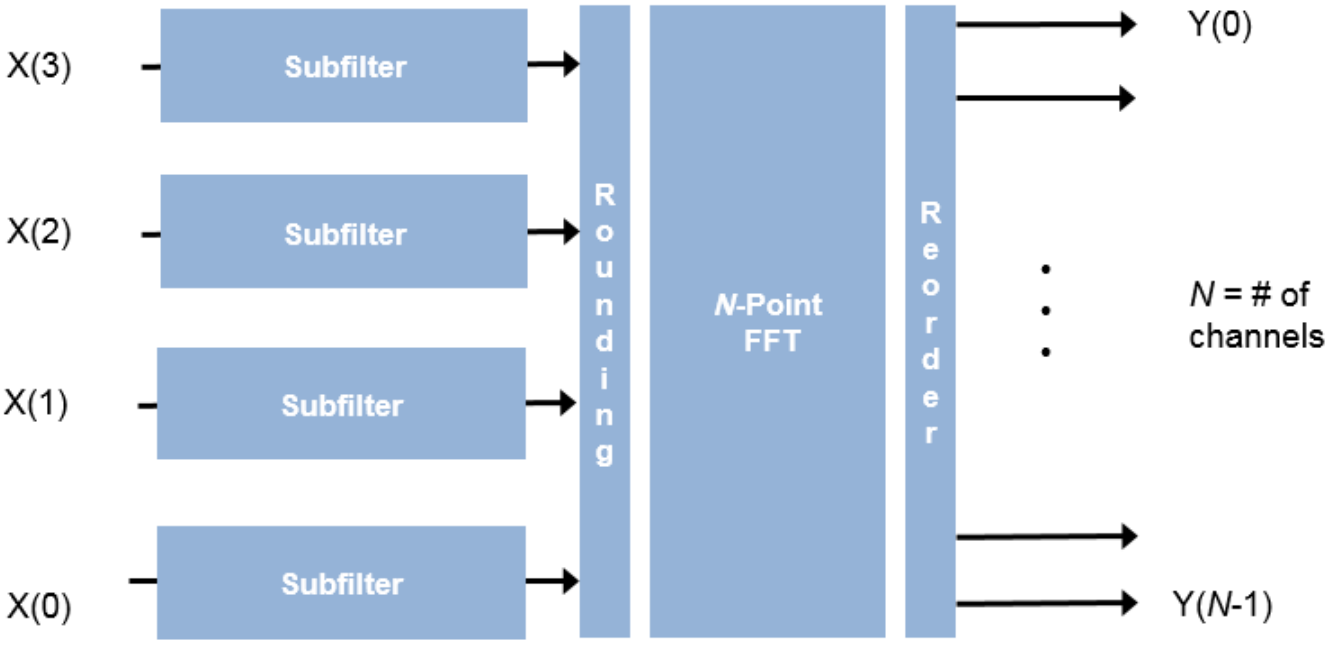
If the vector size is less than *N*, the block implements one subfilter for each input vector element. The subfilter multipliers are shared as necessary to implement *N* channel filters. The shared multiplier taps have a lookup table for *N*/*M* filter coefficients. Each tap is followed by a delay line of *N*/*M*–1 cycles.



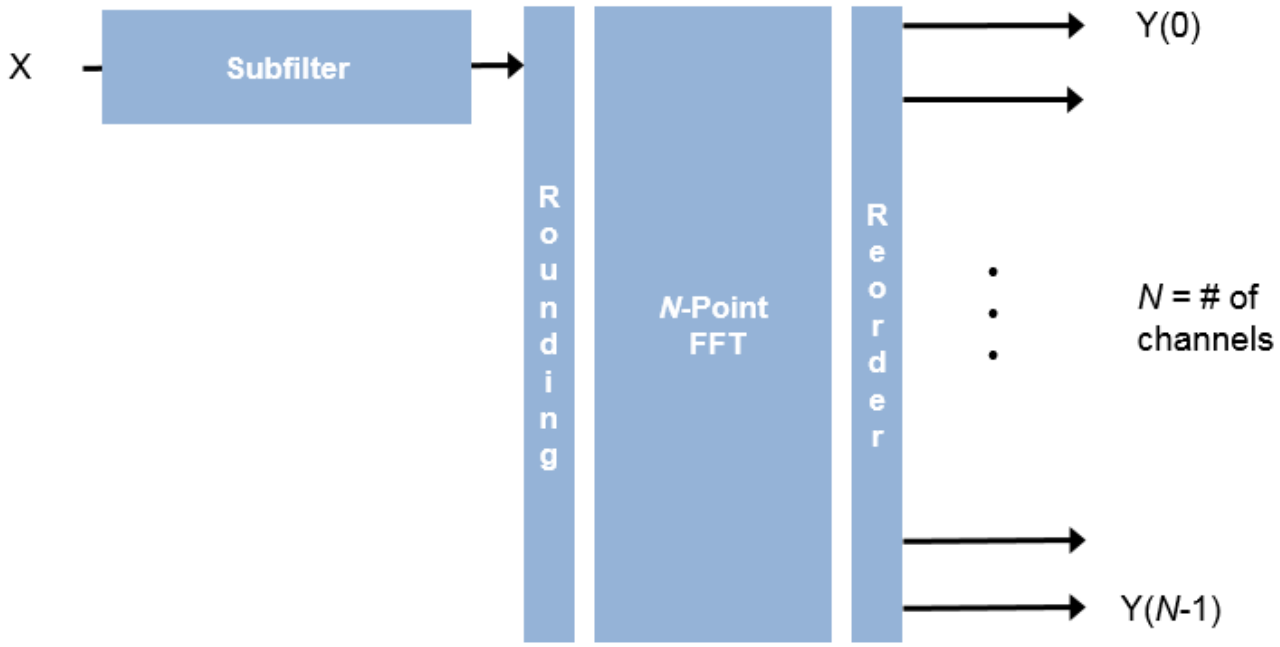
The output of the subfilters is cast to the specified **Filter output**, using the rounding and overflow settings you chose. Each filter tap in the subfilter is pipelined to target the DSP sections of an FPGA.



For instance, for an FFT length of 8, and an input vector size of 4, the block implements four filters. Each multiplier is shared *N*/*M* times, or twice. Each tap applies two coefficients, and the delay line is *N*/*M*–1 cycles.



For scalar input, the block implements one filter. Each multiplier is shared *N* times. Each tap applies *N* coefficients, and the delay line is *N*–1 cycles.



Latency

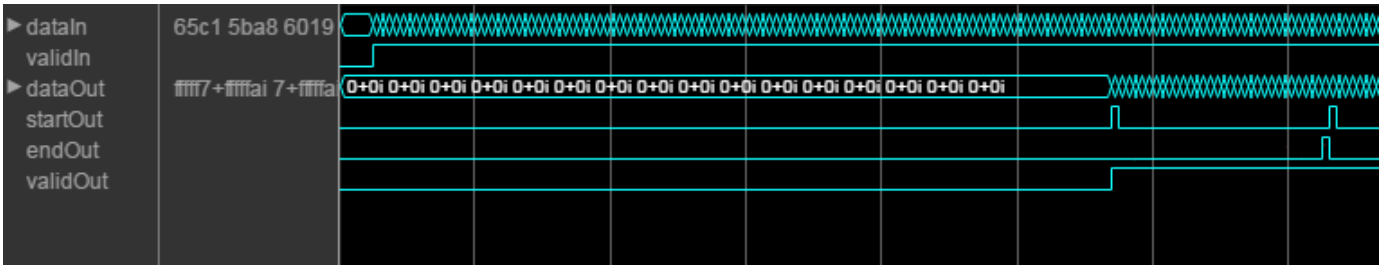
The latency varies with FFT length, vector size, and filter structure. After you update the model, the latency is displayed on the block icon. The displayed latency is the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous. The filter coefficients do not affect the latency. Setting the output size equal to the input size reduces the latency, because the samples are not saved and reordered.

Control Signals

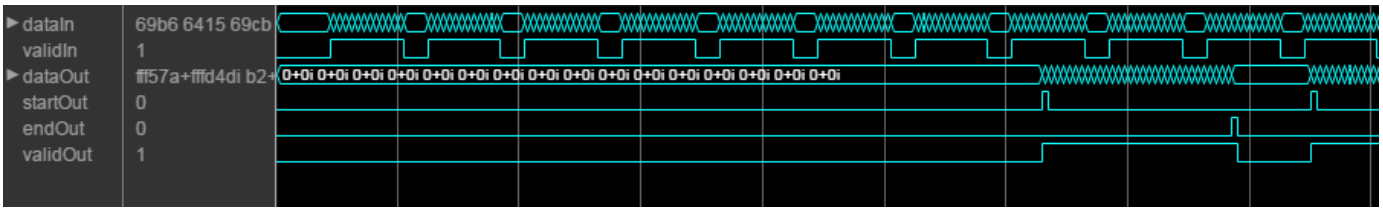
This diagram shows `validIn` and `validOut` signals for contiguous input data with a vector size of 16, an FFT length of 512, and when you select the `Direct form transposed` filter architecture. In this example, the output vector size is specified same as the input vector size.

The diagram also shows the optional `startOut` and `endOut` signals that indicate frame boundaries. When enabled, `startOut` pulses for one cycle with the first `validOut` of the frame, and `endOut` pulses for one cycle with the last `validOut` of the frame.

If you apply continuous input frames (no gap in `validIn` between frames), the output will also be continuous, after the initial latency.



The `validIn` signal can be noncontiguous. Data accompanied by a `validIn` signal is stored until a frame is filled. Then the data in output is a contiguous frame of *N*/*M* cycles. This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16 samples.



Performance

These resource and performance data are the place-and-route results from the generated HDL targeted to a Xilinx Zynq- 7000 ZC706 evaluation board. The three examples in the tables use this common configuration.

- FFT length (default) — 8
- Filter length — 96 coefficients
- Filter structure — Direct form transposed
- 16-bit complex input data
- Coefficient data type — Same word length as input
- Filter output data type — Same word length as input
- Complex multiplication — Use 4 multipliers and 2 adders
- Output scaling — Enabled
- Output vector size — Same as input size

Performance of the synthesized HDL code varies with your target and synthesis options.

For scalar input, the design achieves a clock frequency of 506.84 MHz. The latency is 51 cycles. The subfilters share each multiplier eight (*N*) times. The design uses these resources.

Resource	Number Used
LUT	2898
FFS	3746
Xilinx LogiCORE DSP48	28

For four-sample vector input, the design achieves a clock frequency of 452 MHz. The latency is 37 cycles. The subfilters share each multiplier twice (*N*/*M*). The design uses these resources.

Resource	Number Used
LUT	1991
FFS	8305
Xilinx LogiCORE DSP48	104

For eight-sample vector input, the design achieves a clock frequency of 360 MHz. The latency is 18 cycles. When the input size is the same as the FFT length, the subfilters do not share any multipliers. The design uses these resources.

Resource	Number Used
LUT	1683
FFS	2992
Xilinx LogiCORE DSP48	208

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this block was named Channelizer HDL Optimized, and was included in the DSP System Toolbox **DSP System Toolbox HDL Support** library.

The block now supports fully parallel systolic architecture when you set the **Filter structure** parameter to `Direct form systolic`.

FFT length of 4
Behavior changed in R2022a

This block now supports an FFT length of 4. In previous releases the FFT length had to be a power of 2 from 2^3 to 2^{16}.

Direct form systolic filter structure support
Behavior changed in R2022a

This block now supports direct form systolic filter structure.

References

[1] Harris, F. J., C. Dick, and M. Rice. “Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications.” *IEEE Transactions on Microwave Theory and Techniques*. Vol. 51, No. 4, April 2003.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

See Also

Blocks
Channel Synthesizer | FFT

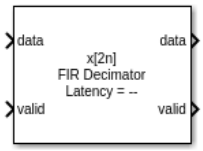
Objects
`dsphdl.Channelizer` | `dsphdl.ChannelSynthesizer`

Topics
"High-Throughput HDL Algorithms"

Introduced in R2017a

FIR Decimator

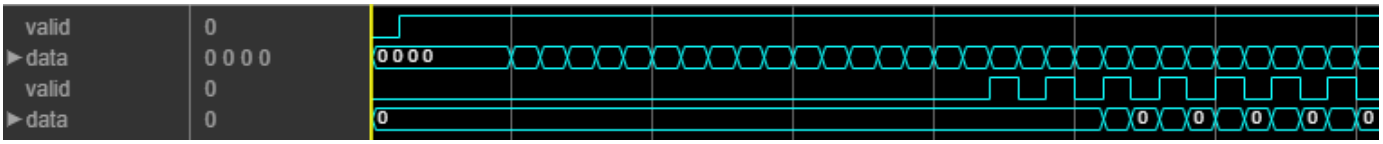
Finite impulse response (FIR) decimation filter
Library: DSP HDL Toolbox / Filtering



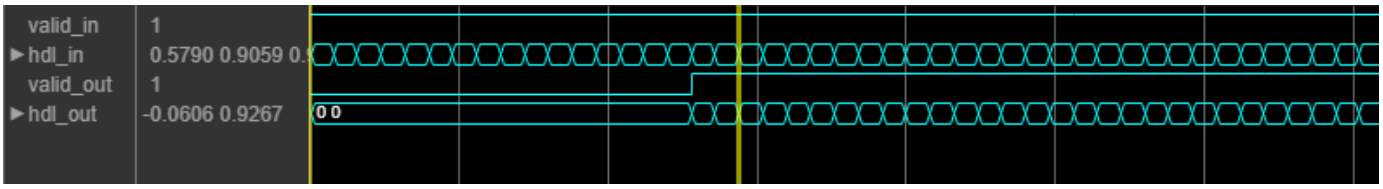
Description

The FIR Decimator block implements a single-rate polyphase FIR decimation filter that is optimized for HDL code generation. The block provides a hardware-friendly interface with input and output control signals. To provide a cycle-accurate simulation of the generated HDL code, the block models architectural latency including pipeline registers and resource sharing.

The block accepts scalar or vector input. When you use vector input and the vector size is less than the decimation factor, the decimation factor must be an integer multiple of the vector size. In this case, the output is scalar and an output valid signal indicates which samples are valid after decimation. The output data is valid every *DecimationFactor*/*VectorSize* samples. The waveform shows an input vector of four samples and a decimation factor of eight. The output data is a scalar that is valid every second cycle.



When you use vector input and the vector size is greater than the decimation factor, the vector size must be an integer multiple of the decimation factor. In this case, the output is a vector of *VectorSize*/*DecimationFactor* samples. The waveform shows an input vector of eight samples and a decimation factor of four. The output data is a vector of two samples on every cycle.



The block provides two filter structures. The direct form systolic architecture provides an implementation that makes efficient use of Intel and Xilinx DSP blocks. This architecture can be fully parallel or serial. To use a serial architecture, the input samples must be spaced out with a regular number of invalid cycles between the valid samples. The direct form transposed architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. 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.

All filter structures optimize hardware resources by sharing multipliers for symmetric or antisymmetric filters and by removing the multipliers for zero-valued coefficients such as in half-band filters and Hilbert transforms.

The block implements one filter for each sample in the input vector. The block then shares this filter between the polyphase subfilters by interleaving the subfilter coefficients in time.

Ports

Input

`data` — Input data

scalar | vector

Input data must be a real- or complex-valued scalar or vector. When you use vector input and the vector size is less than the decimation factor, the decimation factor must be an integer multiple of the vector size. When you use vector input and the vector size is greater than the decimation factor, the vector size must be an integer multiple of the decimation factor. The vector size must be less than or equal to 64.

When the input data type is an integer type or a fixed-point type, the block uses fixed-point arithmetic for internal calculations.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fixed point` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`valid` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

`reset` — Clears internal states

scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select **Enable reset input port**.

Data Types: `Boolean`

Output

`data` — Filtered output data

scalar

Filtered output data, returned as a real- or complex-valued scalar. When the input data type is a floating-point type, the output data inherits the data type of the input data. When the input data type is an integer type or a fixed-point type, the **Output** parameter on the **Data Types** tab specifies the output data type.

The output **valid** signal indicates which samples are valid after decimation. When the input vector size is greater than the decimation factor, the output is a vector of *VectorSize*/*DecimationFactor* samples.

Data Types: `fixed point` | `single` | `double`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

Parameters

Main

`Coefficients` — FIR filter coefficients

`fir1(35,0.4)` (default) | real- or complex-valued vector

FIR filter coefficients, specified as a real- or complex-valued vector. You can specify the vector as a workspace variable or as a call to a filter design function. When the input data type is a floating-point type, the block casts the coefficients to the same data type as the input. When the input data type is an integer type or a fixed-point type, you can set the data type for the coefficients on the **Data Types** tab.

Example: `firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0])` defines coefficients using a linear-phase filter design function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`Filter structure` — HDL filter architecture

`Direct form systolic` (default) | `Direct form transposed`

The block implements a polyphase decomposition filter by using Discrete FIR Filter blocks. Both structures share resources by interleaving the subfilter coefficients over one filter implementation for each sample in the input vector. Specify the HDL filter architecture as one of these structures:

- `Direct form systolic` — This architecture provides a parallel or partly-serial filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For a partly-serial implementation, specify a value greater than 1 for the **Minimum number of cycles between valid input samples** parameter. You cannot use frame-based input with the partly-serial architecture.

  When **Minimum number of cycles between valid input samples** is greater than 1, the block chooses a filter architecture that results in the fewest multipliers. If *N* allows for a single multiplier in each subfilter, then the block implements a single serial filter and decimates the output samples.

- `Direct form transposed` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications.

All implementations share resources by interleaving the subfilter coefficients over one filter implementation for each sample in the input vector.

The block implements a polyphase decomposition filter using Discrete FIR Filter blocks. For architecture details, see “FIR Filter Architectures for FPGAs and ASICs”.

`Decimation factor` — Decimation factor
2 (default) | integer greater than two

Specify an integer decimation factor greater than two. When you use vector input and the vector size is less than the decimation factor, the decimation factor must be an integer multiple of the vector size. When you use vector input and the vector size is greater than the decimation factor, the vector size must be an integer multiple of the decimation factor.

`Minimum number of cycles between valid input samples` — Serialization requirement for input timing
1 (default) | positive integer

Serialization requirement for input timing, specified as a positive integer. This parameter represents *N*, the minimum number of cycles between valid input samples. To implement a fully serial architecture, set **Minimum number of cycles between valid input samples** greater than the filter length, *L*, or to `Inf`.

The block applies coefficient optimizations before serialization, so the sharing factor of the final filter can be lower than the number of cycles that you specified.

Dependencies

To enable this parameter, set **Filter structure** to `Direct form systolic`.

You cannot use frame-based input with **Minimum number of cycles between valid input samples** greater than 1.

Data Types

`Rounding mode` — Rounding mode for type-casting the output
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Rounding mode for type-casting the output to the data type specified by the **Output** parameter. When the input data type is floating point, the block ignores this parameter. For more details, see “Rounding Modes”.

`Saturate on integer overflow` — Overflow handling for type-casting the output
`off` (default) | `on`

Overflow handling for type-casting the output to the data type specified by the **Output** parameter. When the input data type is floating point, the block ignores this parameter. For more details, see “Overflow Handling”.

`Coefficients` — Data type of filter coefficients
`Inherit: Same word length as input` (default) | `<data type expression>`

The block casts the filter coefficients to this data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the block ignores this parameter.

The recommended data type for this parameter is `Inherit: Same word length as input`.

The block returns a warning or error if either of these conditions occur.

- The coefficients data type does not have enough fractional length to represent the coefficients accurately.
- The coefficients data type is unsigned, and the coefficients include negative values.

`Output` — Data type of filter output
`Inherit: Inherit via internal rule` (default) | `Inherit: Same word length as input` | `<data type expression>`

The block casts the output of the filter to this data type. The quantization uses the settings of the **Rounding mode** and **Overflow mode** parameters. When the input data type is floating point, the block ignores this parameter.

The block increases the word length for full precision inside each filter tap and casts the final output to the specified type. The maximum final internal data type (*WF*) depends on the input data type (*WI*), the coefficient data type (*WC*), and the number of coefficients (*L*) and is given by

$WF = WI + WC + \text{ceil}(\log2(L))$.

Because the coefficient values limit the potential growth, usually the actual full-precision internal word length is smaller than *WF*.

Control Ports

`Enable reset input port` — Option to enable reset input port
`off` (default) | `on`

Select this parameter to enable the **reset** input port. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

`Use HDL global reset` — Option to connect data path registers to generated HDL global reset signal
`off` (default) | `on`

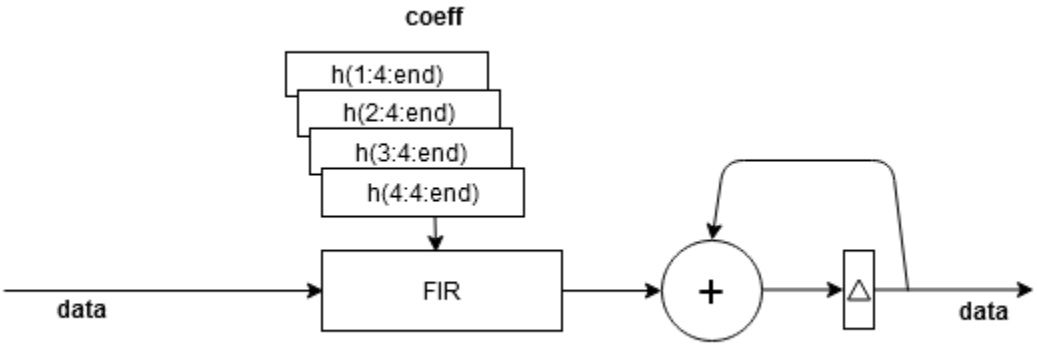
Select this parameter to connect the generated HDL global reset signal to the data path registers. This parameter does not change the appearance of the block or modify simulation behavior in Simulink. When you clear this parameter, the generated HDL global reset clears only the control path registers. 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.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

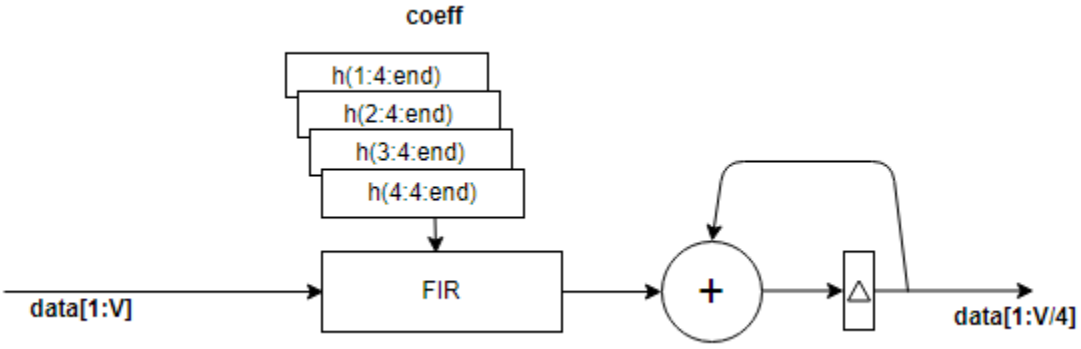
Algorithms

The block implements a polyphase filter bank where the filter coefficients are decomposed into **Decimation factor** subfilters. If the filter length is not divisible by the **Decimation factor** parameter value, then the block zero-pads the coefficients.

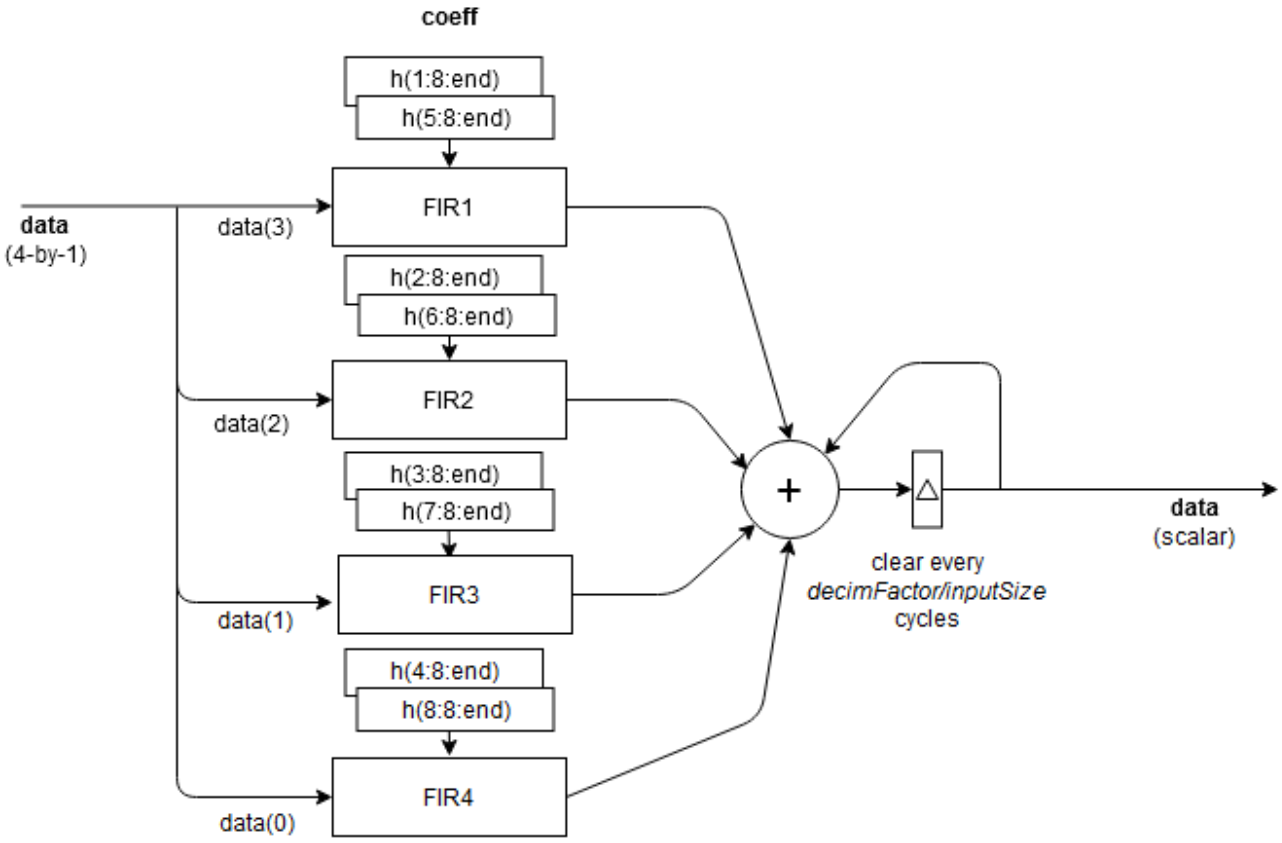
The diagram shows the polyphase filter bank with scalar input, the **Decimation factor** parameter set to 4, and **Minimum number of cycles between valid input samples** set to 1. The four sets of decomposed coefficients are interleaved in time over a single subfilter. The output data sample is valid every 4 cycles.



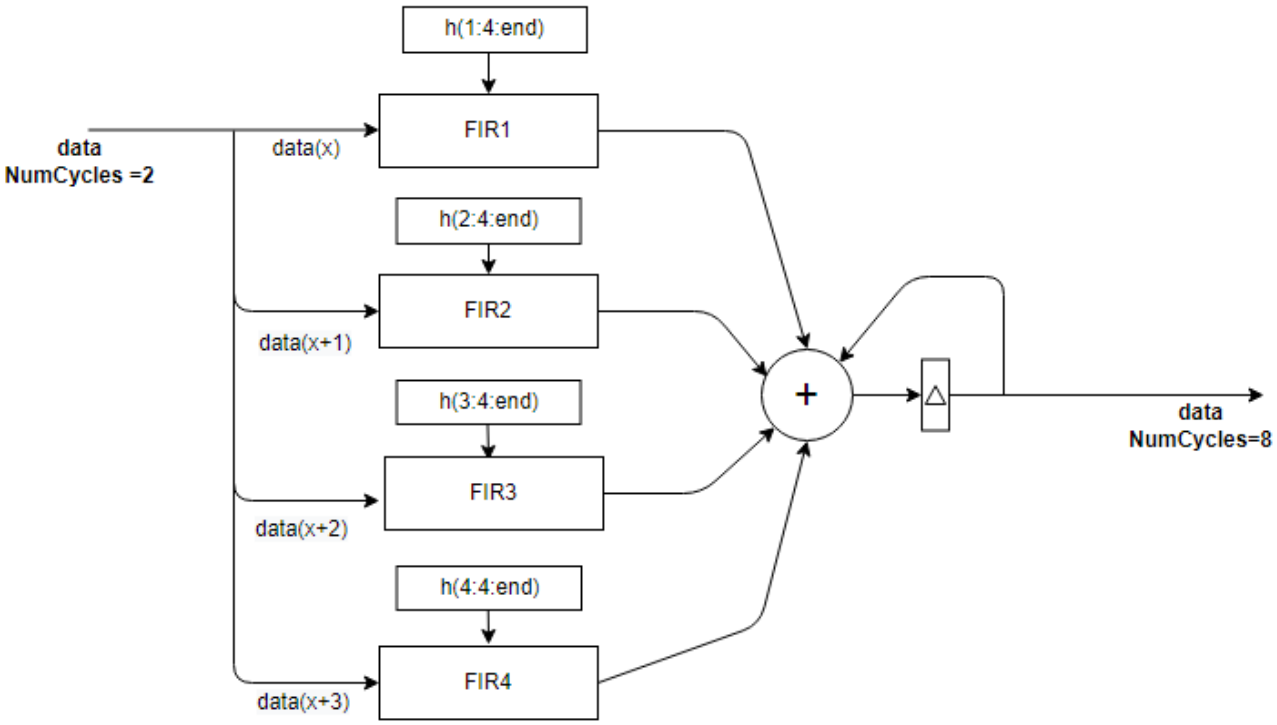
With an input vector size greater than the decimation factor, the block implements the same interleaved filter but with frame-based input. The output vector has *VectorSize*/*DecimationFactor* samples.



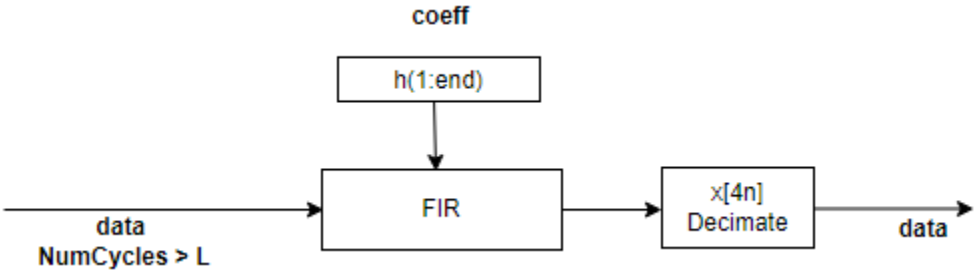
The next diagram shows the polyphase filter bank for an input vector size smaller than the decimation factor. This filter has an input vector of four values and the **Decimation factor** parameter set to eight. Each of the four subfilters has two sets of coefficients interleaved in time.



When the filter has **Minimum number of cycles between valid input samples** greater than one and less than the number of filter coefficients, the block implements a polyphase filter with *DecimationFactor* subfilters. This diagram shows input data with a valid sample every second cycle and a *DecimationFactor* of 4. The output data has one valid sample every eight cycles.



When the filter has **Minimum number of cycles between valid input samples** greater than the number of filter coefficients, the block implements a single fully serial filter and decimates the output samples by the decimation factor.



Each subfilter is implemented with a Discrete FIR Filter block. The adder at the output is pipelined to accommodate higher synthesis frequencies. For architecture details, see “FIR Filter Architectures for FPGAs and ASICs”.

Note The output of the FIR Decimator block does not match the output from the FIR Decimation block from DSP System Toolbox sample-for-sample. This difference is mainly because of the phase that the samples are applied across the subfilters. To match the FIR Decimation block, apply **Decimation factor** – 1 zeros to the FIR Decimator block at the start of the data stream.

The FIR Decimation block also uses slightly different data types for full-precision calculations. The different data types can also introduce differences in output values if the values overflow the internal data types.

Performance

This table shows the post-synthesis resource utilization for the HDL code generated for the default FIR decimation filter using scalar input, a decimation factor of eight, 16-bit input, and 16-bit coefficients. The synthesis targets a Xilinx ZC-706 (XC7Z045ffg900-2) FPGA. The **Global HDL reset type** parameter is `Synchronous`, and the **Minimize clock enables** parameter is selected. The **reset** port is disabled, so only the control path registers are connected to the generated global HDL reset.

Resource	Uses
LUT	676
Slice Reg	878
Slice	257
Xilinx LogiCORE DSP48	5

After place and route, the maximum clock frequency of the design is 526 MHz.

For the same filter with a four-element input vector, the filter uses these resources.

Resource	Uses
LUT	322
Slice Reg	2351
Slice	502
Xilinx LogiCORE DSP48	20

After place and route, the maximum clock frequency of the design is 518 MHz.

For the same filter with scalar input and *numCycles* set to four, the filter uses these resources.

Resource	Uses
LUT	835
Slice Reg	1341
Xilinx LogiCORE DSP48	8

After place and route, the maximum clock frequency of the design is 460 MHz.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this block was named FIR Decimation HDL Optimized, and was included in the DSP System Toolbox **DSP System Toolbox HDL Support** library.

Serial systolic architecture

This block now supports partly and fully serial systolic architecture. This architecture enables you to share hardware resources if there is a regular pattern of invalid cycles between valid input samples. To use the serial systolic architecture, set **Filter structure** to `Direct form systolic` and

Minimum number of cycles between valid input samples to a value greater than 1. You cannot use frame-based input with the serial architecture.

Input vector size can be greater than decimation factor

In previous releases, the block did not support input vector sizes greater than the decimation factor.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

The FIR Decimator block does not support resource sharing optimization through HDL Coder settings. Instead, set the **Filter structure** parameter to `Partly serial systolic`, and configure a serialization factor based on either input timing or resource usage.

See Also

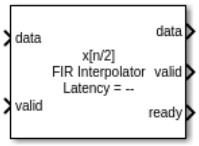
Objects
`dsphdl.FIRDecimator` | `dsphdl.FIRFilter`

Blocks
Discrete FIR Filter

Introduced in R2020b

FIR Interpolator

Finite impulse response (FIR) interpolation filter
Library: DSP HDL Toolbox / Filtering



Description

The FIR Interpolator block implements a single-rate polyphase FIR interpolation filter that is optimized for HDL code generation. The block provides a hardware-friendly interface with input and output control signals. To provide a cycle-accurate simulation of the generated HDL code, the block models architectural latency including pipeline registers and resource sharing.

The block accepts scalar or vector input and outputs a scalar or vector depending on the interpolation factor and the number of cycles between input samples. The block implements a polyphase decomposition with *InterpolationFactor* subfilters. Each subfilter can implement a serial architecture if there is regular spacing between input samples.

The block provides two filter structures. The direct form systolic architecture provides an implementation that makes efficient use of Intel and Xilinx DSP blocks. This architecture can be fully-parallel or serial. To use a serial architecture, the input samples must be spaced out with a regular number of invalid cycles between the valid samples. The direct form transposed architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. 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.

All filter structures optimize hardware resources by sharing multipliers for symmetric or antisymmetric filters and by removing the multipliers for zero-valued coefficients such as in half-band filters and Hilbert transforms.

Ports

Input

`data` — Input data
real or complex scalar or vector

Input data, specified as a real or complex scalar or vector. The vector size must be less than or equal to 64. When the input data type is an integer type or a fixed-point type, the block uses fixed-point arithmetic for internal calculations.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fixed point` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`valid` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

`reset` — Clears internal states

scalar

Control signal that clears internal states. When **reset** is `1` (`true`), the block stops the current calculation and clears internal states. When the **reset** is `0` (`false`) and the input **valid** is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this port, on the **Control Ports** tab, select **Enable reset input port**.

Data Types: `Boolean`

Output

`data` — Interpolated output data

real or complex scalar or vector

Interpolated output data, returned as a real or complex scalar or vector. The vector size is *InputSize* * *InterpolationFactor*. When *NumCycles* is greater than *InterpolationFactor*, scalar output samples are spaced with `floor(`*`NumCycles/InterpolationFactor`*`)` invalid cycles, and the output **valid** signal indicates which samples are valid after interpolation.

When the input data type is a floating-point type, the output data inherits the data type of the input data. When the input data type is an integer type or a fixed-point type, the **Output** parameter on the **Data Types** tab controls the output data type.

Data Types: `fixed point` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

`ready` — Indicates block is ready for new input data

scalar

Control signal that indicates that the block is ready for new input data sample on the next cycle. When **ready** is `1` (`true`), you can specify the **data** and **valid** inputs for the next time step. When **ready** is `0` (`false`), the block ignores any input data in the next time step.

Data Types: `Boolean`

Parameters

Main

`Coefficients` — FIR filter coefficients

`fir1(35,0.4)` (default) | real- or complex-valued vector

FIR filter coefficients, specified as a real- or complex-valued vector. You can specify the vector as a workspace variable or as a call to a filter design function. When the input data type is a floating-point type, the block casts the coefficients to the same data type as the input. When the input data type is an integer type or a fixed-point type, you can set the data type for the coefficients on the **Data Types** tab.

Example: `firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0])` defines coefficients by using a linear-phase filter design function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`Filter structure` — HDL filter architecture

`Direct form systolic` (default) | `Direct form transposed`

Specify the HDL filter architecture as one of these structures:

- `Direct form systolic` — This architecture provides a parallel or partly serial filter implementation that makes efficient use of Intel and Xilinx DSP HDL blocks. For a partly serial implementation, specify a value greater than 1 for the **Minimum number of cycles between valid input samples** parameter. You cannot use frame-based input with the partly serial architecture.
- `Direct form transposed` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications.

The block implements a polyphase decomposition filter by using Discrete FIR Filter blocks. Each filter phase shares resources internally where coefficients and serial options allow. For architecture details, see “FIR Filter Architectures for FPGAs and ASICs”.

`Interpolation factor` — Interpolation factor

2 (default) | integer greater than two

Specify an integer interpolation factor greater than two. The output vector size is *InputSize* * *InterpolationFactor*. The output vector size must be less than 64 samples.

`Minimum number of cycles between valid input samples` — Serialization requirement for input timing

1 (default) | positive integer

Serialization requirement for input timing, specified as a positive integer. This parameter represents *N*, the minimum number of cycles between valid input samples. When you set **Minimum number of cycles between valid input samples** greater than the filter length, *L*, and the input and coefficients are both real, the filter uses **Interpolation factor** multipliers.

Because the block applies coefficient optimizations before serialization, the sharing factor of the final filter can be lower than the number of cycles that you specified.

Dependencies

To enable this parameter, set **Filter structure** to `Direct form systolic`.

You cannot use frame-based input with **Minimum number of cycles between valid input samples** greater than 1.

Data Types

`Rounding mode` — Rounding mode for type-casting the output

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Rounding mode for type-casting the output to the data type specified by the **Output** parameter. When the input data type is floating point, the block ignores this parameter. For more details, see "Rounding Modes".

`Saturate on integer overflow` — Overflow handling for type-casting the output

`off` (default) | `on`

Overflow handling for type-casting the output to the data type specified by the **Output** parameter. When the input data type is floating point, the block ignores this parameter. For more details, see "Overflow Handling".

`Coefficients` — Data type of filter coefficients

`Inherit: Same word length as input` (default) | `<data type expression>`

The block casts the filter coefficients to this data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the block ignores this parameter.

The recommended data type for this parameter is `Inherit: Same word length as input`.

The block returns a warning or error if either of these conditions occur.

- The coefficients data type does not have enough fractional length to represent the coefficients accurately.
- The coefficients data type is unsigned, and the coefficients include negative values.

`Output` — Data type of filter output

`Inherit: Inherit via internal rule` (default) | `Inherit: Same word length as input` | `<data type expression>`

The block casts the output of the filter to this data type. The quantization uses the settings of the **Rounding mode** and **Overflow mode** parameters. When the input data type is floating point, the block ignores this parameter.

The block increases the word length for full precision inside each filter tap and casts the final output to the specified type. The maximum final internal data type (*WF*) depends on the input data type (*WI*), the coefficient data type (*WC*), and the number of coefficients (*L*) and is given by

$WF = WI + WC + \text{ceil}(\log2(L))$.

Because the coefficient values limit the potential growth, usually the actual full-precision internal word length is smaller than *WF*.

Control Ports

`Enable reset input port` — Option to enable reset input port

`off` (default) | `on`

Select this parameter to enable the **reset** input port. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

`Use HDL global reset` — Option to connect data path registers to generated HDL global reset signal
`off` (default) | `on`

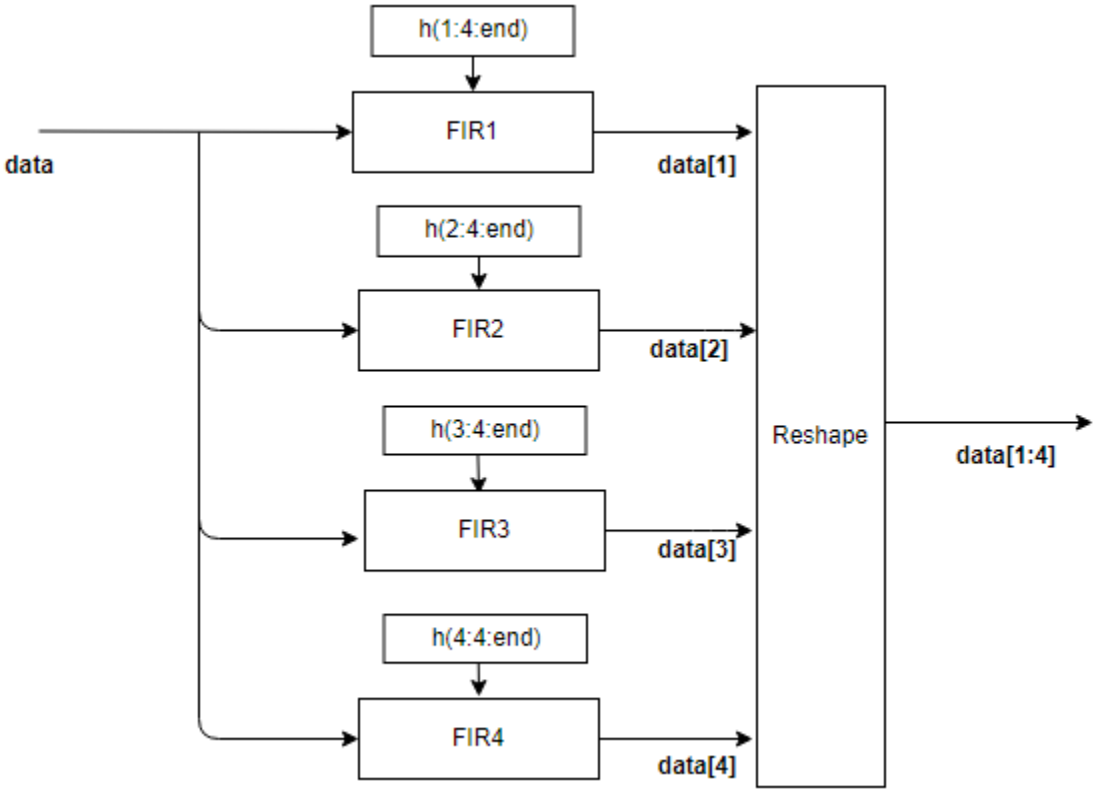
Select this parameter to connect the generated HDL global reset signal to the data path registers. This parameter does not change the appearance of the block or modify simulation behavior in Simulink. When you clear this parameter, the generated HDL global reset clears only the control path registers. 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.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

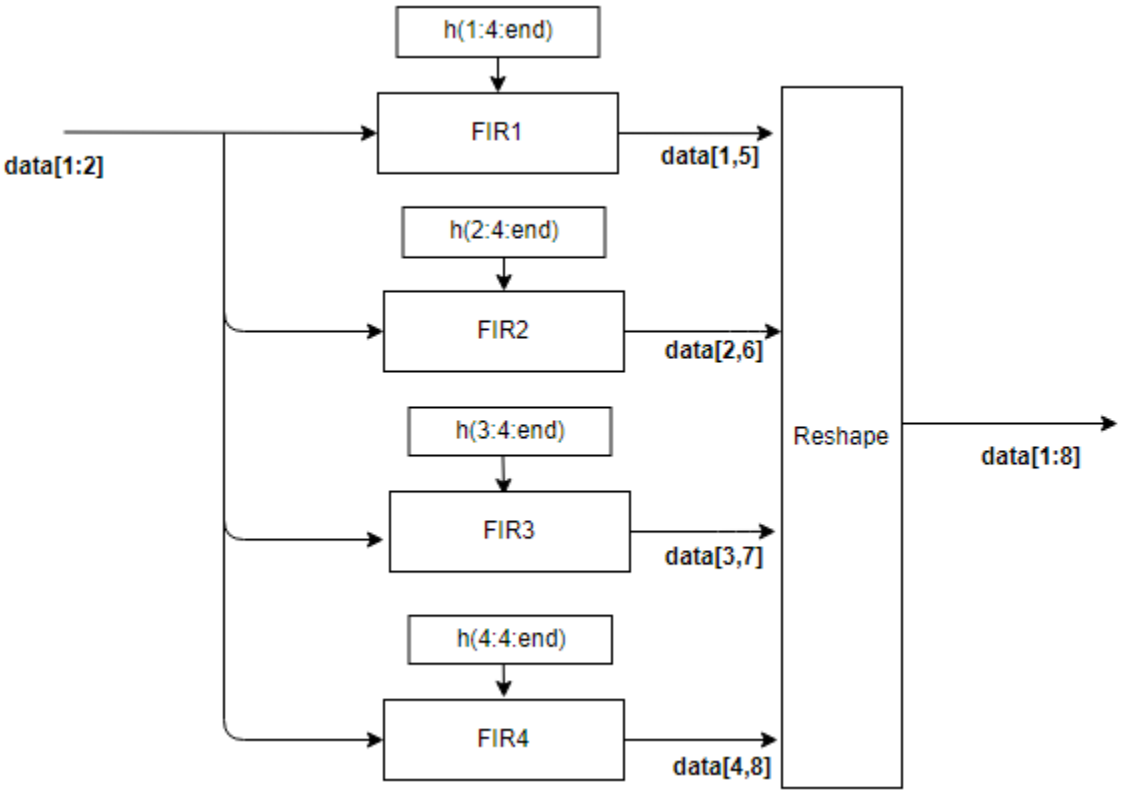
Algorithms

The block implements a polyphase filter bank where the filter coefficients are decomposed into **Interpolation factor** subfilters. If the filter length is not divisible by the **Interpolation factor** parameter value, then the block zero-pads the coefficients.

The diagram shows the polyphase filter bank with scalar input and the **Interpolation factor** parameter set to four. Each subfilter contributes one sample to the output vector. When you set **Minimum number of cycles between valid input samples** greater than 1, the block passes the *NumCycles* value to the FIR filters for each phase, and each FIR filter implements a partly-serial architectures.



The next diagram shows the polyphase filter bank for an input vector of two values and the **Interpolation factor** parameter set to four. Each of the four subfilters generates two samples of the output vector.



Each subfilter is implemented with a Discrete FIR Filter block. For architecture details, see “FIR Filter Architectures for FPGAs and ASICs”.

Performance

This table shows the post-synthesis resource utilization for the HDL code generated for the default FIR interpolation filter using scalar input, an interpolation factor of two, 16-bit input, and 16-bit coefficients. The synthesis targets a Xilinx ZC-706 (XC7Z045ffg900-2) FPGA. The **Global HDL reset type** parameter is `Synchronous`, and the **Minimize clock enables** parameter is selected. The **reset** port is disabled, so only the control path registers are connected to the generated global HDL reset.

Resource	Uses
LUT	18
FF	353
BRAM	0
Xilinx LogiCORE DSP48	72

After place and route, the maximum clock frequency of the design is 455 MHz.

For the same filter configuration but with a four-element input vector, the filter uses these resources.

Resource	Uses
LUT	2080

Resource	Uses
FF	6416
BRAM	0
Xilinx LogiCORE DSP48	288

After place and route, the maximum clock frequency of the design is 385 MHz.

For a design with scalar input, an interpolation factor of four, and **Minimum number of cycles between valid input samples** set to four, the filter uses these resources. You can see the effect of sharing filter resources over the invalid cycles between input samples.

Resource	Uses
LUT	1100
FF	2258
BRAM	0
Xilinx LogiCORE DSP48	24

After place and route, the maximum clock frequency of the design is 540 MHz.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

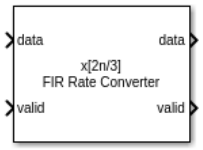
Blocks
FIR Decimator | Discrete FIR Filter

Objects
`dsphdl.FIRInterpolator`

Introduced in R2022a

FIR Rate Converter

Upsample, filter, and downsample input signal
Library: DSP HDL Toolbox / Filtering



Description

The FIR Rate Converter block upsamples, filters, and downsamples input signals. It is optimized for HDL code generation and operates on one sample of each channel at a time. The block implements a polyphase architecture to avoid unnecessary arithmetic operations and high intermediate sample rates.

The block upsamples the input signal by an integer factor of L, applies it to a FIR filter, and downsamples the input signal by an integer factor of M.

You can use the input and output control ports to pace the flow of samples. In the default configuration, the block uses input and output **valid** control signals. For additional flow control, you can enable a **ready** output signal.

The **ready** output port indicates that the block can accept a new input data sample on the next time step. When $L \geq M$, you can use the **ready** signal to achieve continuous output data samples. If you apply a new input sample after each time the block returns **ready** signal as `1`, the block returns a data output sample with the output **valid** signal set to `1` on every time step.

When you disable the **ready** port, you can apply a valid data sample only every `ceil(L/M)` time steps. For example, if:

- L/M = 4/5, then you can apply a new input sample on every time step.
- L/M = 3/2, then you can apply a new input sample on every other time step.

Ports

Input

`data` — Input data sample

scalar | row vector

Input data sample, specified as a scalar, or as a row vector in which each element represents an independent channel. The block accepts real or complex data.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

`valid` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

You can apply a valid data sample every `ceiling(L/M)` time steps.

Data Types: `Boolean`

Output

`data` — Output data sample

scalar | row vector

Output data sample, returned as a scalar or a row vector in which each element represents an independent channel.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

`ready` — Indicates block is ready for new input data

scalar

Control signal that indicates that the block is ready for new input data sample on the next cycle. When **ready** is `1` (`true`), you can specify the **data** and **valid** inputs for the next time step. When **ready** is `0` (`false`), the block ignores any input data in the next time step.

Dependencies

To enable this port, select the **Enable ready output port** checkbox.

Data Types: `Boolean`

Parameters

Main

`Interpolation factor` — Interpolation factor

3 (default) | positive integer

Specify a factor by which the block interpolates the input data sample.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Decimation factor` — Decimation factor
2 (default) | positive integer

Specify a factor by which the block decimates the input data sample.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FIR filter coefficients` — FIR filter coefficients
`firpm(70, [0 0.28 0.32 1],[1 1 0 0])` (default) | row vector

Specify a row vector of coefficients in descending powers of z^{-1}.

Note You can generate filter coefficients using Signal Processing Toolbox™ filter design functions (such as `fir1`). Design a lowpass filter with normalized cutoff frequency no greater than `min(1/L,1/M)`. The block initializes internal filter states to zero.

Data Types

`Rounding mode` — Rounding mode for fixed-point operation
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Select a rounding mode for fixed-point operations. For more information, see Rounding mode.

`Saturate on integer overflow` — Method of overflow action
`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `off` — Overflows wrap to the appropriate value that data type can represent. For example, because 130 does not fit in a signed 8-bit integer, it wraps to -126.
- `on` — Overflows saturate to either the minimum or maximum value that data type can represent. For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

`Coefficients Data Type` — FIR filter coefficients data type
`fixdt(1,16,16)` (default)

FIR filter coefficients data type, specified as a `fixdt(s,wl,fl)` object with `signedness`, `word length`, and `fractional length` properties.

`Output Data Type` — Data type of output data sample
`Inherit: Same word length as input` (default) | `Inherit via internal rule` | `fixdt(s,wl,fl)`

Specify the data type for the output data samples.

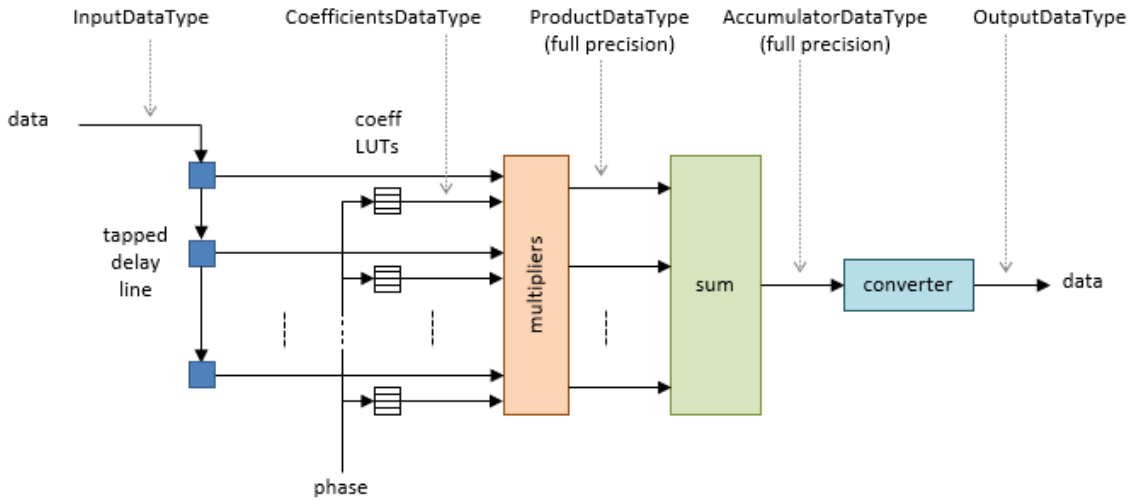
Control Ports

`Enable ready output port` — Option to enable ready control signal
`off` (default) | `on`

Select this parameter to enable the **ready** port.

Algorithms

The FIR Rate Converter block implements a fully parallel polyphase filter architecture. The diagram shows where the block casts the data types based on your configuration.



Delay

Because the block models HDL pipeline latency, an initial delay of several time steps exists before the block returns the first valid output data sample. The latency depends on the filter coefficients and the resampling factors. To determine the latency from the first input data sample to the first output data sample, measure the cycles between asserting the input **valid** signal and the output **valid** signal going high.

Performance

For an example of design performance, generate HDL for the block as configured in the “Control Data Rate Using Ready Signal” example. The example filter resamples at 5/2, and uses a symmetric 71-tap filter. The input samples and filter coefficients are 16 bits wide. The design is targeted to a Xilinx Virtex-6 FPGA, using Xilinx ISE synthesis and place and route tools.

After placement and routing, the design achieves 535 MHz clock frequency and uses these resources of the FPGA device.

LUT	592
FFS	979
Xilinx LogiCORE DSP48	15
Block RAM (16K)	0

Performance of the synthesized HDL code varies depending on your filter coefficients, FPGA target, and synthesis options.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this block was named FIR Rate Conversion HDL Optimized, and was included in the DSP System Toolbox **DSP System Toolbox HDL Support** library.

Remove request port
Behavior changed in R2022a

In previous releases, the block provided an optional **request** port. This port is no longer available. For an alternate way to control the data rate in your model, see “Control Data Rate Using Ready Signal”.

Synchronous ready signal
Behavior changed in R2022a

In previous releases, the **ready** output signal was direct feedthrough without an output pipeline register. This signal is now pipelined at the output of the block.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Blocks
FIR Rate Conversion

Objects
`dsphdl.FIRRateConverter`

Introduced in R2015b

FIR Rate Converter (Obsolete)

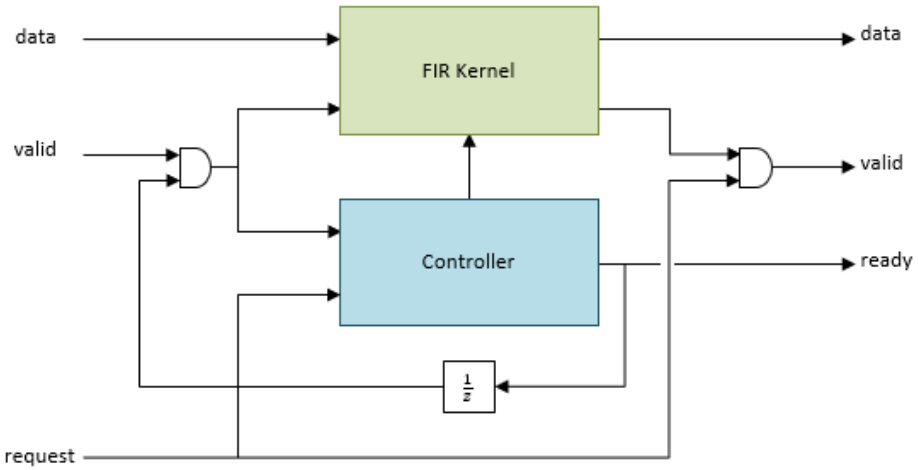
Upsample, filter, and downsample input signal (with request port)

Description

The FIR Rate Converter block upsamples, filters, and downsamples input signals. It is optimized for HDL code generation and operates on one sample of each channel at a time. The block implements a polyphase architecture to avoid unnecessary arithmetic operations and high intermediate sample rates.

The block upsamples the input signal by an integer factor of L, applies it to a FIR filter, and downsamples the input signal by an integer factor of M.

You can use the input and output control ports for pacing the flow of samples. In the default configuration, the block uses input and output **valid** control signals. The block has a **request** port to control the output rate. For input flow control, you can enable a **ready** output signal.



The **ready** output port indicates that the block can accept a new input data sample on the next time step. When $L \geq M$, you can use the **ready** signal to achieve continuous output data samples. If you apply a new input sample after each time the block returns **ready** signal as `1`, the block returns a data output sample with the output **valid** signal set to 1 on every time step.

When you disable the **ready** port, you can apply a valid data sample only every `ceil(L/M)` time steps. For example, if:

- $L/M = 4/5$, then you can apply a new input sample on every time step.

- L/M = 3/2, then you can apply a new input sample on every other time step.

The block returns the next output sample one cycle after the **request** signal is `1` and a valid output sample is available. When no new data is available, block sets the output **valid** signal to `0`.

You can connect the **request** input port to the **ready** output port of a downstream block.

Ports

Input

`data` — Input data sample

scalar | row vector

Input data sample, specified as a scalar, or as a row vector in which each element represents an independent channel. The block accepts real or complex data.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

`valid` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

You can apply a valid data sample every `ceiling(L/M)` time steps.

Data Types: `Boolean`

`request` — Request control signal

scalar

When the **request** port is `1`, and an output data sample is available, on the next cycle the block returns that output data sample on the output **data** port and sets the output **valid** signal to `1`. When no new data is available, the block sets the output **valid** signal to `0`. When the **request** port is `0`, the block holds available data until the **request** port is set to `1`.

You can connect the **request** input port to the **ready** output port of a downstream block.

Data Types: `Boolean`

Output

`data` — Output data sample

scalar | row vector

Output data sample, returned as a scalar or a row vector in which each element represents an independent channel.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

`ready` — Indicates block is ready for new input data

scalar

Control signal that indicates that the block is ready for new input data sample on the next cycle. When **ready** is `1` (`true`), you can specify the **data** and **valid** inputs for the next time step. When **ready** is `0` (`false`), the block ignores any input data in the next time step.

You can connect the **ready** output port to the **request** input port of an upstream block.

Dependencies

To enable this port, select the **Enable ready output port** checkbox.

Data Types: `Boolean`

Parameters

Main

`Interpolation factor` — Interpolation factor

3 (default) | positive integer

Specify a factor by which the block interpolates the input data sample.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Decimation factor` — Decimation factor

2 (default) | positive integer

Specify a factor by which the block decimates the input data sample.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FIR filter coefficients` — FIR filter coefficients

`firpm(70, [0 0.28 0.32 1],[1 1 0 0])` (default) | row vector

Specify a row vector of coefficients in descending powers of z^{-1}.

Note You can generate filter coefficients using Signal Processing Toolbox filter design functions (such as `fir1`). Design a lowpass filter with normalized cutoff frequency no greater than `min(1/L,1/M)`. The block initializes internal filter states to zero.

Data Types

`Rounding mode` — Rounding mode for fixed-point operation

`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Select a rounding mode for fixed-point operations. For more information, see Rounding mode.

`Saturate on integer overflow` — Method of overflow action
`off` (default) | `on`

Specify whether overflows saturate or wrap.

- `off` — Overflows wrap to the appropriate value that data type can represent. For example, because 130 does not fit in a signed 8-bit integer, it wraps to -126.
- `on` — Overflows saturate to either the minimum or maximum value that data type can represent. For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

`Coefficients Data Type` — FIR filter coefficients data type
`fixdt(1,16,16)` (default)

FIR filter coefficients data type, specified as a `fixdt(s,wl,fl)` object with `signedness`, `word length`, and `fractional length` properties.

`Output Data Type` — Data type of output data sample
`Inherit: Same word length as input` (default) | `Inherit via internal rule` | `fixdt(s,wl,fl)`

Specify the data type for the output data samples.

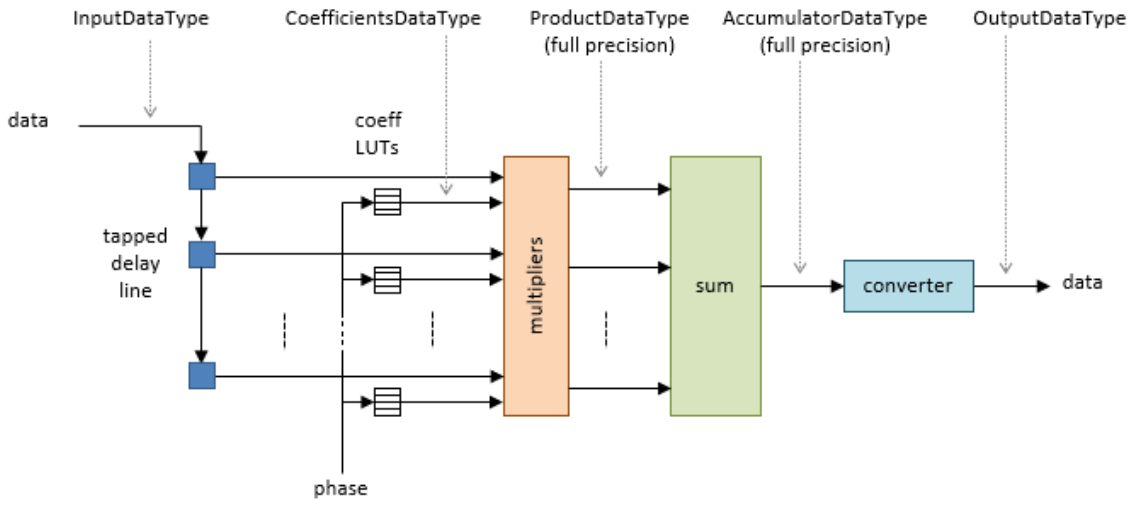
Control Ports

`Enable ready output port` — Option to enable ready control signal
`off` (default) | `on`

Select this parameter to enable the **ready** port.

Algorithms

The FIR Rate Converter block implements a fully parallel polyphase filter architecture. The diagram shows where the block casts the data types based on your configuration.



Delay

Because the block models HDL pipeline latency, an initial delay of several time steps exists before the block returns the first valid output data sample. The latency depends on the filter coefficients and the resampling factors. To determine the latency from the first input data sample to the first output data sample, measure the cycles between asserting the input **valid** signal and the output **valid** signal going high.

Performance

For an example of design performance, generate HDL for the block as configured in the “Control Data Rate Using Ready Signal” example. The example filter resamples at 5/2, and uses a symmetric 71-tap filter. The input samples and filter coefficients are 16 bits wide. The design is targeted to a Xilinx Virtex-6 FPGA, using Xilinx ISE synthesis and place and route tools.

After placement and routing, the design achieves 535 MHz clock frequency and uses these resources of the FPGA device.

LUT	592
FFS	979
Xilinx LogiCORE DSP48	15
Block RAM (16K)	0

Performance of the synthesized HDL code varies depending on your filter coefficients, FPGA target, and synthesis options.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox

Behavior changed in R2022a

Before R2022a, this block was named FIR Rate Conversion HDL Optimized, and was included in the DSP System Toolbox **DSP System Toolbox HDL Support** library.

Remove request port in future release

Behavior changed in R2022a

The **request** port on the FIR Rate Converter block is no longer supported. Existing instances of the FIR Rate Converter block that use the **request** port are forwarded to this block, which has a nonoptional request port. This block will be removed in a future release.

Instead, use the FIR Rate Converter block without the **request** port, and control the input data rate by using a FIFO outside the block. See “Control Data Rate Using Ready Signal”.

One cycle latency between request and validOut

Behavior changed in R2022a

This block now has a register on the **request** input port. This register means there is one cycle latency between setting **request** to `1` (`true`) and the block returning data on the **dataOut** and **validOut** ports.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

See Also

Blocks

FIR Rate Converter

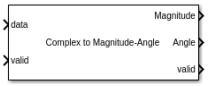
Objects

`dsphdl.FIRRateConverter`

Introduced in R2015b

Complex to Magnitude-Angle

Compute magnitude and phase angle of complex signal using CORDIC algorithm
Library: DSP HDL Toolbox / Math Functions



Description

The Complex to Magnitude-Angle block computes the magnitude and phase angle of a complex signal and provides hardware-friendly control signals. To achieve an efficient HDL implementation, the block uses a pipelined Coordinate Rotation Digital Computer (CORDIC) algorithm.

You can use this block to implement operations such as `atan2` in hardware.

Ports

Input

`data` — Complex input signal
scalar | vector

Complex input signal, specified as a scalar, a column vector representing samples in time, or a row vector representing channels. Using vector input increases data throughput while using more hardware resources. The block implements the conversion logic in parallel for each element of the vector. The input vector can contain up to 64 elements.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

`valid` — Indicates valid output data
scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

Output

`Magnitude` — Magnitude of the input signal
scalar | vector

Magnitude of the input signal, returned as a scalar, a column vector representing samples in time, or a row vector representing channels. The dimensions of this port match the dimensions of the input **data** port.

Dependencies

To enable this port, set the **Output format** parameter to `Magnitude and Angle` or `Magnitude`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt([],N,0)`

`Angle` — Angle of the input signal

scalar | vector

Angle of the input signal, returned as a scalar, a column vector representing samples in time, or a row vector representing channels. The dimensions of this port match the dimensions of the input **data** port.

Dependencies

To enable this port, set the **Output format** parameter to `Magnitude and Angle` or `Angle`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt([],N,0)`

`valid` — Indicates valid output data

scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the **Magnitude** and/or **Angle** ports. When **valid** is `0` (`false`), the values from the **Magnitude** and/or **Angle** ports are not valid.

Data Types: `Boolean`

Parameters

`Number of iterations source` — Source of number of iterations

`Auto` (default) | `Property`

- To set the number of iterations to input *WL* − 1, select `Auto`. If the input is of data type `double` or `single`, the number of iterations is set to 16, by default.
- To specify the number of iterations by using **Number of iterations** parameter, select `Property`.

`Number of iterations` — Number of CORDIC iterations

positive integer

The number of iterations must be less than or equal to input *WL* − 1. The latency of the block depends on the number of iterations performed. For information about latency, see “Latency” on page 1-132.

Dependencies

To enable this parameter, set **Number of iterations source** to `Property`.

`Output format` — Output signal format

`Magnitude and Angle` (default) | `Magnitude` | `Angle`

Use this parameter to specify which output ports are enabled.

- To enable the **Magnitude** and **Angle** output ports, select `Magnitude and Angle` (default).

- To enable the **Magnitude** output port and disable the **Angle** output port, select `Magnitude`.
- To enable the **Angle** output port and disable the **Magnitude** output port, select `Angle`.

`Angle format` — Output angle format
`Normalized` (default) | `Radians`

- To return the **Angle** output as a fixed-point value that normalizes the angles in the range [–1,1], select `Normalized`. For more information see “Normalized Angle Format” on page 1-131.
- To return the **Angle** output as a fixed-point value in the range [-π, π], select `Radians`. When using this block to implement the `atan2` function, set this parameter to `Radians`.

`Scale output` — Scales output
`on` (default) | `off`

Select this parameter to multiply the **Angle** output by the inverse of the CORDIC gain factor. The block implements this gain factor with either CSD logic or a multiplier, according to the **Scaling method** parameter.

Note If you clear this parameter and apply the CORDIC gain elsewhere in your design, you must exclude the π/4 term. Because the quadrant mapping algorithm replaces the first CORDIC iteration by mapping inputs onto the angle range [0, π/4], the initial rotation does not contribute a gain term. The gain factor is the product of $\cos(\tan(2^{-n}))$, for n from 1 to **Number of iterations** – 1.

`Scaling method` — Implementation of CORDIC gain scaling
`CSD` (default) | `Multipliers`

When you set this parameter to `CSD`, the block implements the CORDIC gain scaling by using a shift-and-add architecture for the multiply operation. This implementation uses no multiplier resources and may increase the length of the critical path in your design. When you select `Multipliers`, the block implements the CORDIC gain scaling with a multiplier and increases the latency of the block by four cycles.

Dependencies

To enable this parameter, select the **Scale output** parameter.

Algorithms

CORDIC Algorithm

The CORDIC algorithm is a hardware-friendly method for performing trigonometric functions. It is an iterative algorithm that approximates the solution by converging toward the ideal point. The block uses CORDIC vectoring mode to iteratively rotate the input onto the real axis.

Givens method for rotating a complex number $x+iy$ by an angle θ is as follows. The direction of rotation, d, is +1 for counterclockwise and −1 for clockwise.

$$x_r = x\cos\theta - dy\sin\theta$$
$$y_r = y\cos\theta + dx\sin\theta$$

For a hardware implementation, factor out the $cos\theta$ to leave a $tan\theta$ term.

$$x_r = \cos\theta(x - dy\tan\theta)$$
$$y_r = \cos\theta(y + dx\tan\theta)$$

To rotate the vector onto the real axis, choose a series of rotations of θ_n so that $\tan\theta_n = 2^{-n}$. Remove the *cosθ* term so each iterative rotation uses only shift and add operations.

$$Rx_n = x_{n-1} - d_n y_{n-1} 2^{-n}$$
$$Ry_n = y_{n-1} + d_n x_{n-1} 2^{-n}$$

Combine the missing *cosθ* terms from each iteration into a constant, and apply it with a single multiplier to the result of the final rotation. The output magnitude is the scaled final value of *x*. The output angle, *z*, is the sum of the rotation angles.

$$x_r = (\cos\theta_0\cos\theta_1 ... \cos\theta_n) Rx_N$$
$$z = \sum_0^N d_n\theta_n$$

Modified CORDIC Algorithm

The convergence region for the standard CORDIC rotation is ≈±99.7°. To work around this limitation, before doing any rotation, the block maps the input into the [0, π/4] range using this algorithm.

```
if abs(x) > abs(y)
  input_mapped = [abs(x), abs(y)];
else
  input_mapped = [abs(y), abs(x)];
end
```

At each iteration, the block rotates the vector towards the real axis. The rotation is counterclockwise when *y* is negative, and clockwise when *y* is positive.

Quadrant mapping saves hardware resources and reduces latency by reducing the number of CORDIC pipeline stages by one. The CORDIC gain factor, *Kn*, therefore does not include the *n=0*, or *cos(π/4)*term.

$$K_n = cos\theta_1 ... \cos\theta_n = \cos(26.565) \cdot \cos(14.036) \cdot \cos(7.125) \cdot \cos(3.576)$$

After the CORDIC iterations are complete, the block adjusts the angle back to its original location. First it adjusts the angle to the correct side of π/4.

```
if abs(x) > abs(y)
  angle_unmapped = CORDIC_out;
else
  angle_unmapped = (pi/2) - CORDIC_out;
end
```

Then, the block flips the angle to the original quadrant.

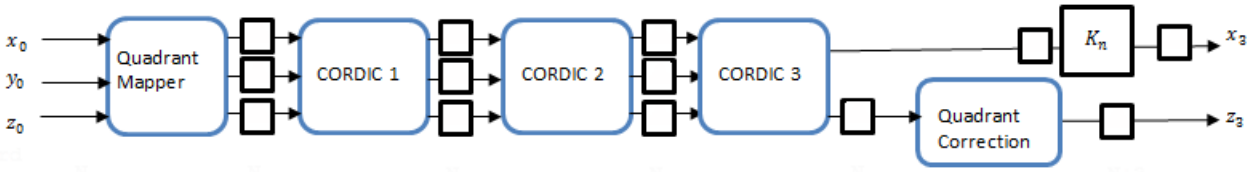
```
if (x < 0)
  if (y < 0)
    output_angle = - pi + angle_unmapped;
  else
    output_angle = pi - angle_unmapped;
else
```

```
if (y<0)
  output_angle = -angle_unmapped;
```

Architecture

The block generates a pipelined HDL architecture to maximize throughput. Each CORDIC iteration is done in one pipeline stage. The gain factor, if enabled, is implemented with canonical signed digit (CSD) logic by default. Set the **Scaling method** parameter to `Multipliers` to implement the gain factor with a multiplier.

If you use vector input, this block replicates this architecture in parallel for each element of the vector.

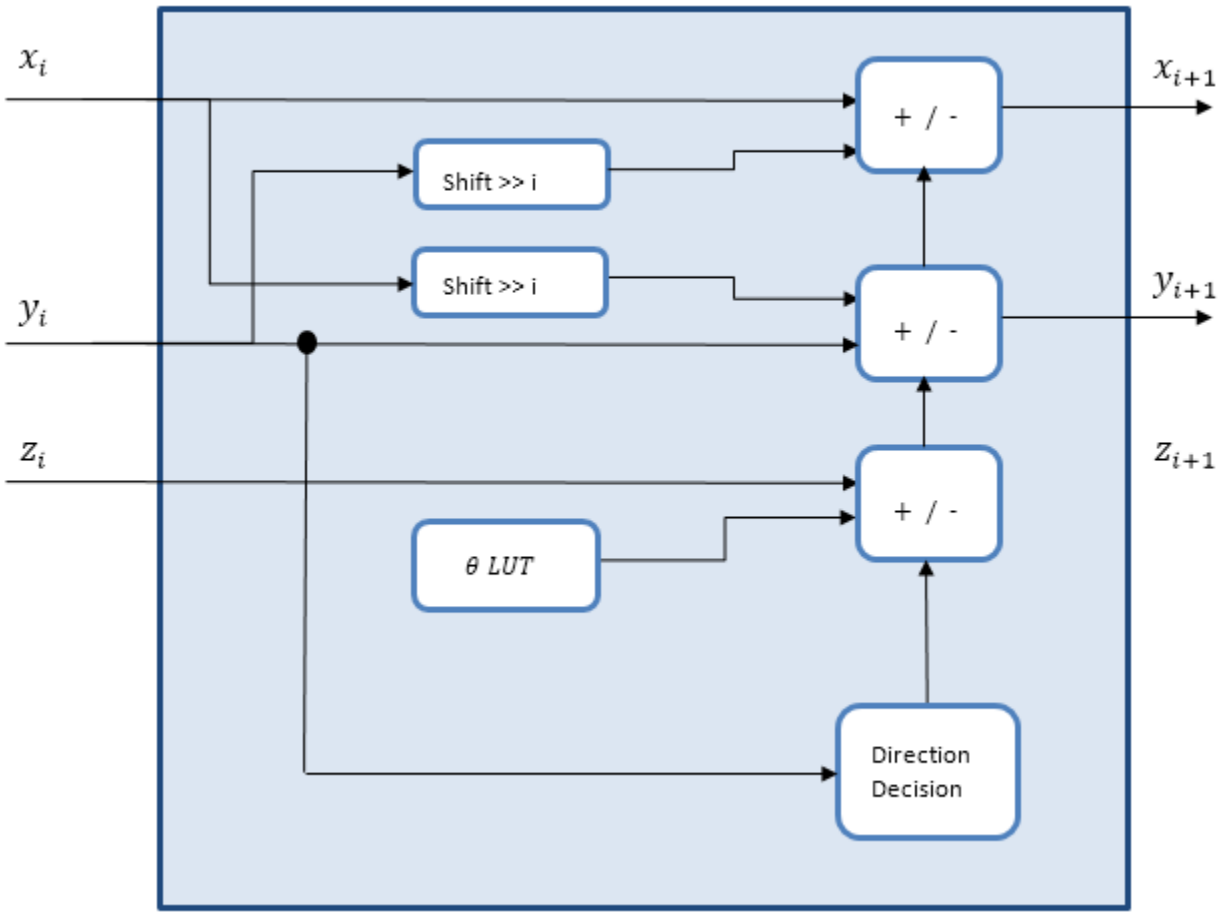


The following table shows **Magnitude** and **Angle** output word length (WL), for particular input word length (WL). FL stands for fractional length used in fixed-point representation.

Input Word Length	Output Magnitude Word Length
fixdt(0,WL,FL)	fixdt(0,WL + 2,FL)
fixdt(1,WL,FL)	fixdt(1,WL + 1,FL)

Input Word Length	Output Angle Word Length	
fixdt([ ],WL,FL)	Radians	fixdt(1,WL + 3,WL)
	Normalized	fixdt(1,WL + 3,WL+2)

The CORDIC logic at each pipeline stage implements one iteration. For each pipeline stage, the shift and angle rotation are constants.

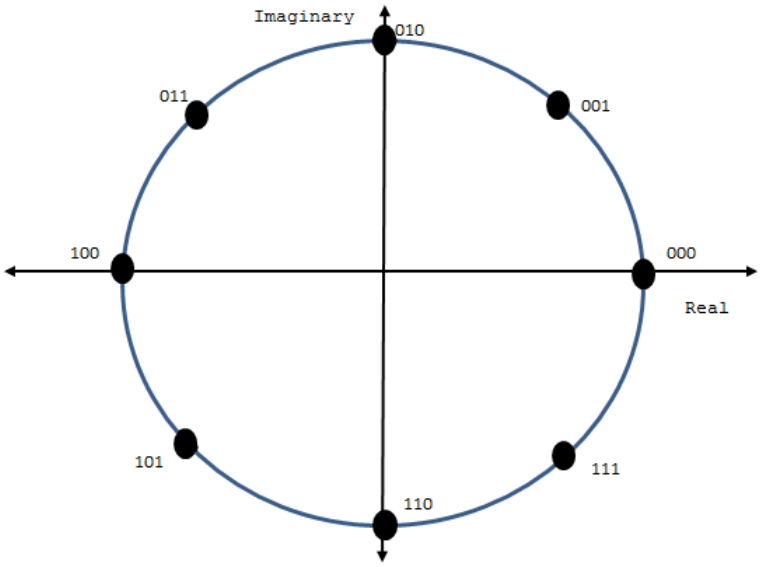


When you set **Output format** to `Magnitude`, the block does not generate HDL code for the angle accumulation and quadrant correction logic.

Normalized Angle Format

This format normalizes the fixed-point radian angle values around the unit circle. This use of bits can be more efficient than the use of the range [0, 2π] radians. Also this normalized angle format enables wraparound of angle at 0 or 2π without additional detect and correct logic.

For example, representing the angle with 3 bits results in these normalized values.



The block normalizes the angles across [0, π/4] and maps them to the correct octant at the end of the calculation.

Latency

When the valid input is applied, the block returns valid output after **Number of iterations** + 4 cycles. The latency is displayed on the block mask.

When you set the **Number of iterations source** parameter to `Property`, the block shows the latency immediately. When you set **Number of iterations source** to `Auto`, the block calculates the latency based on the input port data type and displays the latency when you update the model.

When you set the **Number of iterations source** parameter to `Auto`, the number of iterations is input $WL - 1$, and the latency is input $WL + 3$. If the input is of data type `double` or `single`, the number of iterations is 16, and the latency is 20.

Note When you set the **Scaling method** parameter to `Multipliers`, the block latency increases by four cycles.

Performance

Performance was measured for the default configuration, with output scaling disabled and `fixdt(1,16,12)` input. When the generated HDL code is synthesized into a Xilinx ZC706 (XC7Z045FFG900-2) FPGA, the design achieves 350 MHz clock frequency. It uses the following resources.

Resource	Number Used
LUT	891
FFS	899
Xilinx LogiCORE DSP48	0

Resource	Number Used
Block RAM (16K)	0
Critical path	2.792 ns

When you use a multiplier for the CORDIC gain scaling, the design uses one DSP block and has a shorter critical path. The critical path difference is not significant at this number of bits, but as the size of the data increases, the critical path of the CSD implementation rises faster than the critical path of the multiplier.

Resource	Number Used
LUT	808
FFS	956
Xilinx LogiCORE DSP48	1
Block RAM (16K)	0
Critical path	2.574 ns

Performance of the synthesized HDL code varies depending on your target and synthesis options. When you use vector input, the resource usage is about *VectorSize* times the scalar resource usage.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox

Behavior changed in R2022a

Before R2022a, this block was named Complex to Magnitude-Angle HDL Optimized, and was included in the DSP System Toolbox **DSP System Toolbox HDL Support** library.

Option to use multiplier for scale factor

In previous releases, the block implemented the CORDIC gain for hardware by using shift-and-add logic. To use a multiplier, set the **Scaling method** parameter to `Multipliers`. To use shift-and-add logic, set this property to `CSD`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration option that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Complex Data Support

This block supports code generation for complex signals.

See Also

Complex to Magnitude-Angle | `dsphdl.ComplexToMagnitudeAngle` | `atan2`

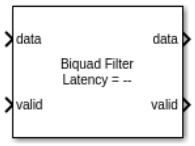
Topics

"HDL QAM Transmitter and Receiver" (Communications Toolbox)

Introduced in R2014b

Biquad Filter

Biquadratic IIR (SOS) filter
Library: DSP HDL Toolbox / Filtering



Description

A biquad filter is a form of infinite-impulse response (IIR) filter where the numerator and denominator are split into a series of second-order sections connected by gain blocks. This type of filter can replace a large FIR filter that uses an impractical amount of hardware resources. Designs often use biquad filters as DC blocking filters or to meet a specification originally implemented with an analog filter, such as a pre-emphasis filter.

Ports

Input

`data` — Input data
scalar or column vector of real values

Input data, specified as a scalar or column vector of real values. When the input has an integer or fixed-point data type, the block uses fixed-point arithmetic for internal calculations.

Vector input is supported only when you set **Filter structure** to `Pipelined feedback form`. The block accepts vectors up to 64 samples, but large vector sizes can make the calculation of internal data types challenging. Vector sizes of up to 16 samples are practical for hardware implementation.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fixed point` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`valid` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When **valid** is `1` (`true`), the block captures the values from the input **data** port. When **valid** is `0` (`false`), the block ignores the values from the input **data** port.

Data Types: `Boolean`

Output

`data` — Filtered output data
scalar or column vector of real values

Filtered output data, returned as a scalar or column vector of real values. The output dimensions match the input dimensions. When the input data type is a floating-point type, the output data

inherits the data type of the input data. When the input data type is an integer type or a fixed-point type, the **Output** parameter on the **Data Types** tab controls the output data type.

Data Types: `fixed point` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`valid` — Indicates valid output data
scalar

Control signal that indicates if the data from the output **data** port is valid. When **valid** is `1` (`true`), the block returns valid data from the output **data** port. When **valid** is `0` (`false`), the values from the output **data** port are not valid.

Data Types: `Boolean`

Parameters

Main

`Filter structure` — HDL filter architecture
`Direct form II` (default) | `Direct form II transposed` | `Pipelined feedback form`

Both the `Direct form II` and `Direct form II transposed` architectures are pipelined and quantized to fit well into FPGA DSP blocks. The output of these filters matches the output of the DSP System Toolbox System objects `dsp.SOSFilter` and `dsp.FourthOrderSectionFilter`. These architectures minimize the number of multipliers used by the filter but have a critical path through the feedback loop and sometimes cannot achieve higher clock rates.

`Pipelined feedback form` implements a pipelined architecture that uses more multipliers than either direct-form II structure, but achieves higher clock rates after synthesis. Frame-based input is supported only when you use `Pipelined feedback form`. The output of the pipelined filter is slightly different than the DSP System Toolbox functions `dsp.SOSFilter` and `dsp.FourthOrderSectionFilter` because of the timing of data samples applied in the pipelined filter stages.

`Numerator coefficients of filter` — Coefficients for numerator
`[1,2,1]` (default) | *NumSections*-by-3 matrix

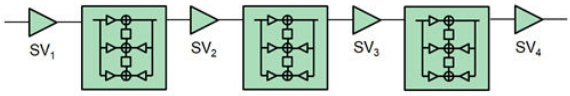
Specify the numerator coefficients as a matrix of *NumSections*-by-3 values. *NumSections* is the number of second-order filter sections. The block infers the number of filter sections from the size of the numerator and denominator coefficients. The numerator coefficient and denominator coefficient matrices must be the same size. The default filter has one section.

`Denominator coefficients of filter` — Coefficients for denominator
`[1,.1,.2]` (default) | *NumSections*-by-3 matrix

Specify the denominator coefficients as a matrix of *NumSections*-by-3 values. The block assumes the first denominator coefficient of each section is `1.0`. *NumSections* is the number of second-order filter sections. The block infers the number of sections from the size of the numerator and denominator coefficients. The numerator coefficient and denominator coefficient matrices must be the same size. The default filter has one section.

`Scale values of filter` — Gain values applied before and after second-order filter sections
`[1]` (default) | vector of 1 to *NumSections*+1 values

Specify the gain values as a vector of up to *NumSections*+1 values. *NumSections* is the number of second-order filter sections. The block infers the number of sections from the size of the numerator and denominator coefficients. If the vector has only one value, the block applies that gain before the first section. If you specify fewer values than there are filter sections, the block sets the remaining section gain values to one. The diagram shows a 3-section filter and the locations of the four scale values before and after the sections.



Implementing these gain factors outside the filter sections reduces the multipliers needed to implement the numerator of the filter.

Data Types

`Rounding mode` — Rounding mode for type-casting the output
`Floor` (default) | `Ceiling` | `Convergent` | `Nearest` | `Round` | `Zero`

Rounding mode for type-casting the output and accumulator values to the data types specified by the **Output** and **Accumulator** parameters. When the input data type is floating point, the block ignores this parameter. For more details, see “Rounding Modes”.

`Saturate on integer overflow` — Overflow handling for type-casting the output
`off` (default) | `on`

Overflow handling for type-casting the output and accumulator values to the data types specified by the **Output** and **Accumulator** parameters. When the input data type is floating point, the block ignores this parameter. For more details, see “Overflow Handling”.

`Numerator` — Data type of numerator coefficients
`Inherit: Same word length as first input` (default) | `<data type expression>`

The block casts the numerator coefficients to this data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the block ignores this parameter.

The block returns a warning if the data type of the coefficients does not have enough fractional length to represent the coefficients accurately.

`Denominator` — Data type of denominator coefficients
`Inherit: Same word length as first input` (default) | `<data type expression>`

The block casts the denominator coefficients to this data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the block ignores this parameter.

The block returns a warning if the data type of the coefficients does not have enough fractional length to represent the coefficients accurately.

`ScaleValues` — Data type of gain
`Inherit: Same word length as first input` (default) | `<data type expression>`

The block casts the scale values to this data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the block ignores this parameter.

`Accumulator` — Data type of accumulator signals within each section
`Inherit: Same as first input` (default) | `<data type expression>`

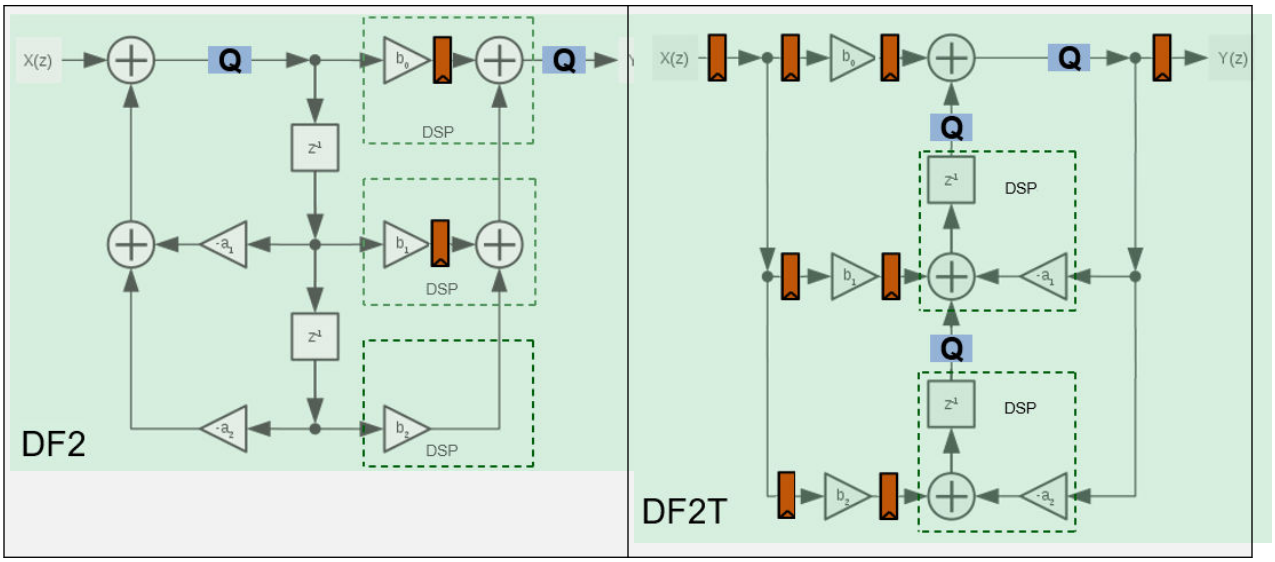
The block casts the internal accumulator signals (as indicated in the diagrams in the “Algorithms” on page 1-138 section) to this data type. The quantization uses the settings of the **Rounding mode** and **Saturate on integer overflow** parameters. When the input data type is floating point, the block ignores this parameter.

`Output` — Data type of filter output
`Inherit: Inherit via internal rule` (default) | `Inherit: Same as first input` | `<data type expression>`

The block casts the output of the filter to this data type. The quantization uses the settings of the **Rounding mode** and **Saturate on integer overflow** parameters. When the input data type is floating point, the block ignores this parameter.

Algorithms

The direct form II and direct form II transposed architectures are pipelined and quantized to fit well into FPGA DSP blocks. The architecture diagrams show the accumulator quantization points and pipeline registers for the direct form II and direct form II transposed architectures. The dashed lines indicate how the multiply-add operation fits into a DSP block on an FPGA.



The transposed biquad filter implementations have a long critical path through the feedback loop between poles z^{-1} and z^{-2}. They use the lowest number of multipliers but are not suitable for high-speed applications.

The pipelined biquad architecture uses more multipliers but reduces the critical path and achieves higher clock rates. The filter uses the Parhi method [1], which adds pipeline registers by increasing the order of the denominator, and then compensates for the modified denominator by adding poles in the numerator. The pipelined denominator uses order z^{-4} and z^{-8} rather than z^{-1} and z^{-2}. Since the

poles are less than one for a stable filter, the poles from the higher orders are smaller than the starting values, which adds to filter stability.

To support frame-based input, the pipelined architecture increases the order of the denominator to the power of the input size. A frame-based pipelined denominator uses z^{-N*4} and z^{-N*8}, where N is the size of the input vector. The new numerator that compensates for the additional poles in the denominator has `2*N*4-1` coefficients. This implementation uses more resources but supports high-throughput applications.

Performance

For a comparison of hardware resource use and synthesized clock speed of the three biquad filter architectures, see “High Performance DC Blocker for FPGA”.

References

[1] Chung, Jin-Gyun, and Keshab K. Parhi. *Pipelined Lattice and Wave Digital Recursive Filters*. Boston: Springer US, 1996. https://doi.org/10.1007/978-1-4613-1307-6.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block supports C/C++ code generation for Simulink accelerator and rapid accelerator modes and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Discrete FIR Filter | `dsphdl.BiquadFilter`

Introduced in R2022a

System Objects

dsphdl.CICInterpolator

Package: `dsphdl`

Interpolate signal using CIC filter

Description

The `dsphdl.CICInterpolator` System object™ interpolates an input signal by using a cascaded integrator-comb (CIC) interpolation filter. CIC interpolation filters are a class of linear phase finite impulse response (FIR) filters consisting of a comb part and an integrator part. The CIC interpolation filter structure consists of *N* sections of cascaded comb filters, a rate change factor of *R*, and *N* sections of cascaded integrators. For more information about CIC interpolation filters, see “Algorithms” on page 2-11.

The System object supports these combinations of input and output data.

- Scalar input and scalar output — Support for fixed and variable interpolation rates
- Scalar input and vector output — Support for fixed interpolation rates only
- Vector input and vector output — Support for fixed interpolation rates only

The System object provides an architecture suitable for HDL code generation and hardware deployment.

The System object supports real and complex fixed-point inputs.

To filter input data with a CIC interpolator filter, follow these steps:

1. Create the `dsphdl.CICInterpolator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
cicIntFilt = dsphdl.CICInterpolator
cicIntFilt = dsphdl.CICInterpolator(Name,Value)
```

Description

`cicIntFilt = dsphdl.CICInterpolator` creates a CIC interpolator filter System object, `cicIntFilt`, with default properties.

`cicIntFilt = dsphdl.CICInterpolator(Name,Value)` creates the filter with properties set using one or more name-value arguments. Enclose each property name in single quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`InterpolationSource` — Source of interpolation factor

`'Property'` (default) | `'Input port'`

Specify whether the System object operates with a fixed or variable interpolation rate.

- `'Property'` — Use a fixed interpolation rate specified by the `InterpolationFactor` property.
- `'Input port'` — Use a variable interpolation rate specified by the R input argument.

Note The System object does not support variable interpolation for these two combinations of input and output:

- Scalar input and vector output
- Vector input and vector output

`InterpolationFactor` — Interpolation factor

2 (default) | integer from 1 to 2048

Specify the interpolation factor as an integer from 1 to 2048. The range of available values depends on the type of input and output data. This value gives the rate at which the System object interpolates the input.

Input Data	Output Data	`InterpolationFactor` Valid Values
Scalar	Scalar	Integer from 1 to 2048
Scalar	Vector	Integer from 1 to 64
Vector	Vector	Integer from 1 to 64

Note For vector inputs, select the interpolation factor rate and input vector length such that their multiplication value does not exceed 64.

Dependencies

To enable this property, set the InterpolationSource property to `'Property'`.

`MaxInterpolationFactor` — Upper bound of variable interpolation factor

2 (default) | integer from 1 to 2048

Specify the upper bound of the range of valid values for the R input argument as an integer from 1 to 2048.

Note The System object does not support variable interpolation for these two combinations of input and output:

- Scalar input and vector output
- Vector input and vector output

Dependencies

To enable this property, set the InterpolationSource property to `'Input port'`.

`DifferentialDelay` — Differential delay

1 (default) | 2

Specify the differential delay of the comb part of the filter as either 1 or 2 cycles.

`NumSections` — Number of integrator and comb sections

2 (default) | 1 | 3 | 4 | 5 | 6

Specify the number of sections in either the comb part or the integrator part of the System object.

`NumCycles` — Minimum number of cycles between valid input samples

1 (default) | factors or multiples of *R*

Specify the minimum number of cycles between the valid input samples as 1, factors of *R*, or multiples of *R* based on the type of input and output data, where *R* is the interpolation factor.

Input Data	Output Data	Minimum Number of Cycles Between Valid Input Samples
Scalar	Scalar	greater than or equal to *R*
Scalar	Vector	factors less than *R*
Vector	Vector	1

Dependencies

To enable this property, set the InterpolationSource property to `'Property'`.

`GainCorrection` — Output gain compensation

`false` (default) | `true`

Set this property to `true` to compensate for the output gain of the filter.

The latency of the System object changes depending on the type of interpolation you specify, the number of sections, and the value of this property. For more information on the latency of the System object, see “Latency” on page 2-14.

`OutputDataType` — Data type of output

`'Full precision'` (default) | `'Same word length as input'` | `'Minimum section word lengths'`

Choose the data type of the filtered output data.

- `'Full precision'` — The output data type has a word length equal to the input word length plus gain bits.

- `'Same word length as input'` — The output data type has a word length equal to the input word length.
- `'Minimum section word lengths'` — The output data type uses the word length you specify in the OutputWordLength property.

`OutputWordLength` — Word length of output
16 (default) | integer from 2 to 104

Word length of the output, specified as an integer from 2 to 104.

Dependencies

To enable this property, set the OutputDataType property to `'Minimum section word lengths'`.

`ResetInputPort` — Reset argument
`false` (default) | `true`

When you set this property to `true`, the System object expects a `reset` input argument.

Usage

Syntax

```
[dataOut,validOut] = cicIntFilt(dataIn,validIn)
[dataOut,validOut] = cicIntFilt(dataIn,validIn,R)
[dataOut,validOut] = cicIntFilt(dataIn,validIn,reset)
[dataOut,validOut] = cicIntFilt(dataIn,validIn,R,reset)
```

Description

`[dataOut,validOut] = cicIntFilt(dataIn,validIn)` filters and interpolates the input data using a fixed interpolation factor only when `validIn` is `true`.

`[dataOut,validOut] = cicIntFilt(dataIn,validIn,R)` filters the input data using the specified variable interpolation factor `R`. The `InterpolationSource` property must be set to `'Input port'`.

`[dataOut,validOut] = cicIntFilt(dataIn,validIn,reset)` filters the input data when `reset` is `false` and clears filter internal states when `reset` is `true`. The System object expects the `reset` argument only when you set the ResetIn property to `true`.

`[dataOut,validOut] = cicIntFilt(dataIn,validIn,R,reset)` filters the input data when `reset` is `false` and clears filter internal states when `reset` is `true`. The System object expects the `reset` argument only when you set the ResetIn property to `true`. The `InterpolationSource` property must be set to `'Input port'`.

Input Arguments

`dataIn` — Input data
scalar | column vector

Specify input data as a scalar or a column vector with a length from 1 to 64. The input data must be a signed integer or signed fixed point with a word length less than or equal to 32.

Data Types: `int8` | `int16` | `int32` | `fi`
Complex Number Support: Yes

`validIn` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

`R` — Variable interpolation rate

scalar

Use this argument to dynamically specify the variable interpolation rate during run time.

This value must have the data type `fi(0,12,0)` and must be an integer in the range from 1 to the `MaxInterpolationFactor` property value.

Dependencies

To enable this argument, set the InterpolationSource property to `'Input port'`.

Data Types: `fi(0,12,0)`

`reset` — Clears internal states

logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the object captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set the ResetIn property to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — CIC-interpolated output data

scalar | column vector

CIC-interpolated output data, returned as a scalar or a column vector with a length from 1 to 64.

The OutputDataType property sets the data type of this argument.

Data Types: `int8` | `int16` | `int32` | `fi`
Complex Number Support: Yes

`validOut` — Indicates valid output data

scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

`ready` — Indicates object is ready for new input data
logical scalar

Control signal that indicates that the object is ready for new input data sample on the next cycle. When `ready` is `1` (`true`), you can specify the `data` and `valid` inputs for the next time step. When `ready` is `0` (`false`), the object ignores any input data in the next time step.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `dsphdl.CICInterpolator`

getLatency	Latency of CIC interpolation filter

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create CIC Interpolation Filter for HDL Code Generation

This example shows how to use a `dsphdl.CICInterpolator` System object™ to filter and upsample data. This object supports scalar and vector inputs. In this example, two functions are provided to work with scalar and vector input data separately. You can generate HDL code from these functions.

Generate Frames of Random Input Samples

Set up workspace variables for the object to use. The object supports fixed and variable interpolation rates for scalar inputs and only a fixed interpolation rate for vector inputs. The example runs the `HDLCICInterp_maxR8` function when you set the scalar variable to `true` and runs the `HDLCICInterp_vec` function when you set the scalar variable to `false`. For scalar inputs, choose a range of 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 64 and `R` must be an integer multiple of the input frame size.

```
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, 5, 6, 7, 8];
    vecSize = 1;
else
    varRValue = R; %#ok
    fac = (factor(R));
    vecSize = fac(randi(length(fac),1,1));
end

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

Generate Reference Output from `dsp.CICInterpolator` System Object

Generate frames of random input samples and apply the samples to the `dsp.CICInterpolator` System object. Later in this example, you use the output generated by the System object as reference data for comparison. The System object does not support a variable interpolation rate, so you must create and release the object for each change in interpolation factor value.

```
totalsamples = 0;
for i = 1:numFrames
    framesize(i) = varRValue(i)*randi([5 20],1,1);
    dataSamples{i} = fi(randn(vecSize,framesize(i)),1,16,8);
    ref_cic = dsp.CICInterpolator('DifferentialDelay',M, ...
                                  'NumSections',N, ...
                                  'InterpolationFactor',varRValue(i));
    refOutput = [refOutput,ref_cic(dataSamples{i}(:)).']; %#ok
    release(ref_cic);
end
```

Run Function Containing `dsphdl.CICInterpolator` System Object

Set the properties of the System object to match the input data parameters and run the function for your input type. These functions operate on a stream of data samples rather than a frame. You can generate HDL code from these functions.

The example uses the `HDLCICInterp_maxR8` function for a scalar input.

```
function [dataOut,validOut] = HDLCICInterp_maxR8(dataIn,validIn,R)
%HDLCICInterp_maxR8
% Performs CIC interpolation with an input interpolation factor up to 8.
% sampleIn is a scalar fixed-point value.
% validIn is a logical scalar value.

  persistent cic8;
  if isempty(cic8)
    cic8 = dsphdl.CICInterpolator('InterpolationSource','Input port', ...
                                  'MaxInterpolationFactor',8, ...
                                  'DifferentialDelay',1, ...
                                  'NumSections',3);
  end
```

```
  [dataOut,validOut] = cic8(dataIn,validIn,R);
end
```

The example uses the `HDLCICInterp_vec` function for a vector input.

```
function [dataOut,validOut] = HDLCICInterp_vec(dataIn,validIn)
%HDLCICInterp_vec
% Performs CIC interpolation with an input vector.
% sampleIn is a fixed-point vector.
% validIn is a logical scalar value.

  persistent cicVec;
  if isempty(cicVec)
    cicVec = dsphdl.CICInterpolator('InterpolationSource','Property', ...
                                    'InterpolationFactor',8, ...
                                    'DifferentialDelay',1, ...
                                    'NumSections',3);
  end
  [dataOut,validOut] = cicVec(dataIn,validIn);
end
```

To flush the remaining data, run the object by inserting the required number of idle cycles after each frame using the `latency` variable. For more information, see the “GainCorrection” on page 2-0 property.

Initialize the output to a size large enough to accommodate the output data. The final size is smaller than `totalsamples` due to interpolation.

```
if scalar
    latency = 3 + N + 9;
    dataOut = zeros(1,totalsamples*R+numFrames*latency);

else
    latency = 3 + (N*(vecSize*R))+ 3*N + 9; %#ok
    dataOut = zeros(vecSize*R,totalsamples+numFrames*latency);
end
validOut = zeros(1,size(dataOut,2));
idx=0;
for ij = 1:numFrames
    if scalar
        dataIn = upsample(dataSamples{ij},R);
        validIn = upsample(true(1,length(dataSamples{ij})),R);
        % scalar input with variable interpolation
        for ii = 1:length(validIn)
            idx = idx+1;
            [dataOut(:,idx),validOut(idx)] = HDLCICInterp_maxR8( ...
                dataIn(ii), ...
                validIn(ii), ...
                fi(varRValue(ij),0,12,0));
        end
        for ii = 1:latency
            idx = idx+1;
            [dataOut(:,idx),validOut(idx)] = HDLCICInterp_maxR8( ...
                fi(0,1,16,8), ...
                false, ...
```

```
                fi(varRValue(ij),0,12,0));
            end

        else
            % vector input with fixed interpolation
            for ii = 1:size(dataSamples{ij},2)  %#ok
                idx = idx+1;
                [dataOut(:,idx),validOut(idx)] = HDLCICInterp_vec( ...
                    dataSamples{ij}(:,ii), ...
                    true);
            end
            for ii = 1:latency
                idx = idx+1;
                [dataOut(:,idx),validOut(idx)] = HDLCICInterp_vec( ...
                    fi(zeros(vecSize,1),1,16,8), ...
                    false);
            end
        end
    end
```

Compare Function Output with Reference Data

Compare the function results against the output from the `dsp.CICInterpolator` object.

```
cicOutput = dataOut(:,validOut==1);
refOutput = refOutput(:);

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

```
CIC Interpolator

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

Explore Latency of CIC Interpolator Object

The latency of the `dsphdl.CICInterpolator` System object™ varies depending on how many integrator and comb sections your filter has, the input vector size, and whether you enable gain correction. Use the `getLatency` function to find the latency of a particular filter configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is continuously valid.

Create a `dsphdl.CICInterpolator` System object and request the latency. The default System object filter has two integrator and comb sections, and the gain correction is disabled.

```
hdlcic = dsphdl.CICInterpolator
```

```
hdlcic =
  dsphdl.CICInterpolator with properties:

    InterpolationSource: 'Property'
    InterpolationFactor: 2
      DifferentialDelay: 1
```

```
            NumSections: 2
              NumCycles: 1
          GainCorrection: false

  Show all properties
```

```
L_def = getLatency(hdlcic)
```

```
L_def = 13
```

Modify the filter object so it has three integrator and comb sections. Check the resulting change in latency.

```
hdlcic.NumSections = 3;
L_3sec = getLatency(hdlcic)
```

```
L_3sec = 18
```

Enable the gain correction on the filter object with vector input size 2. Check the resulting change in latency.

```
hdlcic.GainCorrection = true;
vecSize = 2;
L_wgain = getLatency(hdlcic,vecSize)
```

```
L_wgain = 33
```

Algorithms

CIC Interpolation Filter

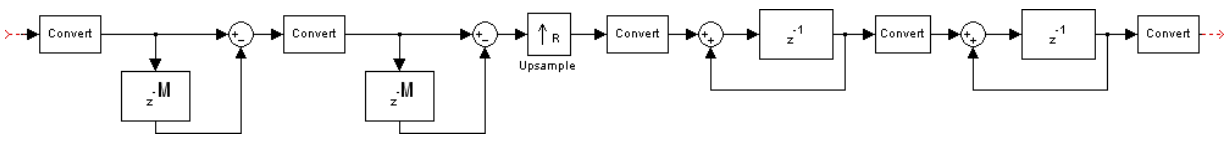
The transfer function of a CIC interpolation filter is

$$H(z) = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N = \frac{(1-z^{-RM})^N}{(1-z^{-1})^N} = \frac{(1-z^{-RM})^N}{1} \cdot \frac{1}{(1-z^{-1})^N} = H_C{}^N(z) \cdot H_I{}^N(z)\,.$$

- H_C is the transfer function of the comb part of the CIC filter.
- H_I is the transfer function of the integrator part of the CIC filter.
- N is the number of sections in either the comb part or integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- R is the interpolation factor.
- M is the differential delay.

CIC Filter Structure

The `dsphdl.CICInterpolator` System object has the CIC filter structure shown in this figure. The structure consists of N sections of cascaded comb filters, a rate change factor of R, and N sections of cascaded integrators [1].



You can locate the unit delay in the integrator part of the CIC filter in either the feedforward or feedback path. These two configurations yield an identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency of the paths. Because this configuration is preferred for HDL implementation, this System object puts the unit delay in the feedforward path of the integrator.

Fixed and Variable Interpolation

The System object upsamples the comb stage output using R, either using the fixed interpolation rate provided using the InterpolationFactor property or the variable interpolation rate provided using the R input argument. At the upsampling stage, the System object uses a counter to count the valid input samples, which depend on the interpolation rate. Whenever the interpolation rate changes, the System object resets and starts a new calculation from the next sample. This mechanism prevents the object from accumulating false values. Then, the System object provides the interpolated output to the integrator part of the CIC filter.

Gain Correction

The gain of the CIC interpolation filter at each stage is given by

$$G_i = \left\{ \begin{array}{ll} 2^i & i = 1, 2, ..., N \\ \frac{2^{2N-i}(RM)^{i-N}}{R} & i = N+1, ..., 2N \end{array} \right\}.$$

- G_i is the gain at ith stage.
- R is the `InterpolationFactor` property value.
- M is the `DifferentialDelay` property value.
- N is the `NumSections` property value.

The output of the System object is amplified by a specific gain value. This gain equals the gain of the $2N$th stage of the CIC interpolation filter and is given by $Gain = \frac{(R \times M)^N}{R}$.

The System object implements gain correction in two parts: coarse gain and fine gain. In coarse gain correction, the System object calculates the shift value, adds the shift value to the fractional bits to create a numeric type, and performs a bit-shift left and reinterpretcast. In fine gain correction, the System object divides the remaining gain with the coarse gain if the gain is not a power of 2. Then, the System object multiplies the corrected coarse gain value by the inverse value of the fine gain. Before the System object starts processing, all possible shift and fine gain values are precalculated and stored in an array.

You can modify this equation to $Gain = 2^{cGain} \times fGain$. In this equation, $cGain$ is the coarse gain and $fGain$ is the fine gain. These gains are given by these equations.

- $cGain = floor(\log_2 Gain)$
- $fGain = Gain/2^{cGain} = Gain \times 2^{-cGain}$

To perform gain correction when the `InterpolationSource` property is set to `'Input port'`, the System object sets the output data type configured with the maximum interpolation rate and bit-shifts left for all of the values under the maximum interpolation rate. The bit-shift value is equal to $Maximum\ gain - \log_2(current\ gain)$.

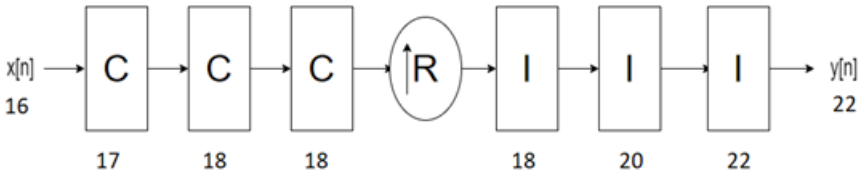
Output Data Type

This section explains how the System object outputs data is based on the output data type selection. Consider a System object with R, M, and N values of 8, 1, and 3, respectively, and an input width of 16. The word length at the ith stage is calculated as $B_i = B_{\text{In}} + [\log_2(G_i)]$, where:

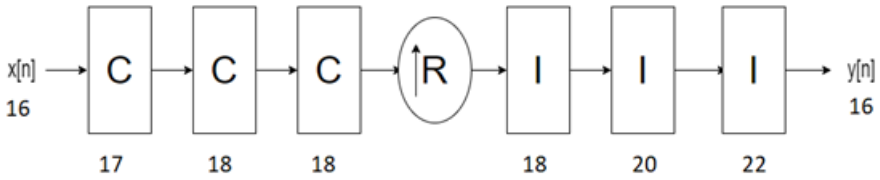
- G_i is the gain at ith stage.
- B_{In} is the input word length.
- B_i is the word length at ith stage.

The output word length is calculated as $B_{Out} = B_{In} + N - 1$, where B_{Out} is the output word length.

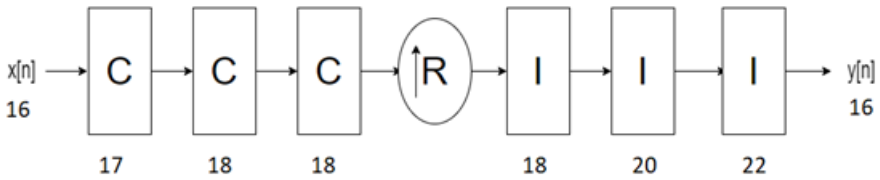
When you set the `OutputDataType` property to `'Full precision'`, the System object returns data with a word length of 22 by adding 6 gain bits to the input word length of 16. The word lengths of the internal comb and integrator stages are set to accommodate the bit growth.



When you set the `OutputDataType` property to `'Same word length as input'`, the object outputs data with a word length of 16, which is the same length as the input word length. The word lengths of the internal comb and integrator stages are set in the same way as in `'Full precision'` mode.



When you set the `OutputDataType` property to `'Minimum section word lengths'` and the `OutputWordLength` to `16`, the System object returns data with a word length of 16. The word lengths of the internal comb and integrator stages are set in the same way as in `'Full precision'` mode.



Latency

The latency of the System object changes depending on the type of input, the interpolation you specify, the value of the `NumSections` property, `GainCorrection` property, and the `NumCycles` property. This table shows the latency of the System object. *N* is the number of sections, *vecLen* is the length of the vector, and *R* is the interpolation factor.

Common latency is equal to 2 + (*N* x (*vecLen* x *R*)) + 3 x *N*, when *R* is equal to 1 and it is equal to 3 + (*N* x (*vecLen* x *R*)) + 3 x *N*, when *R* is greater than 1.

Input Data	Output Data	Interpolation Type	Gain Correction	Minimum number of cycles between valid input samples (NumCycles)	Latency in Clock Cycles
Scalar	Scalar	Fixed	`off`	*NumCycles* = R and > R	3 + *N* 2 + *N*, when *R* = 1.
			`on`	*NumCycles* = R and > R	3 + *N* + 9 2 + *N* + 9, when *R* = 1.
Scalar	Scalar	Variable	`off`	NA	4 + *N* 3 + *N*, when R_{max} = 1.
			`on`	NA	4 + *N* + 9 3 + *N* + 9, when R_{max} = 1.
Scalar	Vector	Fixed	`off`	*NumCycles* = 1	*Common latency* + 1, when *R* is greater than *N*. *Common latency*, when *R* is less than or equal to *N*. *Common latency* – (1 + floor(*N*/(3 x *R*))), when *R* is less than *N* and (*vecLen* == 2 && (*R* == 2 && (*N* == 4 || *N* == 5 || *N* == 6)) || (*R*== 3 && *N* == 6))
				NumCycles < R	3 + *N* + ((*R* + 1) x *N* + 2) + 1 + (*N* – 1) x *NumCycles*.
			`on`	*NumCycles* = 1	*Common latency* + 1 + 9, when *R* is greater than *N*. *Common latency* + 9, when *R* is less than or equal to *N*. *Common latency* – (1 + floor(*N*/(3 x *R*))) + 9, when *R* is less than *N* and (*vecLen* == 2 && (*R* == 2 && (*N* == 4 || *N* == 5 || *N* == 6)) || (*R* == 3 && *N* == 6)).
				NumCycles < R	3 + *N* + ((*R* + 1) x *N* + 2) + 1 + (*N* – 1) x *NumCycles* + 9

Input Data	Output Data	Interpolation Type	Gain Correction	Minimum number of cycles between valid input samples (NumCycles)	Latency in Clock Cycles
Vector	Vector	Fixed	`off`	*NumCycles* = 1	*Common latency* *Common latency* – 1, when (*vecLen* == 2 && (*R* == 2 && (*N* == 4 || *N* == 5 || *N* == 6)) || (*R*== 3 && *N* == 6)) || (*vecLen* == 3 && (*R* == 2 && *N* == 6)) *Common latency* – ((*N* >1) + (*N* > 4)), when R = 1 and *vecLen* == 2. *Common latency* – ((*N* > (*vecLen* – 1)), when *R* = 1 and *vecLen* > 2.
			`on`	*NumCycles* = 1	*Common latency* + 9 *Common latency* – 1 + 9, when (*vecLen* == 2 && (*R* == 2 && (*N* == 4 || *N* == 5 || *N* == 6)) || (*R*== 3 && *N* == 6)) || (*vecLen* == 3 && (*R* == 2 && *N* == 6)) *Common latency* – ((*N* >1) + (*N* > 4)) + 9, when *R* = 1 and *vecLen* == 2. *Common latency* – ((*N* > (*vecLen* – 1)) + 9, when *R* = 1 and *vecLen* > 2.

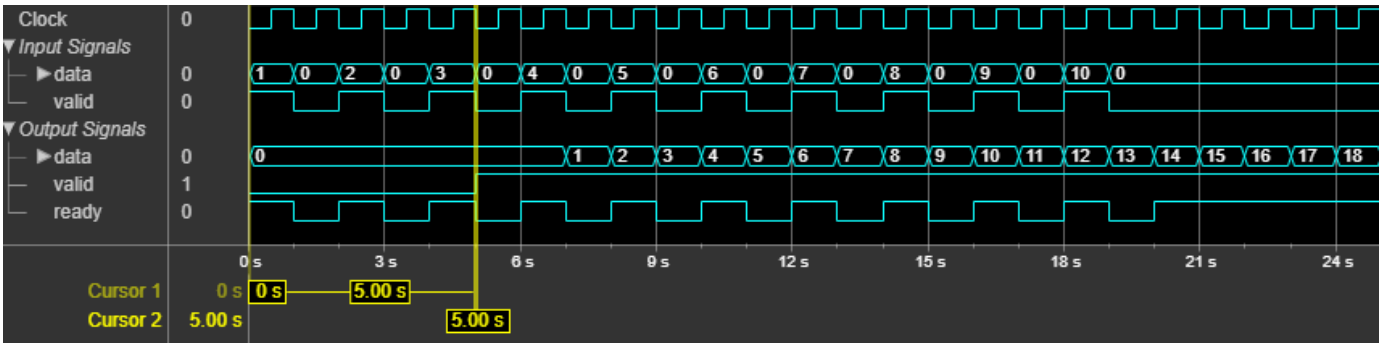
Note The System object does not support variable interpolation for these two combinations of input and output:

- Scalar input and vector output
- Vector input and vector output

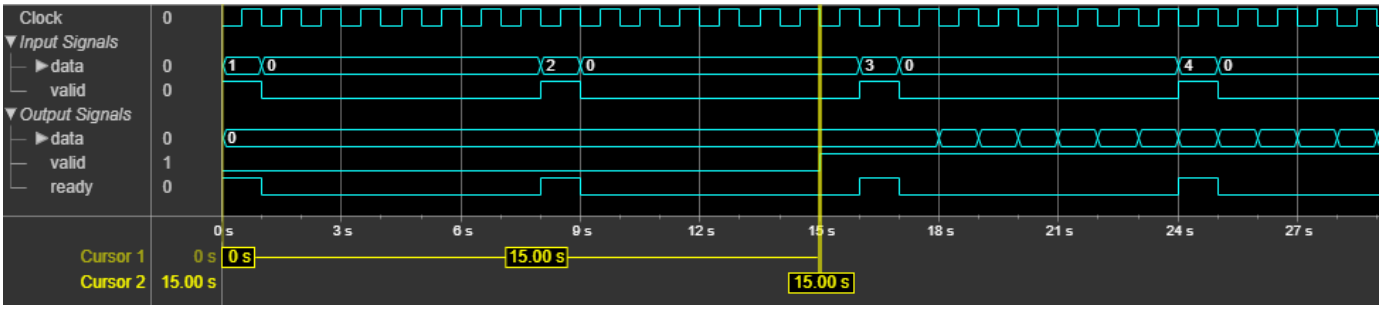
Scalar Input

This section shows the output of the System object for a scalar input with different *R*, *M*, and *N* values.

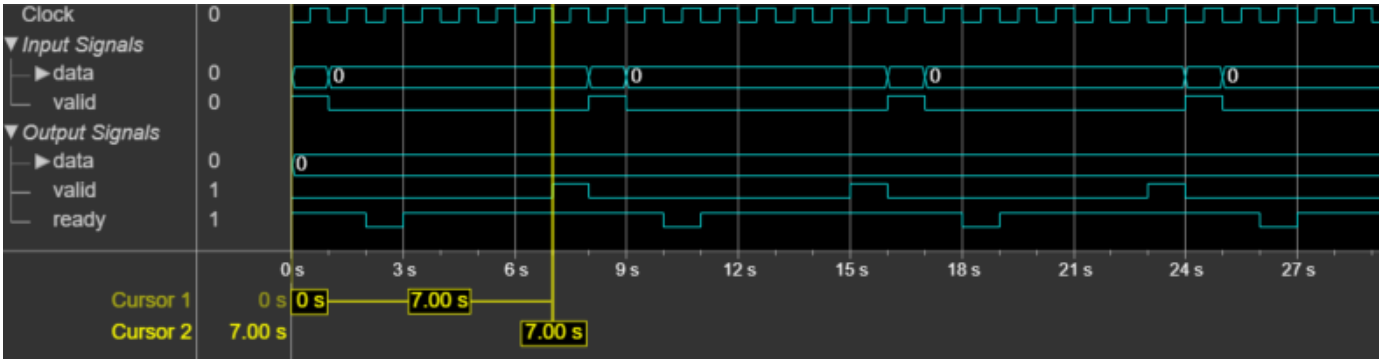
This figure shows the output of the System object with the default configuration (that is, with a fixed interpolation rate and *R*, *M*, and *N* values of 2, 1, and 2, respectively). The latency of the System object is 5 clock cycles and is calculated as 3 + *N*, where *N* is the number of sections.



This figure shows the output of the System object with a fixed interpolation rate, *R*, *M*, and *N* values of 8, 1, and 3, respectively, and `GainCorrection` set to `true`. The latency of the object is 15 clock cycles and is calculated as 3 + *N* + 9, where *N* is the number of sections.



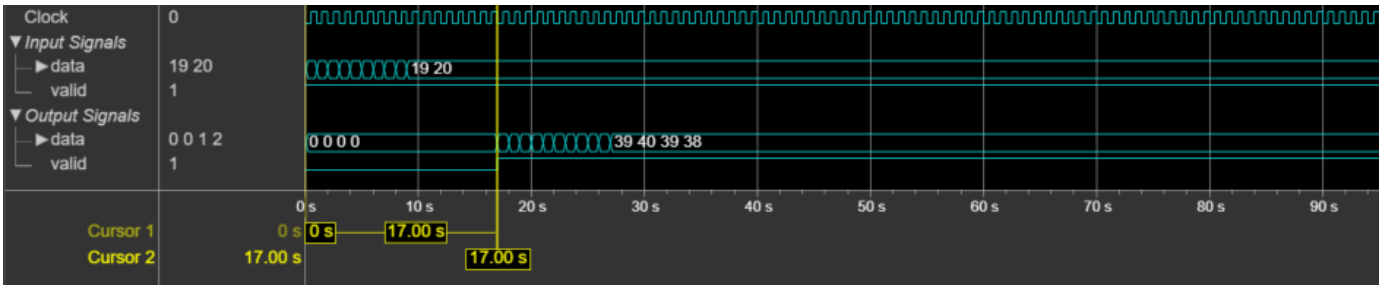
This figure shows the output of the System object with variable interpolation rate (`R` input argument) values of 2, 4, and 8 and with *M* and *N* values of 1 and 3, respectively. In this case, the `GainCorrection` property is set to `false`. The System object accepts `R` argument value changes only when `validIn` is `1` (`true`). The latency of the System object is 7 clock cycles and is calculated as 4 + *N*, where *N* is the number of sections.



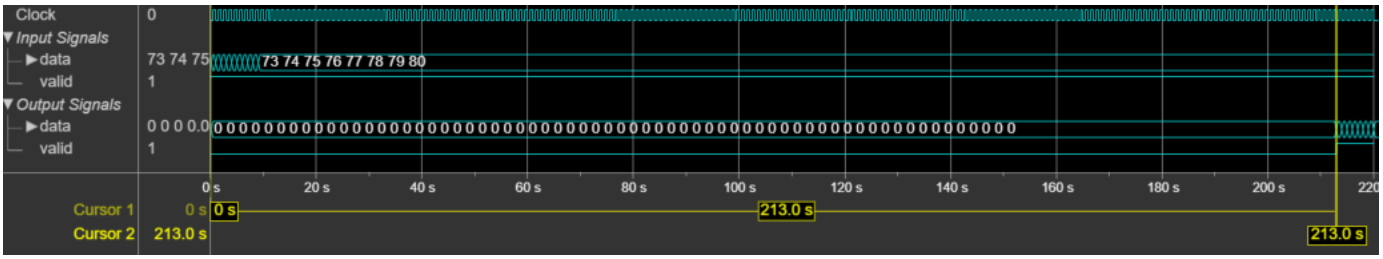
Vector Input

This section shows the output of the System object for a vector input with different *R*, *M*, and *N* values.

This figure shows the output of the System object for a two-element column vector input with the default configuration, that is, with a fixed interpolation rate and *R*, *M*, and *N* values of 2, 1, and 2, respectively. The latency of the System object is 38 clock cycles.



This figure shows the output of the System object for an eight-element column vector input with a fixed interpolation rate, *R*, *M*, and *N* values of 8, 1, and 3, respectively, and the `GainCorrection` property to `true`. The latency of the System object is 209 clock cycles.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. It also depends on the input data type.

This table shows the resource and performance data synthesis results of the System object for a scalar input with fixed and variable interpolation rates and for a two-element column vector of type `fixdt(1,16,0)` with a fixed interpolation rate when *R*, *M*, and *N* are 2, 1, and 2, respectively. The generated HDL code is targeted to the Xilinx Zynq- 7000 ZC706 Evaluation Board.

Input Data	Interpolation Type	Slice LUTs	Slice Registers	Maximum Frequency in MHz
Scalar	Fixed rate	68	90	844.12
	Variable rate	143	115	451.83
Vector	Fixed rate	480	921	376.51

The resources and frequencies vary based on the type of input data, and the values of *R*, *M*, and *N*, as well as other properties.

References

[1] Hogenauer, E. “An Economical Class of Digital Filters for Decimation and Interpolation.” *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29, no. 2 (April 1981): 155–62. https://doi.org/10.1109/TASSP.1981.1163535.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB® simulations, and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Objects

`dsphdl.CICDecimator`

Blocks

CIC Interpolator | CIC Decimator

Introduced in R2022a

dsphdl.CICDecimator

Package: `dsphdl`

Decimate signal using CIC filter

Description

The `dsphdl.CICDecimator` System object decimates an input signal by using a cascaded integrator-comb (CIC) decimation filter. CIC filters are a class of linear phase finite impulse response (FIR) filters consisting of a comb part and an integrator part. The CIC decimation filter structure consists of *N* sections of cascaded integrators, a rate change factor of *R*, and *N* sections of cascaded comb filters. For more information about CIC decimation filters, see “Algorithms” on page 2-27.

The System object supports these combinations of input and output data.

- Scalar input and scalar output — Support for fixed and variable decimation rates
- Vector input and scalar output — Support for fixed decimation rates only
- Vector input and vector output — Support for fixed decimation rates only

The System object provides an architecture suitable for HDL code generation and hardware deployment.

The System object supports real and complex fixed-point inputs.

To filter input data with an HDL-optimized CIC decimation filter, follow these steps:

1. Create the `dsphdl.CICDecimator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
cicDecFilt = dsphdl.CICDecimator
cicDecFilt = dsphdl.CICDecimator(Name,Value)
```

Description

`cicDecFilt = dsphdl.CICDecimator` creates an HDL-optimized CIC decimation filter System object, `cicDecFilt`, with default properties.

`cicDecFilt = dsphdl.CICDecimator(Name,Value)` creates the filter with properties set using one or more name-value arguments. Enclose each property name in single quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`DecimationSource` — Source of decimation factor
`'Property'` (default) | `'Input port'`

Specify whether the System object operates with a fixed or variable decimation rate.

- `'Property'` — Use a fixed decimation rate specified by the `DecimationFactor` property.
- `'Input port'` — Use a variable decimation rate specified by the `R` input argument.

For vector inputs, the System object does not support a variable decimation rate.

`DecimationFactor` — Decimation factor
2 (default) | integer from 1 to 2048

Specify the decimation factor as an integer from 1 to 2048. This value gives the rate at which the System object decimates the input.

Dependencies

To enable this property, set the `DecimationSource` property to `'Property'`.

`MaxDecimationFactor` — Upper bound of variable decimation factor
2 (default) | integer from 1 to 2048

Specify the upper bound of the range of valid values for the `R` input argument as an integer from 1 to 2048.

Note For vector inputs, the System object does not support variable decimation.

Dependencies

To enable this property, set the `DecimationSource` property to `'Input port'`.

`DifferentialDelay` — Differential delay
1 (default) | 2

Specify the differential delay of the comb part of the filter as either `1` or `2` cycles.

`NumSections` — Number of integrator or comb sections
2 (default) | 1 | 3 | 4 | 5 | 6

Specify the number of sections in either the integrator or the comb part of the System object.

`GainCorrection` — Output gain compensation
`false` (default) | `true`

Set this property to `true` to compensate for the output gain of the filter.

The latency of the System object varies depending on the type of input, the decimation you specify, the number of sections, and the value of this property. For more information on the latency of the System object, see “Latency” on page 2-30.

`OutputDataType` — Data type of output
`'Full precision'` (default) | `'Same word length as input'` | `'Minimum section word lengths'`

Choose the data type of the filtered output data.

- `'Full precision'` — The output data type has a word length equal to the input word length plus gain bits.
- `'Same word length as input'` — The output data type has a word length equal to the input word length.
- `'Minimum section word lengths'` — The output data type uses the word length you specify in the `OutputWordLength` property. When you choose this option, the System object applies a pruning algorithm internally. For more information about pruning, see “Output Data Type” on page 2-28.

`OutputWordLength` — Word length of output
16 (default) | integer from 2 to 104

Word length of the output, specified as an integer from 2 to 104.

Note When this value is 2, 3, 4, 5, or 6, the System object can overflow the output data.

Dependencies

To enable this property, set the `OutputDataType` property to `'Minimum section word lengths'`.

`ResetInputPort` — Reset argument
`false` (default) | `true`

When you set this property to `true`, the System object expects a `reset` input argument.

Usage

Syntax

```
[dataOut,validOut] = cicDecFilt(dataIn,validIn)
[dataOut,validOut] = cicDecFilt(dataIn,validIn,R)
[dataOut,validOut] = cicDecFilt(dataIn,validIn,reset)
[dataOut,validOut] = cicDecFilt(dataIn,validIn,R,reset)
```

Description

`[dataOut,validOut] = cicDecFilt(dataIn,validIn)` filters and decimates the input data using a fixed decimation factor only when `validIn` is `true`.

`[dataOut,validOut] = cicDecFilt(dataIn,validIn,R)` filters the input data using the specified variable decimation factor `R`. The `DecimationSource` property must be set to `'Input port'`.

`[dataOut,validOut] = cicDecFilt(dataIn,validIn,reset)` filters the input data when `reset` is `false` and clears filter internal states when `reset` is `true`. The System object expects the `reset` argument only when you set the `ResetInputPort` property to `true`.

`[dataOut,validOut] = cicDecFilt(dataIn,validIn,R,reset)` filters the input data when `reset` is `false` and clears filter internal states when `reset` is `true`. The System object expects the `reset` argument only when you set the `ResetInputPort` property to `true`. The `DecimationSource` property must be set to `'Input port'`.

Input Arguments

`dataIn` — Input data
scalar | column vector

Specify input data as a scalar or a column vector with a length from 1 to 64. The input data must be a signed integer or signed fixed point with a word length less than or equal to 32. The `DecimationFactor` property must be an integer multiple of the input frame size.

Data Types: `int8` | `int16` | `int32` | `fi`
Complex Number Support: Yes

`validIn` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

`R` — Variable decimation rate
scalar

Specify the decimation rate.

The `R` value must have the data type `fi(0,12,0)` and it must be an integer in the range from 1 to the `MaxDecimationFactor` property value.

Dependencies

To enable this argument, set the `DecimationSource` property to `'Input port'`.

Data Types: `fi(0,12,0)`

`reset` — Clears internal states
logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the object captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set the `ResetInputPort` property to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — CIC-decimated output data
scalar | column vector

CIC-decimated output data, returned as a scalar or a column vector with a length from 1 to 64.

The `OutputDataType` property sets the data type of this argument. See “Output Data Type” on page 2-28.

Data Types: `int8` | `int16` | `int32` | `fi`
Complex Number Support: Yes

`validOut` — Indicates valid output data
scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `dsphdl.CICDecimator`

getLatency	Latency of CIC decimation filter

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create CIC Decimation Filter for HDL Code Generation

This example shows how to use a `dsphdl.CICDecimator` System object™ to filter and downsample data. This object supports scalar and vector inputs. In this example, two functions are provided to work with scalar and vector inputs separately. You can generate HDL code from these functions.

Generate Frames of Random Input Samples

Set up workspace variables for the object to use. The object supports fixed and variable decimation rates for scalar inputs and only a fixed decimation rate for vector inputs. The example runs the `HDLCIC_maxR8` function when you set the scalar variable to `true` and runs the `HDLCIC_vec` function when you set the scalar variable to `false`. For scalar inputs, choose a range of 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 and `R` must be an integer multiple of the 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 = [2, 4, 5, 6, 7, 8];
    vecSize = 1;
else
    varRValue = R; %#ok
    fac = (factor(R));
    vecSize = fac(randi(length(fac),1,1));
end

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

Generate Reference Output from `dsp.CICDecimator` System Object

Generate frames of random input samples and apply the samples to the `dsp.CICDecimator` System object. Later in this example, you use the output generated by the System object as reference data for comparison. The System object does not support a variable decimation rate, so you must create and release the object for each change in decimation factor value.

```
totalsamples = 0;
for i = 1:numFrames
    framesize(i) = varRValue(i)*randi([5 20],1,1);
    dataSamples{i} = fi(randn(vecSize,framesize(i)),1,16,8);
    ref_cic = dsp.CICDecimator('DifferentialDelay',M, ...
                               'NumSections',N, ...
                               'DecimationFactor',varRValue(i));
    refOutput = [refOutput,ref_cic(dataSamples{i}(:)).']; %#ok
    release(ref_cic);
end
```

Run Function Containing `dsphdl.CICDecimator` System Object

Set the properties of the System object to match the input data parameters and run the function for your input type. These functions operate on a stream of data samples rather than a frame. You can generate HDL code from these functions.

The example uses the `HDLCIC_maxR8` function for a scalar input.

```
function [dataOut,validOut] = HDLCIC_maxR8(dataIn,validIn,R)
%HDLCIC_maxR8
% Performs CIC decimation with an input decimation factor up to 8.
% dataIn is a scalar fixed-point value.
% validIn is a logical scalar value.

  persistent cic8;
  if isempty(cic8)
    cic8 = dsphdl.CICDecimator('DecimationSource','Input port', ...
                               'MaxDecimationFactor',8, ...
                               'DifferentialDelay',1, ...
                               'NumSections',3);
  end
  [dataOut,validOut] = cic8(dataIn,validIn,R);
end
```

The example uses the `HDLCIC_vec` function for a vector input.

```
function [dataOut,validOut] = HDLCIC_vec(dataIn,validIn)
%HDLCIC_vec
% Performs CIC decimation with an input vector.
% dataIn is a fixed-point vector.
% validIn is a logical scalar value.

  persistent cicVec;
  if isempty(cicVec)
    cicVec = dsphdl.CICDecimator('DecimationSource','Property', ...
                                 'DecimationFactor',8, ...
                                 'DifferentialDelay',1, ...
                                 'NumSections',3);
  end
  [dataOut,validOut] = cicVec(dataIn,validIn);
end
```

To flush the remaining data, run the object by inserting the required number of idle cycles after each frame using the `latency` variable. For more information, see **“Latency” on page 2-30**.

Initialize the output to a size large enough to accommodate the output data. The final size is smaller than `totalsamples` due to decimation.

```
latency = floor((vecSize - 1)*(N/vecSize)) + 1 + N + (2+(vecSize+1)*N) + 9;
dataOut = zeros(1,totalsamples+numFrames*latency);
validOut = zeros(1,totalsamples+numFrames*latency);
idx=0;
for ij = 1:numFrames
    if scalar
        % scalar input with variable decimation
        for ii = 1:length(dataSamples{ij})
            idx = idx+1;
            [dataOut(idx),validOut(idx)] = HDLCIC_maxR8( ...
                dataSamples{ij}(ii), ...
                true, ...
                fi(varRValue(ij),0,12,0));
```

```
        end
        for ii = 1:latency
            idx = idx+1;
            [dataOut(idx),validOut(idx)] = HDLCIC_maxR8( ...
                fi(0,1,16,8), ...
                false, ...
                fi(varRValue(ij),0,12,0));
        end

    else
        % vector input with fixed decimation
        for ii = 1:size(dataSamples{ij},2) %#ok
            idx = idx+1;
            [dataOut(idx),validOut(idx)] = HDLCIC_vec( ...
                dataSamples{ij}(:,ii), ...
                true);
        end
        for ii = 1:latency
            idx = idx+1;
            [dataOut(idx),validOut(idx)] = HDLCIC_vec( ...
                fi(zeros(vecSize,1),1,16,8), ...
                false);
        end
    end
end
```

Compare Function Output with Reference Data

Compare the function results against the output from the `dsp.CICDecimator` object.

```
cicOutput = dataOut(validOut==1);

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

```
CIC Decimator

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

Explore Latency of CIC Decimator Object

The latency of the `dsphdl.CICDecimator` System object™ varies depending on how many integrator and comb sections your filter has, the input vector size, and whether you enable gain correction. Use the `getLatency` function to find the latency of a particular filter configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is continuously valid.

Create a `dsphdl.CICDecimator` System object and request the latency. The default System object filter has two integrator and comb sections, and the gain correction is disabled.

```
hdlcic = dsphdl.CICDecimator
```

```
hdlcic =
  dsphdl.CICDecimator with properties:
```

```
     DecimationSource: 'Property'
     DecimationFactor: 2
    DifferentialDelay: 1
          NumSections: 2
       GainCorrection: false

  Show all properties
```

```
L_def = getLatency(hdlcic)
```

```
L_def = 5
```

Modify the filter object so it has three integrator and comb sections. Check the resulting change in latency.

```
hdlcic.NumSections = 3;
L_3sec = getLatency(hdlcic)
```

```
L_3sec = 6
```

Enable the gain correction on the filter object with vector input size 2. Check the resulting change in latency.

```
hdlcic.GainCorrection = true;
vecSize = 2;
L_wgain = getLatency(hdlcic,vecSize)
```

```
L_wgain = 25
```

Algorithms

CIC Decimation Filter

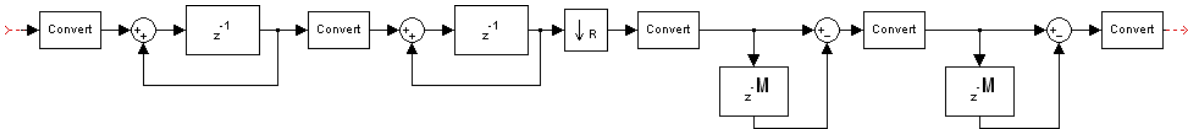
The transfer function of a CIC decimation filter is

$$H(z) = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N = \frac{\left(1 - z^{-RM}\right)^N}{\left(1 - z^{-1}\right)^N} = \frac{1}{\left(1 - z^{-1}\right)^N} \cdot \frac{\left(1 - z^{-RM}\right)^N}{1} = H_I{}^N(z) \cdot H_c{}^N(z).$$

- H_I is the transfer function of the integrator part of the CIC filter.
- H_C is the transfer function of the comb part of the CIC filter.
- N is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part or the integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.
- R is the decimation factor.
- M is the differential delay.

CIC Filter Structure

The `dsphdl.CICDecimator` System object has the CIC filter structure shown in this figure. The structure consists of N sections of cascaded integrators, a rate change factor of R, and N sections of cascaded comb filters [1].



Designs can put the unit delay in the integrator part of the CIC filter in either the feedforward or feedback path. These two configurations yield an identical filter frequency response. However, the numerical outputs from these two configurations are different due to the latency of the paths. This System object puts the unit delay in the feedforward path of the integrator.

Fixed and Variable Decimation

The System object downsamples the integrator stage output using R, either based on the fixed decimation rate provided using the `DecimationFactor` property or the variable decimation rate provided using the R input argument. At the downsampler stage, the System object uses a counter to count the valid input samples, which depend on the decimation rate. Whenever the decimation rate changes, the object resets and starts a new calculation from the next sample. This mechanism prevents the System object from accumulating invalid values. Then, the System object provides the decimated output to the comb part.

Gain Correction

The gain of the System object is given by $Gain = (R \times M)^N$.

- R is the `DecimationFactor` property value.
- M is the `DifferentialDelay` property value.
- N is the `NumSections` property value.

The System object implements gain correction in two parts: coarse gain and fine gain. In coarse gain correction, the System object calculates the shift value, adds the shift value to the fractional bits to create a numeric type, and then performs a bit-shift left. In fine gain correction, the System object divides the remaining gain with the coarse gain if the gain is not a power of 2 and then multiplies the coarse gain corrected value with the inverse value of fine gain. All possible shift and fine gain values are precalculated and stored in an array before the System object starts processing.

You can modify this equation as $Gain = 2^{cGain} \times fGain$, where *cGain* means coarse gain and *fGain* means fine gain.

- $cGain = floor(\log_2 Gain)$
- $fGain = Gain/2^{cGain} = Gain \times 2^{-cGain}$

To perform `GainCorrection` when the `InterpolationSource` property is set to `'Input port'`, the System object sets the output data type configured with the maximum decimation rate and bit-shifts left for all the values under the maximum decimation rate. The bit-shift value is equal to $Maximum\ gain - \log_2(current\ gain)$.

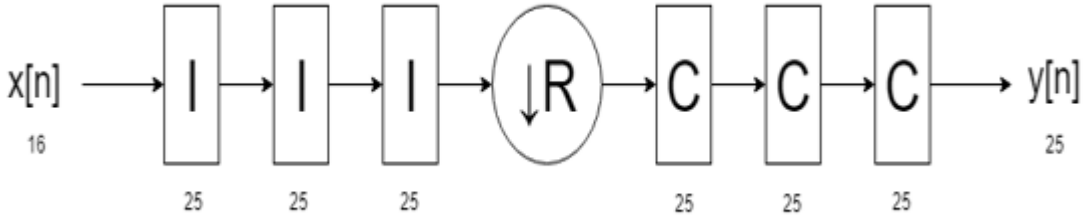
Output Data Type

This section explains how the System object determines the output data type. For example, consider a filter with `DecimationFactor`, `DifferentialDelay`, and `NumSections` values of 8, 1, and 3, respectively, with an input width of 16 bits.

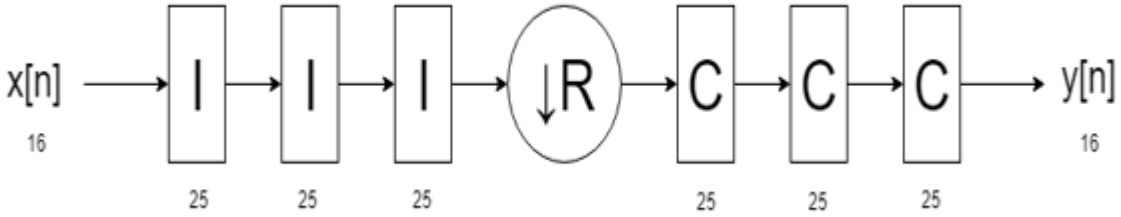
The output word length is calculated as $B_{Out} = B_{In} + [\log_2(Gain)]$.

- $Gain = (R \times M)^N$
- B_{In} is the input word length.
- B_{Out} is the output word length.

When you set the `OutputDataType` property to `'Full precision'`, the System object returns data with a word length of 25 bits, by adding nine gain bits to the input word length.

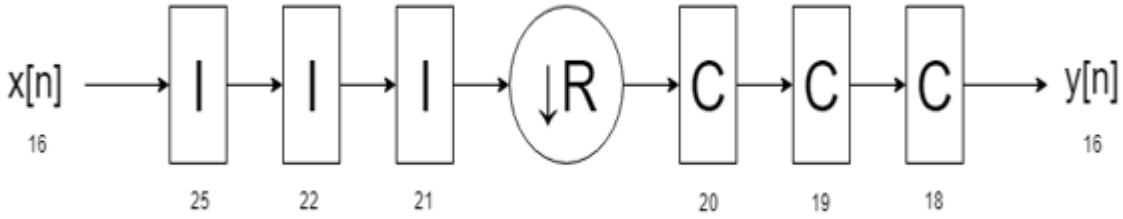


When you set the `OutputDataType` property to `'Same word length as input'`, the object outputs data with a word length of 16, which is the same length as the input word length. The internal integrator and comb stages use the full-precision data type with 25 bits.



When you set the `OutputDataType` property to `'Minimum section word lengths'` and the `OutputWordLength` property to `16`, the System object returns data with a word length of 16 bits. In this case, the object changes the bit width at each stage, based on the pruning algorithm.

If the `OutputWordLength` property value is less than the number of bits requested at the output, the least significant bits (LSBs) at the earlier stages are pruned. The Hogenauer algorithm provides the number of LSBs to discard at each stage. This algorithm minimizes the loss of information in the output data [1].



Latency

The latency of the System object varies depending on the type of input, the decimation you specify, the number of sections, and the value of the `GainCorrection` property. This table shows the latency of the System object. N is the number of sections and *vecLen* is the length of the vector.

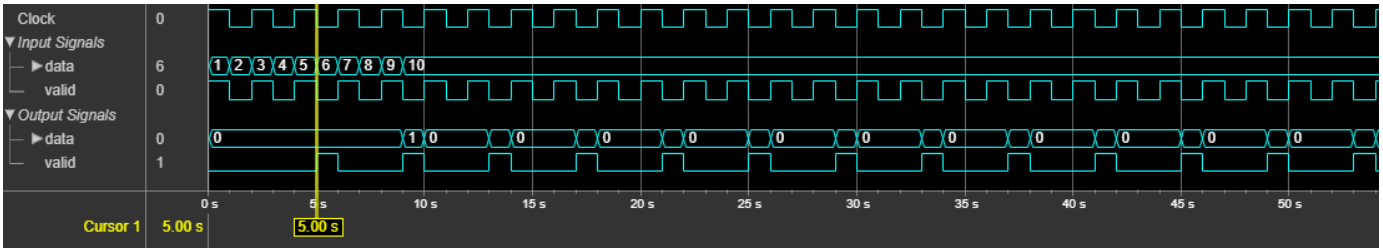
Input Data	Output Data	Decimation Type	Gain Correction	Latency in Clock Cycles
Scalar	Scalar	Fixed	`off`	$3 + N$. When $R = 1$, $2 + N$.
			`on`	$3 + N + 9$. When $R = 1$, $2 + N + 9$.
Scalar	Scalar	Variable	`off`	$4 + N$. When $R_{max} = 1$, $3 + N$.
			`on`	$4 + N + 9$. When $R_{max} = 1$, $3 + N + 9$.
Vector	Scalar	Fixed	`off`	`floor`((*vecLen* – 1) x (*N*/*vecLen*)) + 1 + *N* + (2 + (*vecLen* + 1) x *N*
			`on`	`floor`((*vecLen* – 1) x (*N*/*vecLen*)) + 1 + *N* + (2 + (*vecLen* + 1) x *N*) + 9
Vector	Vector	Fixed	`off`	`floor`((*vecLen* – 1) x (*N*/*vecLen*)) + 1 + *N* + (2 + (*vecLen* + 1) x *N*
			`on`	`floor`((*vecLen* – 1) x (*N*/*vecLen*)) + 1 + *N* + (2 + (*vecLen* + 1) x *N*) + 9

Note For vector inputs, the System object does not support variable decimation.

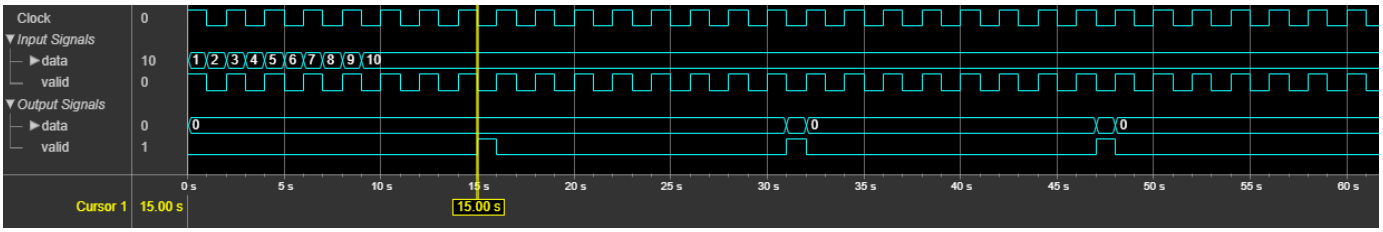
Scalar Input

This section shows the output of the System object for a scalar input with different R, M, and N values.

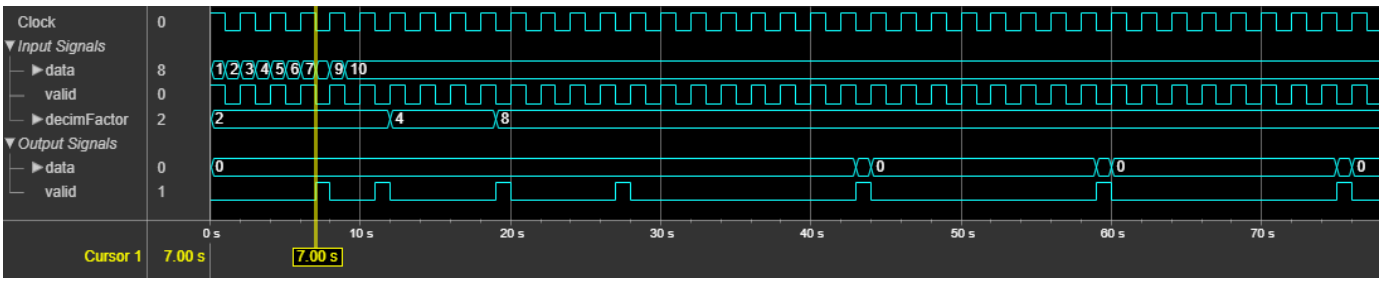
This figure shows the output of the System object with the default configuration, that is, with a fixed decimation rate and `DecimationFactor`, `DifferentialDelay`, and `NumSections` values of 2, 1, and 2, respectively. The System object returns valid output data at every second cycle based on the fixed `DecimationFactor` value of 2. The latency of the System object is 5 clock cycles, calculated as $3 + N$.



This figure shows the output of the System object with a fixed decimation rate, `DecimationFactor`, `DifferentialDelay`, and `NumSections` values of 8, 1, and 3, respectively, and `GainCorrection` set to `true`. The System object returns valid output data at every eighth cycle based on the fixed `DecimationFactor` value of 8. The latency of the object is 15 clock cycles and is calculated as $3 + N + 9$.



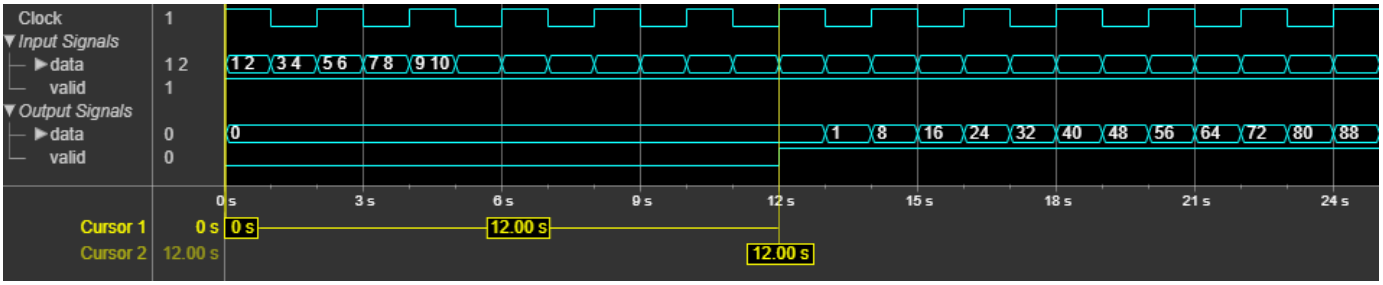
This figure shows the output of the System object for variable `R` values of 2, 4, and 8 along with M and N values of 1 and 3. The `GainCorrection` property is set to `false`. The System object returns valid output data at the second, fourth, and eighth cycles corresponding to the `R` values 2, 4, and 8, respectively. The System object accepts `R` argument value changes only when the input `validIn` is `1` (`true`). The latency of the System object is 7 clock cycles, calculated as 4 + N.



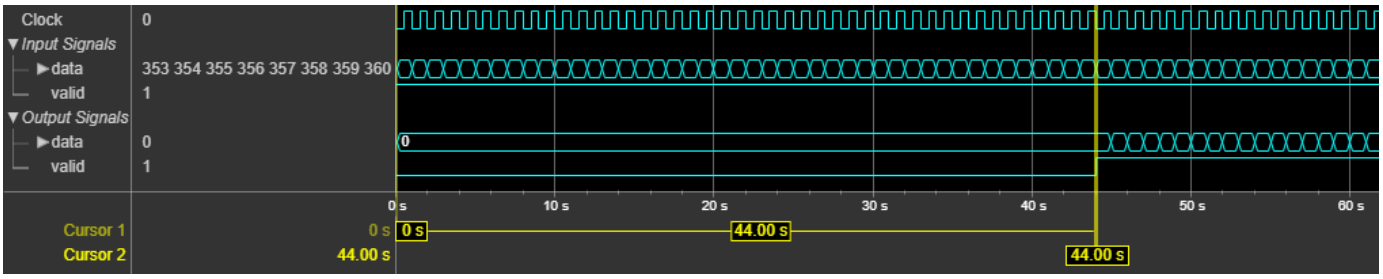
Vector Input

This section shows the output of the System object for a vector input with different R, M, and N values.

This figure shows the output of the System object for a two-element column vector input with the default configuration, that is, with a fixed decimation rate and `DecimationFactor`, `DifferentialDelay`, and `NumSections` values of 2, 1, and 2, respectively. The latency of the object is 12 clock cycles.



This figure shows the output of the System object for an eight-element column vector input with a fixed decimation rate, R, M, and N values of 8, 1, and 3, respectively, and `GainCorrection` set to `true`. The latency of the object is 44 clock cycles.



Performance

The performance of the synthesized HDL code varies with your target and synthesis options. It also varies based on the input data type.

This table shows the resource and performance data synthesis results of the object for a scalar input of type `fixdt(1,16,0)` with fixed and variable decimation rates and for a two-element column vector input with fixed decimation rate `DecimationFactor`, `DifferentialDelay`, and `NumSections` values are 2, 1, and 2, respectively. The generated HDL is targeted to the Xilinx Zynq-7000 ZC706 Evaluation Board.

Input Data	Decimation Type	Slice LUTs	Slice Registers	Maximum Frequency in MHz
Scalar	Fixed rate	101	166	711.74
	Variable rate	206	186	441.70
Vector	Fixed rate	218	627	624.61

The resources and frequencies vary based on the type of input data and the values of R, M, and N, as well as other properties. Using a vector input can increase the throughput, however, doing so also increases the number of hardware resources that the System object uses.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox

Behavior changed in R2022a

Before R2022a, this System object was named `dsp.HDLCICDecimation`, and was included in the DSP System Toolbox.

Changes to decimation factor arguments

Behavior changed in R2022a

- In previous releases, a decimation factor of `1` was invalid. You can now set the decimation factor to `1`.

Configuration	Before R2022a	After 2022a
Variable decimation factor	Select the `VariableDownsample` property and set the `DecimationFactor` parameter to the maximum expected decimation factor.	Set the `DecimationSource` property to `Input port` and set the `MaxDecimationFactor` property to the maximum expected decimation factor. The `decimFactor` port is renamed to `R`.
Fixed decimation factor	Clear the `VariableDownsample` property and set the `DecimationFactor` property to the desired decimation factor.	Set the `DecimationSource` property to `Property` and set the `DecimationFactor` property to the desired decimation factor.

- `ResetIn` property is renamed to `ResetInputPort`.

References

[1] Hogenauer, E. “An Economical Class of Digital Filters for Decimation and Interpolation.” *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29, no. 2 (April 1981): 155–62. https://doi.org/10.1109/TASSP.1981.1163535.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Objects

`dsphdl.CICInterpolator`

Blocks

CIC Decimator | CIC Interpolator

Introduced in R2019b

dsphdl.FIRFilter

Package: `dsphdl`

Finite impulse response filter

Description

The `dsphdl.FIRFilter` System object models finite-impulse response filter architectures optimized for HDL code generation. The object accepts scalar or vector input, and provides an option for programmable coefficients. It provides a hardware-friendly interface with input and output control signals. To provide a cycle-accurate simulation of the generated HDL code, the object models architectural latency including pipeline registers and resource sharing.

The object provides three filter structures. The direct form systolic architecture provides a fully parallel implementation that makes efficient use of Intel and Xilinx DSP blocks. The direct form transposed architecture is a fully parallel implementation and is suitable for FPGA and ASIC applications. The partly-serial systolic architecture provides a configurable serial implementation that makes efficient use of FPGA DSP blocks. 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.

All three structures optimize hardware resources by sharing multipliers for symmetric or antisymmetric filters and by removing the multipliers for zero-valued coefficients such as in half-band filters and Hilbert transforms.

The latency between valid input data and the corresponding valid output data depends on the filter structure, serialization options, the number of coefficients, and whether the coefficient values provide optimization opportunities.

To filter input data with an HDL-optimized FIR filter:

1. Create the `dsphdl.FIRFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
firFilt = dsphdl.FIRFilter
firFilt = dsphdl.FIRFilter(num)
firFilt = dsphdl.FIRFilter( ___ ,Name,Value)
```

Description

`firFilt = dsphdl.FIRFilter` creates an HDL-optimized discrete FIR filter System object, `firFilt`, with default properties.

`firFilt = dsphdl.FIRFilter(num)` creates a filter with the `Numerator` property set to `num`.

`firFilt = dsphdl.FIRFilter( ___ ,Name,Value)`sets properties using one or more name-value pairs. Enclose each property name in single quotes.

For example:

```
Numerator = firpm(10,[0,0.1,0.5,1],[1,1,0,0]);
fir = dsphdl.FIRFilter(Numerator,'FilterStructure','Direct form transposed');
...
[dataOut,validOut] = fir(dataIn,validIn);
```

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Main

`NumeratorSource` — Source of filter coefficients

`'Property'` (default) | `'Input port (Parallel interface)'`

You can enter constant filter coefficients as a property or provide time-varying filter coefficients using an input argument.

Setting this property to `'Input port (Parallel interface)'` enables the `coeff` argument, and the `NumeratorPrototype` property. Specify a prototype to enable the object to optimize the filter implementation according to the symmetry of your coefficients. To use `'Input port (Parallel interface)'`, set the `FilterStructure` property to `'Direct form systolic'` or `'Direct form transposed'`.

`Numerator` — Discrete FIR filter coefficients

`[0.5 0.5]` (default) | real or complex vector

Discrete FIR filter coefficients, specified as a vector of real or complex values. You can also specify the vector as a workspace variable, or as a call to a filter design function. When the input data type is a floating-point type, the object casts the coefficients to the same data type as the input. When the input data type is an integer type or a fixed-point type, you can modify the coefficient data type by using the `CoefficientsDataType` property.

Example: `dsphdl.FIRFIlter('Numerator',firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0]))` defines coefficients using a linear-phase filter design function.

Dependencies

To enable this property, set `NumeratorSource` to `'Property'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`NumeratorPrototype` — Prototype filter coefficients

`[]` (default) | real or complex vector

Prototype filter coefficients, specified as a vector of real or complex values. The prototype specifies a sample coefficient vector that is representative of the symmetry and zero-value locations of the expected input coefficients. If all of your input coefficient vectors have the same symmetry and zero-value coefficient locations, set `NumeratorPrototype` to one of those vectors. If your coefficients are unknown or not expected to share symmetry or zero-value locations, set `NumeratorPrototype` to `[]`. The object uses the prototype to optimize the filter by sharing multipliers for symmetric or antisymmetric coefficients and by removing multipliers for zero-value coefficients.

Coefficient optimizations affect the expected size of the `coeff` input argument. Provide only the nonduplicate coefficients as the argument. For example, if you set the `NumeratorPrototype` property to a symmetric 14-tap filter, the object shares one multiplier between each pair of duplicate coefficients, so the object expects a vector of 7 values for the `coeff` argument. You must still provide zeros in the input `coeff` vector for the nonduplicate zero-value coefficients.

Dependencies

To enable this property, set `NumeratorSource` to `'Input port (Parallel interface)'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`FilterStructure` — HDL filter architecture

`'Direct form systolic'` (default) | `'Direct form transposed'` | `'Partly serial systolic'`

HDL filter architecture, specified as one of these structures:

- `'Direct form systolic'` — This architecture provides a fully parallel filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For architecture details, see “Fully Parallel Systolic Architecture”.
- `'Direct form transposed'` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. For architecture details, see “Fully Parallel Transposed Architecture”.
- `'Partly serial systolic'` — This architecture provides a serial filter implementation and options for tradeoffs between throughput and resource utilization. It makes efficient use of Intel and Xilinx DSP blocks. The object implements a serial L-coefficient filter with M multipliers and requires input samples that are at least N cycles apart, such that $L = N \times M$. You can specify either M or N. For this implementation, the object provides an output signal, `ready`, that indicates when the object is ready for new input data. For architecture and performance details, see “Partly Serial Systolic Architecture ($1 < N < L$)” and “Fully Serial Systolic Architecture ($N \geq L$)”. You cannot use frame-based input with the partly-serial architecture.

All implementations share multipliers for symmetric and antisymmetric coefficients and remove multipliers for zero-valued coefficients.

`SerializationOption` — Rule to define serial implementation

`'Minimum number of cycles between valid input samples'` (default) | `'Maximum number of multipliers'`

Specify the rule that the object uses to serialize the filter as one of:

- `'Minimum number of cycles between valid input samples'` – Specify a requirement for input data timing by using the `NumCycles` property.
- `'Maximum number of multipliers'` – Specify a requirement for resource usage by using the `NumberOfMultipliers` property.

For a filter with L coefficients, the object implements a serial filter with not more than M multipliers and requires input samples that are at least N cycles apart, such that $L = N \times M$. The object might remove additional multipliers when it applies coefficient optimizations, so the actual M or N values of the filter implementation can be lower than the value that you specified.

Dependencies

To enable this property, set `FilterStructure` to `'Partly serial systolic'`.

`NumCycles` — Serialization requirement for input timing

2 (default) | positive integer

Serialization requirement for input timing, specified as a positive integer. This property represents N, the minimum number of cycles between valid input samples. In this case, the object calculates $M = L/N$. To implement a fully-serial architecture, set `NumCycles` to a value greater than the filter length, L, or to `Inf`.

The object might remove multipliers when it applies coefficient optimizations, so the actual M and N values of the filter can be lower than the value you specified.

Dependencies

To enable this property, set `FilterStructure` to `'Partly serial systolic'` and set `SerializationOption` to `'Minimum number of cycles between valid input samples'`.

`NumberOfMultipliers` — Serialization requirement for resource usage

2 (default) | positive integer

Serialization requirement for resource usage, specified as a positive integer. This property represents M, the maximum number of multipliers in the filter implementation. In this case, the object calculates $N = L/M$. If the input data is complex, the object allocates `floor(`M`/2)` multipliers for the real part of the filter and `floor(`M`/2)` multipliers for the imaginary part of the filter. To implement a fully-serial architecture, set `NumberOfMultipliers` to `1` for real input with real coefficients, `2` for complex input and real coefficients or real coefficients with complex input, or `3` for complex input and complex coefficients.

The object might remove multipliers when it applies coefficient optimizations, so the actual M and N values of the filter can be lower than the value you specified.

Dependencies

To enable this property, set the `FilterStructure` to `'Partly serial systolic'`, and set `SerializationOption` to `'Maximum number of multipliers'`.

Data Types

`Rounding` — Rounding method for type-casting the output

`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method for type-casting the output, specified as `'Floor'`, `'Ceiling'`, `'Convergent'`, `'Nearest'`, `'Round'`, or `'Zero'`. The rounding method is used when casting the output to the data type specified by the `OutputDataType` property. When the input data type is floating point, the object ignores the `RoundingMethod` property. For more details, see “Rounding Modes”.

`OverflowAction` — Overflow handling for type-casting the output

`'Wrap'` (default) | `'Saturate'`

Overflow handling for type-casting the output, specified as `'Wrap'` or `'Saturate'`. Overflow handling is used when casting the output to the data type specified by the `OutputDataType` property. When the input data type is floating point, the object ignores the `OverflowAction` property. For more details, see “Overflow Handling”.

`CoefficientsDataType` — Data type of discrete FIR filter coefficients
`'Same word length as input'` (default) | `numerictype` object

Data type of discrete FIR filter coefficients, specified as `'Same word length as input'` or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- `s` is `1` for signed and `0` for unsigned.
- `w` is the word length in bits.
- `f` is the number of fractional bits.

The object type-casts the filter coefficients of the discrete FIR filter to the specified data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the object ignores the `Coefficients` property.

Dependencies

To enable this property, set `NumeratorSource` to `'Property'`.

`OutputDataType` — Data type of discrete FIR filter output
`'Full precision'` (default) | `'Same word length as input'` | `numerictype` object

Data type of discrete FIR filter output, specified as `'Same word length as input'`, `'Full precision'`, or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- `s` is `1` for signed and `0` for unsigned.
- `w` is the word length in bits.
- `f` is the number of fractional bits.

The object type-casts the output of the discrete FIR filter to the specified data type. The quantization uses the settings of the `RoundingMethod` and `OverflowAction` properties. When the input data type is floating point, the object ignores the `OutputDataType` property.

The object increases the word length for full precision inside each filter tap and casts the final output to the specified type. The maximum final internal data type (*WF*) depends on the input data type (*WI*), the coefficient data type (*WC*), and the number of coefficients (*L*) and is given by

WF = *WI* + *WC* + ceil(log2(*L*)).

When you specify a fixed set of coefficients, usually the actual full-precision internal word length is smaller than *WF* because the values of the coefficients limit the potential growth. When you use programmable coefficients, the object cannot calculate the dynamic range, and the internal data type is always *WF*.

Control Arguments

`ResetInputPort` — Option to enable reset input argument
`false` (default) | `true`

Option to enable reset input argument, specified as `true` or `false`. When you set this property to `true`, the object expects a value for the `reset` input argument. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

`HDLGlobalReset` — Option to connect data path registers to generated HDL global reset signal
`false` (default) | `true`

Option to connect data path registers to generated HDL global reset signal, specified as `true` or `false`. Set this property to `true` to connect the generated HDL global reset signal to the data path registers. This property does not change the arguments of the object or modify simulation behavior in MATLAB. When you set this property to `false`, the generated HDL global reset clears only the control path registers. The generated HDL global reset can be synchronous or asynchronous depending on your HDL code generation settings.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Usage

Syntax

```
[dataOut,validOut] = firFilt(dataIn,validIn)
[dataOut,validOut,ready] = firFilt(dataIn,validIn)
[dataOut,validOut] = firFilt(dataIn,validIn,coeff)
[dataOut,validOut] = firFilt(dataIn,validIn,reset)
```

Description

`[dataOut,validOut] = firFilt(dataIn,validIn)` filters the input data only when `validIn` is `true`.

`[dataOut,validOut,ready] = firFilt(dataIn,validIn)` returns `ready` set to `true` when the object is ready to accept new input data on the next call.

The object returns the `ready` argument only when you set the `FilterStructure` property to `'Partly serial systolic'`. For example:

```
firFilt = dsphdl.FIRFilter(Numerator,...
            'FilterStructure','Partly serial systolic',...
            'SerializationOption','Minimum number of cycles between valid input samples',...
            'NumCycles',8)
...
for k=1:length(dataIn)
    [dataOut,validOut,ready] = firFilt(dataIn(k),validIn(k));
```

`[dataOut,validOut] = firFilt(dataIn,validIn,coeff)` filters data using the coefficients, `coeff`. The object expects the `coeff` argument only when you set the `NumeratorSource` property to `'Input port (Parallel interface)'`. For example:

```
firFilt = dsphdl.FIRFilter(NumeratorSource,'Input Port (Parallel interface)')
...
for k=1:length(dataIn)
    Numerator = myGetNumerator();  %calculate coefficients
    [dataOut,validOut] = firFilt(dataIn(k),validIn(k),Numerator);
```

`[dataOut,validOut] = firFilt(dataIn,validIn,reset)` filters data when `reset` is `false`. When `reset` is `true`, the object resets the filter registers. The object expects the `reset` argument only when you set the `ResetInputPort` property to `true`. For example:

```
firFilt = dsphdl.FIRFilter(Numerator,'ResetInputPort',true)
...
% reset the filter
firFilt(0,false,true);
for k=1:length(dataIn)
    [dataOut,validOut] = firFilt(dataIn(k),validIn(k),false);
```

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Input Arguments

`dataIn` — Input data

scalar or column vector of real or complex values

Input data, specified as a scalar or column vector of real or complex values. When the input data type is an integer type or fixed-point type, the object uses fixed-point arithmetic for internal calculations.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`
Complex Number Support: Yes

`validIn` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

`coeff` — Filter coefficients

real or complex vector

Filter coefficients, specified as a vector of real or complex values. You can change the input coefficients at any time. The size of the vector depends on the size and symmetry of the sample coefficients specified in the `NumeratorPrototype` property. The prototype specifies a sample coefficient vector that is representative of the symmetry and zero-value locations of the expected input coefficients. The object uses the prototype to optimize the filter by sharing multipliers for symmetric or antisymmetric coefficients, and removing multipliers for zero-value coefficients. Therefore, provide only the nonduplicate coefficients in the argument. For example, if you set the `NumeratorPrototype` property to a symmetric 14-tap filter, the object expects a vector of 7 values for the `coeff` argument. You must still provide zeros in the input `coeff` vector for the nonduplicate zero-value coefficients.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Dependencies

To enable this argument, set the `NumeratorSource` property to `'Input port (Parallel interface)'`.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`reset` — Clears internal states

logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set the `ResetInputPort` property to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — Filtered output data

scalar or column vector of real or complex values

Filtered output data, returned as a scalar or column vector of real or complex values. The dimensions of the output data match the dimensions of the input data. When the input data is floating point, the output data inherits the data type of the input data. When the input data is an integer type or fixed-point type, the OutputDataType property determines the output data type.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`validOut` — Indicates valid output data

scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

`ready` — Indicates object is ready for new input data

scalar

Control signal that indicates that the object is ready for new input data sample on the next cycle. When `ready` is `1` (`true`), you can specify the `data` and `valid` inputs for the next time step. When `ready` is `0` (`false`), the object ignores any input data in the next time step.

When using the partly-serial architecture, the object processes one sample at a time. If your design waits for the object to return `ready` set to `0` (`false`) before de-asserting `validIn`, then one extra input data value arrives at the object. The object stores this extra data while processing the current data, and then does not set `ready` to `1` (`true`) until the extra input is processed.

Dependencies

To enable this argument, set the `FilterStructure` property to `'Partly serial systolic'`.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to dsphdl.FIRFilter

getLatency	Latency of FIR filter

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create HDL FIR Filter System Object with Default Settings

Create an HDL FIR filter System object with default settings.

```
firFilt = dsphdl.FIRFilter;
```

Create an input signal of random noise, and allocate memory for outputs.

```
L = 100;
dataIn = randn(L,1);
dataOut = zeros(L,1);
validOut = false(L,1);
```

Call the object on the input signal, asserting that the input data is always valid. The object processes one data sample at a time.

```
for k=1:L
    [dataOut(k),validOut(k)] = firFilt(dataIn(k),true);
end
```

Implement a Partly-Serial Streaming FIR Filter

This example shows how to configure the `dsphdl.FIRFilter` System object™ as a partly-serial 31-tap lowpass filter.

Design the filter coefficients. Then create an HDL FIR filter System object. Set the `FilterStructure` to `'Partly serial systolic'`. By default, the `SerializationOption` property is `'Minimum number of cycles between valid input samples'`, and so you must specify the serialization rule using the `NumCycles` property. To share each multiplier between 10 coefficients, set the `NumCycles` to 10.

```
numerator = firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0]);
numCycles = 10;
```

```
firFilt = dsphdl.FIRFilter('Numerator',numerator, ...
    'FilterStructure','Partly serial systolic','NumCycles',numCycles);
```

This serial filter implementation requires 10 time steps to calculate each output. Create input signals `dataIn` and `validIn` such that new data is applied only every `NumCycles` time steps.

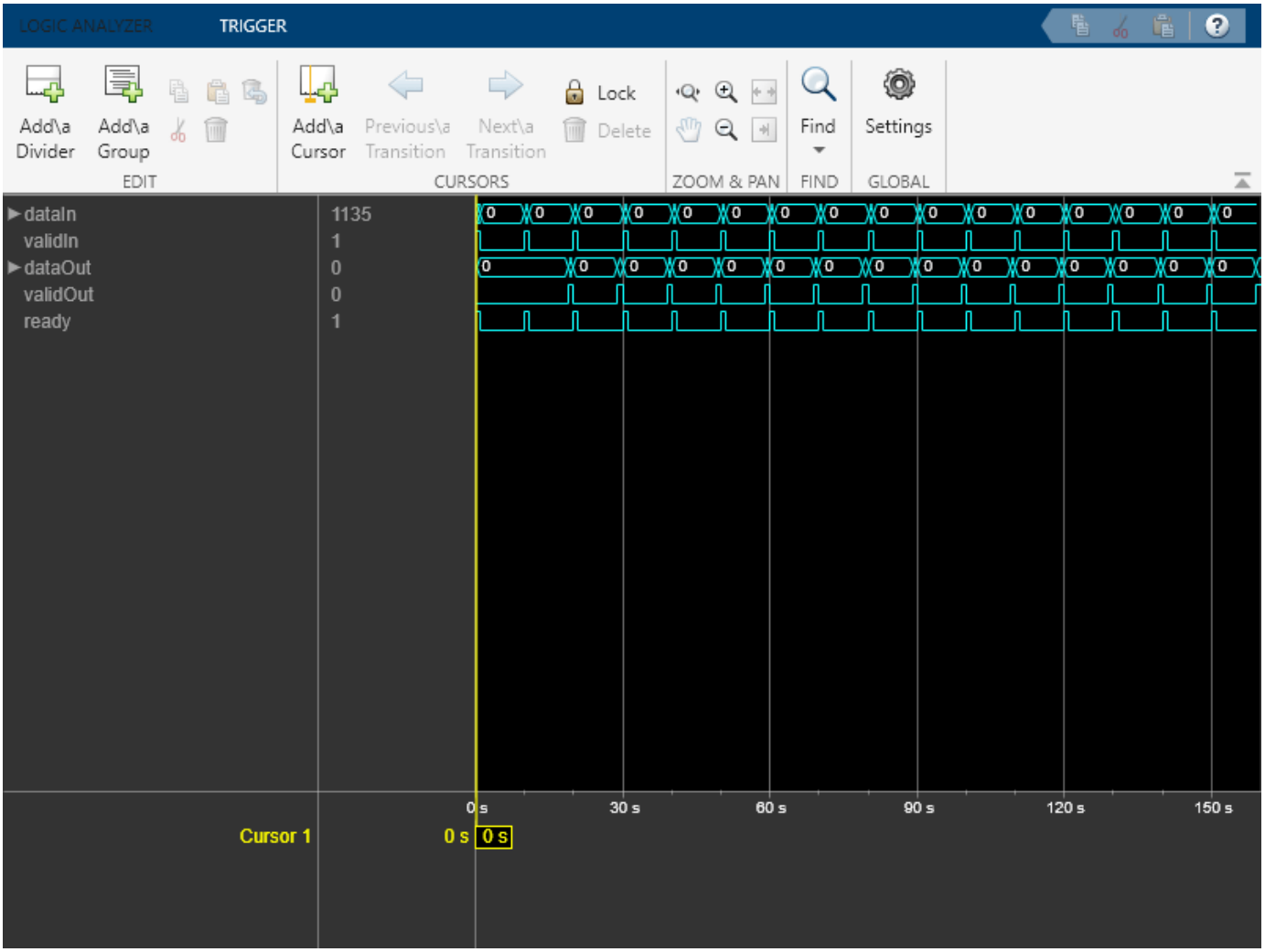
```
L = 16;
x = fi(randn(L,1),1,16);
dataIn = zeros(L*numCycles,1,'like',x);
dataIn(1:numCycles:end) = x;
validIn = false(L*numCycles,1);
validIn(1:numCycles:end) = true;
```

Create a `LogicAnalyzer` object to view the inputs and output signals.

```
la = dsp.LogicAnalyzer('NumInputPorts',5, ...
    'SampleTime',1,'TimeSpan',length(dataIn));
tags = getDisplayChannelTags(la);
modifyDisplayChannel(la,tags{1},'Name','dataIn');
modifyDisplayChannel(la,tags{2},'Name','validIn');
modifyDisplayChannel(la,tags{3},'Name','dataOut');
modifyDisplayChannel(la,tags{4},'Name','validOut');
modifyDisplayChannel(la,tags{5},'Name','ready');
```

Call the filter System object on the input signals, and view the results in the Logic Analyzer. The object models HDL pipeline registers and resource sharing, so the waveform shows an initial delay before the object returns valid output samples.

```
for k=1:length(dataIn)
    [dataOut,validOut,ready] = firFilt(dataIn(k),validIn(k));
    la(dataIn(k),validIn(k),dataOut,validOut,ready)
end
```



Create HDL FIR Filter System Object for HDL Code Generation

To generate HDL code from a System object™, create a function that contains and calls the object.

Create Function

Write a function that creates and calls an 11-tap HDL FIR filter System object. You can generate HDL code from this function.

```
function [dataOut,validOut] = HDLFIR11Tap(dataIn, validIn)
%HDLFIR11Tap
% Process one sample of data by using the dsphdl.FIRFilter System
% object.
% dataIn is a fixed-point scalar value.
% You can generate HDL code from this function.
    persistent fir
    if isempty(fir)
```

```
        Numerator = firpm(10,[0 0.1 0.5 1],[1 1 0 0]);
        fir = dsphdl.FIRFilter('Numerator',Numerator);
    end
    [dataOut,validOut] = fir(dataIn,validIn);
end
```

Create Test Bench for Function

Clear the workspace, create an input signal of random noise, and allocate memory for outputs.

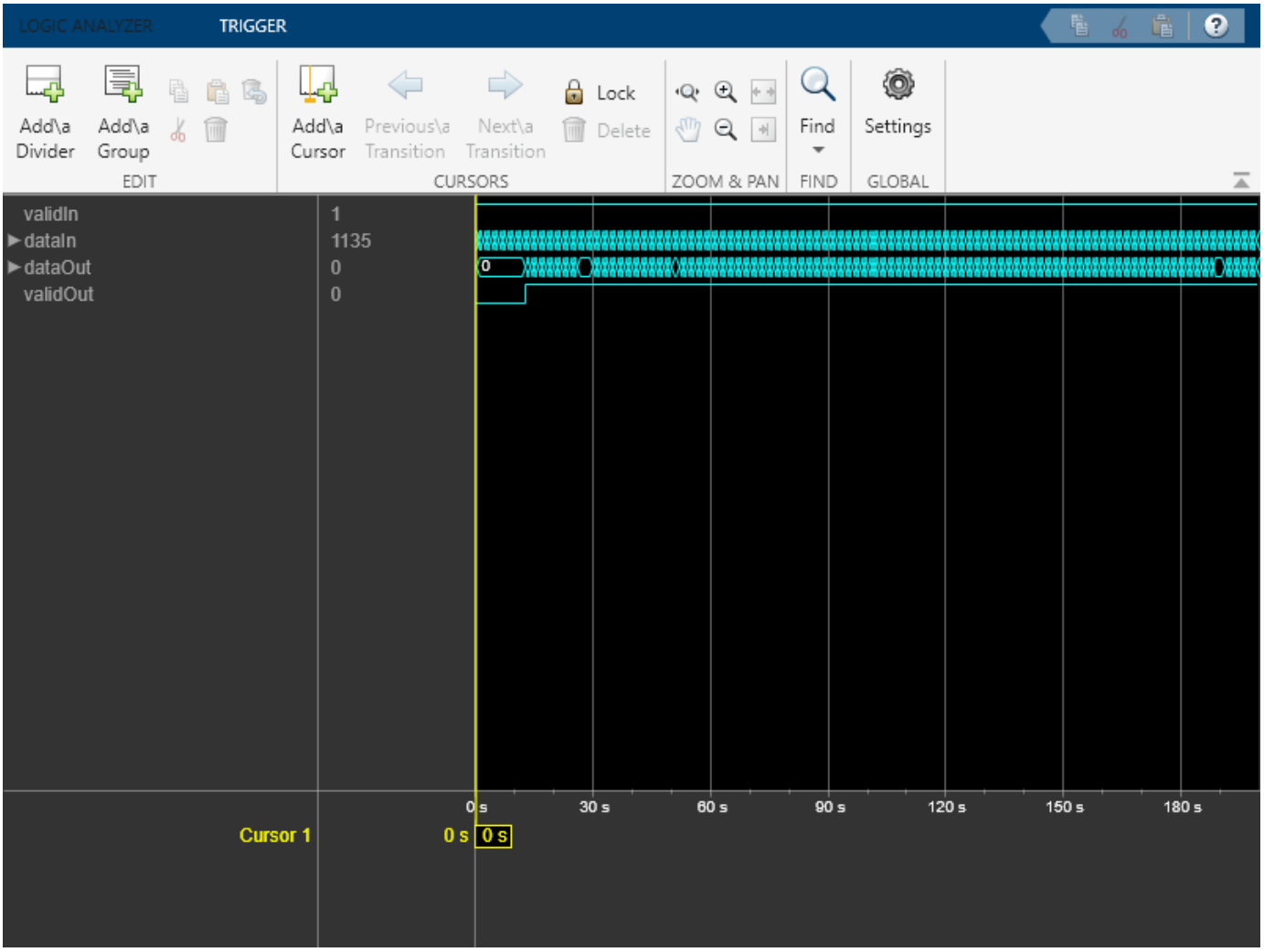
```
clear variables
clear HDLFIR11Tap
L = 200;
dataIn = fi(randn(L,1),1,16);
validIn = ones(L,1,'logical');
dataOut = fi(zeros(L,1),1,16);
validOut = false(L,1);
```

Call the function on the input signal.

```
for k = 1:L
    [dataOut(k),validOut(k)] = HDLFIR11Tap(dataIn(k), validIn(k));
end
```

Plot the signals with the Logic Analyzer.

```
la = dsp.LogicAnalyzer('NumInputPorts',4,'SampleTime',1,'TimeSpan',L);
tags = getDisplayChannelTags(la);
modifyDisplayChannel(la,tags{1},'Name','validIn');
modifyDisplayChannel(la,tags{2},'Name','dataIn');
modifyDisplayChannel(la,tags{3},'Name','dataOut');
modifyDisplayChannel(la,tags{4},'Name','validOut');
la(validIn,dataIn,dataOut,validOut)
```



Explore Latency of FIR Object

The latency of the `dsphdl.FIRFilter` System object™ varies with filter structure, serialization options, input vector size, and whether the coefficient values provide optimization opportunities. Use the `getLatency` function to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output.

Create a `dsphdl.FIRFilter` System object™ and request the latency. The default architecture is fully parallel systolic. The default data type for the coefficients is `'Same word length as input'`. Therefore, when you call the `getLatency` object function, you must specify an input data type. The object casts the coefficient values to the input data type, and then checks for symmetric coefficients. This `Numerator` has 31 symmetric coefficients, so the object optimizes for the shared coefficients, and implements 16 multipliers.

```
Numerator = firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0]);
Input_type = numerictype(1,16,15); % object uses only the word length for coefficient type cast
hdlfir = dsphdl.FIRFilter('Numerator',Numerator);
L_sysp = getLatency(hdlfir,Input_type)
```

```
L_sysp = 23
```

For the same fully parallel filter with vector input, the latency is lower. Call `getLatency` with an input vector size of four to check the latency for that case. The empty arguments are placeholders for when you use programmable coefficients or complex input data.

```
L_syspv = getLatency(hdlfir,Input_type,[],[],4)
```

```
L_syspv = 17
```

Check the latency for a partly serial systolic implementation of the same filter. By default, the `SerializationOption` property is `'Minimum number of cycles between valid input samples'`, and so you must specify the serialization rule using the `NumCycles` property. To share each multiplier between 8 coefficients, set the `NumCycles` to 8. The object then optimizes based on the coefficient symmetry, so there are 16 unique coefficients shared 8 times each over 2 multipliers. This serial filter implementation requires input samples that are valid every 8 cycles.

```
hdlfir = dsphdl.FIRFilter('Numerator',Numerator,'FilterStructure','Partly serial systolic','NumCy
L_syss = getLatency(hdlfir,Input_type)
```

```
L_syss = 19
```

Check the latency of a nonsymmetric fully parallel systolic filter. The `Numerator` has 31 coefficients.

```
Numerator = sinc(0.4*[-30:0]);
hdlfir = dsphdl.FIRFilter('Numerator',Numerator);
L_sysp = getLatency(hdlfir,Input_type)
```

```
L_sysp = 37
```

Check the latency of the same nonsymmetric filter implemented as a partly serial systolic filter. In this case, specify the `SerializationOption` by the number of multipliers. The object implements a filter that has 2 multipliers and requires 8 cycles between input samples.

```
hdlfir = dsphdl.FIRFilter('Numerator',Numerator,'FilterStructure','Partly serial systolic',...
                          'SerializationOption','Maximum number of multipliers','NumberOfMultipli
L_syss = getLatency(hdlfir,Input_type)
```

```
L_syss = 37
```

Check the latency of a fully parallel transposed architecture. The latency for this filter structure with scalar input is always 6 cycles.

```
hdlfir = dsphdl.FIRFilter('Numerator',Numerator,'FilterStructure','Direct form transposed');
L_trans = getLatency(hdlfir,Input_type)
```

```
L_trans = 6
```

The latency of the transposed filter increases with input vector size.

```
L_transv4 = getLatency(hdlfir,Input_type,[],[],4)
```

```
L_transv4 = 9
```

```
L_transv8 = getLatency(hdlfir,Input_type,[],[],16)
```

```
L_transv8 = 11
```

Algorithms

This System object implements the algorithms described on the Discrete FIR Filter block reference page.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox

Behavior changed in R2022a

Before R2022a, this System object was named `dsp.HDLFIRFilter` and was part of the DSP System Toolbox product.

High-throughput interface

This object supports high-throughput data. You can apply input data as a *N*-by-1 vector, where *N* can be up to 64 values. You cannot use frame-based input with the partly-serial architecture.

Input coefficients must be a row vector

Behavior changed in R2022a

When you use programmable coefficients with this object, you must supply the coefficients as a row vector (1-by-*N* matrix). Before R2022a, the object accepted a one-dimensional array (for example, `ones(5)`), a column vector(*M*-by-1 matrix), or a row vector of coefficients.

RAM-based party-serial architecture

This object uses a RAM-based partly-serial architecture which uses fewer resources than the former register-based architecture. Uninitialized RAM locations can result in `X` values at the start of your HDL simulation. You can avoid `X` values by having your test initialize the RAM, or by enabling the **Initialize all RAM blocks** Configuration Parameter. This parameter sets the RAM locations to `0` for simulation and is ignored by synthesis tools.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Blocks
Discrete FIR Filter

Objects
`dsp.FIRFilter`

Introduced in R2017a

dsphdl.FarrowRateConverter

Package: `dsphdl`

Polynomial sample-rate converter

Description

The `dsphdl.FarrowRateConverter` System object converts the sample rate of a signal by using FIR filters to implement a polynomial sinc approximation. A Farrow filter is an efficient rate converter when the rate conversion factor is a ratio of large integer decimation and interpolation factors. Specify the rate conversion factor by providing the input sample rate and the desired output sample rate. You can provide the rate conversion factor as a fixed property or as a time-varying input signal.

You can use this object with the default coefficients for most rate conversions. The default coefficients are a LaGrange interpolation that matches the `dsp.FarrowRateConverter` System object. Or, you can specify a custom set of coefficients if the default does not meet your specifications.

The object provides a hardware-friendly interface with input and output control signals. To provide a cycle-accurate simulation of the generated HDL code, the object models architectural latency including pipeline registers and multiplier optimizations.

To filter and resample input data with an HDL-optimized Farrow rate converter:

1. Create the `dsphdl.FarrowRateConverter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
rc = dsphdl.FarrowRateConverter
rc = dsphdl.FarrowRateConverter(Name,Value)
```

Description

`rc = dsphdl.FarrowRateConverter` creates an HDL-optimized Farrow filter System object with default properties.

`rc = dsphdl.FarrowRateConverter(Name,Value)` sets properties by using one or more name-value pairs. Enclose each property name in single quotes.

For example:

```
rc = dsphdl.FarrowRateConverter('RateChange',441e3/96e3, ...
                                'FilterStructure','Direct form transposed');
[dataOut,validOut,ready] = rc(dataIn,validIn);
```

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Main

`RateChangeSource` — Source of rate change

`'Property'` (default) | `'Input port'`

You can enter a constant rate change as a property or provide a time-varying rate change by using an input argument.

Setting this property to `'Input port'` enables the `rate` argument of the object.

`RateChange` — Rate change factor

`147/160` (default) | positive real scalar

Specify the rate change factor as a ratio of the input sample rate and the output sample rate, F_{in}/F_{out}, or provide a rational value. There are no limits on the rate change factor. Specify the data type for this value by using the `RateChangeDataType` property.

Dependencies

To enable this property, set `RateChangeSource` to `'Property'`.

Data Types: `double`

`Coefficients` — FIR filter coefficients

`[-1/6 1/2 -1/3 0;1/2 -1 -1/2 1;-1/2 1/2 1 0;1/6 0 -1/6 0]` (default) | matrix of real values

Specify FIR filter coefficients as an *M*-by-*N* matrix of real values, where *N* is the number of filters and *M* is the number of coefficients in each filter. *N* must be less than six. The object implements a polynomial of order *N* – 1. The default value is a special closed-form LaGrange solution that accomplishes most rate changes.

Data Types: `double`

`FilterStructure` — HDL filter architecture

`'Direct form systolic'` (default) | `'Direct form transposed'`

Specify the HDL filter architecture as one of these structures:

- `'Direct form systolic'` — This architecture provides a fully parallel filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For architecture details, see “Fully Parallel Systolic Architecture”.
- `'Direct form transposed'` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. For architecture details, see “Fully Parallel Transposed Architecture”.

This object implements the FIR filter stages by using the same architectures as the `dsphdl.FIRFilter` object. All implementations share multipliers for symmetric and antisymmetric coefficients and remove multipliers for zero-valued coefficients.

Data Types

`Rounding` — Rounding method for type-casting the output
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method for type-casting the output, specified as `'Floor'`, `'Ceiling'`, `'Convergent'`, `'Nearest'`, `'Round'`, or `'Zero'`. The rounding method is used when casting the output to the data type specified by the `OutputDataType` property. When the input data type is a floating-point data type, the object ignores the `RoundingMethod` property. For more details, see “Rounding Modes”.

`OverflowAction` — Overflow handling for type-casting the output
`'Wrap'` (default) | `'Saturate'`

Overflow handling for type-casting the output, specified as `'Wrap'` or `'Saturate'`. Overflow handling is used when casting the output to the data type specified by the `OutputDataType` property. When the input data type is a floating-point data type, the object ignores the `OverflowAction` property. For more details, see “Overflow Handling”.

`CoefficientsDataType` — Data type of discrete FIR filter coefficients
`'Same word length as input'` (default) | `numerictype` object

Data type of discrete FIR filter coefficients, specified as `'Same word length as input'` or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- `s` is `1` for signed and `0` for unsigned.
- `w` is the word length in bits.
- `f` is the number of fractional bits.

The object type-casts the filter coefficients of the discrete FIR filter to the specified data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is a floating-point data type, the object ignores this property.

The recommended data type for this parameter is `'Same word length as input'`. When selecting this data type, consider the size supported by the DSP blocks on your target FPGA.

`RateChangeDataType` — Data type of rate change factor
`fixdt(1,16)` (default) | `<data type expression>`

The object casts the `RateChange` property value to this data type and uses this data type to derive the data type for the internal accumulator. The accumulator data type is `fixdt(1,`*`fractionalWL`*`+1,`*`fractionalWL`*`)`, where *fractionalWL* is the fraction length of the rate change data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the object ignores this parameter.

The data type of the rate change must have at least one integer bit and one fractional bit. This data type must have enough integer bits to represent the `fsIn/fsOut` value. If the data type specified does not have enough integer bits, the object returns an error. The default setting does not specify a number of fractional bits, so the object can compute the necessary integer size. The fractional part of this data type determines the accuracy of the phase timing, but also increases the critical path. When

the rate change word length is large, you can limit hardware resources by fitting the multiplicand data type to the DSP blocks on the FPGA.

Dependencies

To enable this port, set the `RateChangeSource` parameter to `'Property'`.

`MultiplicandDataType` — Data type of multiplicand
`'Full precision'` (default) | `<data type expression>`

The object casts the output of the accumulator to this data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is a floating-point data type, the object ignores this parameter. When the rate change is large, you can limit hardware resource use by controlling the multiplicand data type. When selecting this data type, consider the size supported by the DSP blocks on your target FPGA.

`OutputDataType` — Data type of filter output
`'Same word length as input'` (default) | `<data type expression>`

The object casts the output of each filter stage to this data type. The quantization uses the settings of the `RoundingMethod` and `OverflowAction` properties. When the input data type is a floating-point data type, the object ignores this parameter.

Control Arguments

`ResetInputPort` — Option to enable reset input argument
`false` (default) | `true`

Option to enable reset input argument, specified as `true` or `false`. When you set this property to `true`, the object expects a value for the `reset` input argument. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

`HDLGlobalReset` — Option to connect data path registers to generated HDL global reset signal
`false` (default) | `true`

Option to connect data path registers to generated HDL global reset signal, specified as `true` or `false`. Set this property to `true` to connect the generated HDL global reset signal to the data path registers. This property does not change the arguments of the object or modify simulation behavior in MATLAB. When you set this property to `false`, the generated HDL global reset clears only the control path registers. The generated HDL global reset can be synchronous or asynchronous depending on your HDL code generation settings.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Usage

Syntax

```
[dataOut,validOut,ready] = rc(dataIn,validIn)
```

```
[dataOut,validOut,ready] = rc(dataIn,validIn, rate)
```

Description

`[dataOut,validOut,ready] = rc(dataIn,validIn)` filters the input data only when `validIn` is `true`, and returns `ready` set to `true` when the object is ready to accept new input data on the next call.

`[dataOut,validOut,ready] = rc(dataIn,validIn, rate)` filters data to achieve the input to output sample rate ratio, `rate`. The object expects the `rate` argument only when you set the `RateChangeSource` property to `'Input port'`. For example:

```
rc = dsphdl.FarrowRateConverter('RateChangeSource','Input port')
...
for k=1:length(dataIn)
    rate = myGetRate();  %calculate desired rate change
    [dataOut(k),validOut(k)] = rc(dataIn(k),validIn(k),rate);
```

Input Arguments

`dataIn` — Input data

real or complex scalar

Input data, specified as a real or complex scalar. When the input data type is an integer type or fixed-point type, the object uses fixed-point arithmetic for internal calculations.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`
Complex Number Support: Yes

`validIn` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

`rate` — Rate change factor

scalar

Specify the rate change factor as a rational value that is the ratio of the input sample rate and the output sample rate, F_{in}/F_{out}. There are no limits on the rate change factor.

When this input value changes, the object resets the internal phase accumulator. This reset means you can change the rate change factor from decimation to interpolation. For example, you can use this object to align data streams that have similar but varying sample clocks.

The data type of the rate change must have at least one integer bit and one fractional bit. The object derives the data type of the internal accumulator from the data type of this signal. The accumulator data type is `fixdt(1,`*`fractionalWL`*`+1,`*`fractionalWL`*`)`, where *fractionalWL* is the fraction length of the rate change data type. The *fractionalWL* determines the accuracy of the phase timing, but also increases the critical path. When the rate change word length is large, you can limit hardware resource use by fitting the multiplicand data type to the DSP blocks on the FPGA. .

Dependencies

To enable this port, set the `RateChangeSource` parameter to `'Input port'`.

Data Types: `fi`

`reset` — Clears internal states

logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set the `ResetInputPort` property to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — Filtered output data

real or complex scalar

Filtered output data, returned as a real or complex scalar. When the input data type is a floating-point data type, the output data inherits the data type of the input data. When the input data type is an integer type or fixed-point type, the OutputDataType property determines the output data type.

Data Types: `fi` | `single` | `double`

`validOut` — Indicates valid output data

scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

`ready` — Indicates object is ready for new input data

logical scalar

Control signal that indicates that the object is ready for new input data sample on the next cycle. When `ready` is `1` (`true`), you can specify the `data` and `valid` inputs for the next time step. When `ready` is `0` (`false`), the object ignores any input data in the next time step.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Algorithms

This System object implements the algorithms described on the Farrow Rate Converter block reference page.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Blocks

Farrow Rate Converter

Objects

`dsphdl.FIRFilter`

Introduced in R2022a

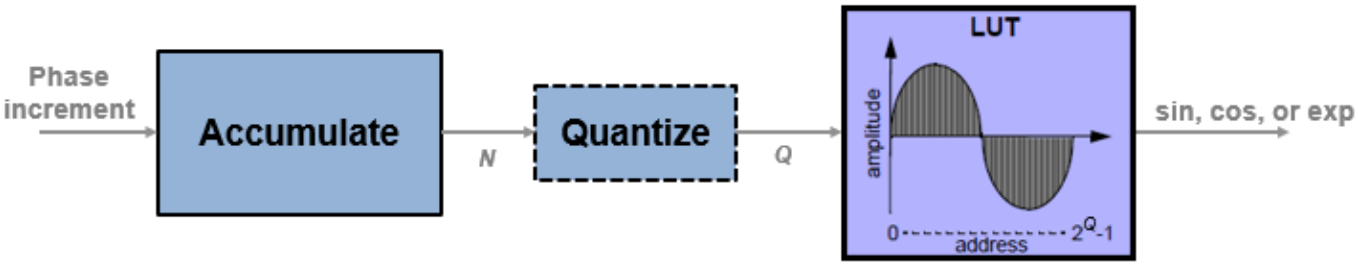
dsphdl.NCO

Package: `dsphdl`

Generate real or complex sinusoidal signals

Description

The NCO System object generates real or complex sinusoidal signals, while providing hardware-friendly control signals. A numerically-controlled oscillator (NCO) accumulates a phase increment and uses the quantized output of the accumulator as the index to a lookup table that contains the sine wave values. The wrap around of the fixed-point accumulator and quantizer data types provide periodicity of the sine wave, and quantization reduces the necessary size of the table for a given frequency resolution.



For an example of how to generate a sine wave using this System object, see “Design a HDL-Compatible NCO Source” on page 2-66. For more information on configuration and implementation, refer to the “Algorithms” on page 1-51 section.

The NCO System object provides these features.

- Optional frame-based output.
- A lookup table compression option to reduce the lookup table size. This compression results in less than one LSB loss in precision. See “Lookup Table Compression” on page 2-68 for more information.
- An optional input argument for external dither.
- An optional reset argument that resets the phase accumulator to its initial value.
- An optional output argument for the current NCO phase.

To generate real or complex sinusoidal signals:

1. Create the `dsphdl.NCO` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
hdlnco = dsphdl.NCO
hdlnco = dsphdl.NCO(Name,Value)
hdlnco = dsphdl.NCO(Inc,'PhaseIncrementSource','Property')
```

Description

`hdlnco = dsphdl.NCO` creates a numerically controlled oscillator (NCO) System object, `hdlnco`, that generates a real or complex sinusoidal signal. The amplitude of the generated signal is always 1.

`hdlnco = dsphdl.NCO(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example,

```
hdlnco = dsphdl.NCO('NumQuantizerAccumulatorBits',12, ...
                    'AccumulatorWL',16);
```

`hdlnco = dsphdl.NCO(Inc,'PhaseIncrementSource','Property')` creates an NCO with the `PhaseIncrement` property set to `Inc`, an integer scalar. To use the PhaseIncrement property, set the PhaseIncrementSource property to `'Property'`. You can add other `Name,Value` pairs before or after `PhaseIncrementSource`.

Properties

Note This object supports floating-point types for simulation but not for HDL code generation. When all input values are fixed-point type or all input arguments are disabled, the object determines the output type using the `OutputDataType` property. When any input value is floating-point type, the object ignores the `OutputDataType` property. In this case, the object returns the waveform and optional `Phase` as floating-point values.

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Waveform Generation

`PhaseIncrementSource` — Source of phase increment
`'Input port'` (default) | `'Property'`

You can set the phase increment with an input argument or by specifying a value for the property. Specify `'Property'` to configure the phase increment using the `PhaseIncrement` property. Specify `'Input port'` to set the phase increment using the `inc` argument.

`PhaseIncrement` — Phase increment for generated waveform
`100` (default) | integer

Phase increment for generated waveform, specified as an integer. The object casts this value to match the accumulator word length.

Dependencies

To enable this property, set the `PhaseIncrementSource` property to `'Property'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixdt([],N,0)`

`PhaseOffsetSource` — Source of phase offset

`'Input port'` (default) | `'Property'`

You can set the phase offset with an input argument or by specifying a value for the property. Specify `'Property'` to configure the phase increment using the `PhaseOffset` property. Specify `'Input port'` to set the phase increment using the `offset` argument.

`PhaseOffset` — Phase offset for generated waveform

`0` (default) | integer

Phase offset for the generated waveform, specified as an integer.

Dependencies

To enable this property, set the `PhaseOffsetSource` property to `'Property'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixdt([],N,0)`

`DitherSource` — Source of number of dither bits

`'Input port'` (default) | `'Property'` | `'None'`

You can set the number of dither bits from an input argument or from a property, or you can disable dither. Specify `'Property'` to configure the number of dither bits using the `NumDitherBits` property. Specify `'Input port'` to set the number of dither bits using the `dither` argument. Specify `'None'` to disable dither.

`NumDitherBits` — Bits used to express dither

`4` (default) | positive integer

Number of dither bits, specified as a positive integer.

Dependencies

To enable this property, set the `DitherSource` property to `'Property'`.

`SamplesPerFrame` — Vector size for frame-based output

`1` (default) | positive integer

Vector size for frame-based output, specified as a positive integer. When you set this value to `1`, the object has scalar input and output. When this value is greater than `1`, the `Dither` input argument must be a column vector of length `SamplesPerFrame` and the `Y` and `Phase` output arguments return column vectors of length `SamplesPerFrame`.

`LUTCompress` — Lookup table compression

`false` or `0` (default) | `true` or `1`

Lookup table compression, specified as a logical 0 (`false`) or 1 (`true`). By default, the object implements a noncompressed lookup table, and the output matches the output of the `dsp.NCO`

System object. When you enable this option, the object implements a compressed lookup table. The Sunderland compression method reduces the size of the lookup table, losing less than one LSB of precision. The spurious free dynamic range (SFDR) is empirically 1-3 dB lower than the noncompressed case. The hardware savings of the compressed lookup table allow room to improve performance by increasing the word length of the accumulator and the number of quantize bits. For details of the compression method, see “Algorithms” on page 2-67.

`Waveform` — Type of output waveform
`'Sine'` (default) | `'Cosine'` | `'Complex exponential'` | `'Sine and cosine'`

Type of output waveform. If you select `'Sine'` or `'Cosine'`, the object returns a `sin` or `cos` value. If you select `'Complex exponential'`, the output value, `exp`, is of the form `cosine + j*sine`. If you select `'Sine and cosine'`, the object returns two values, `sin` and `cos`.

When you set the `Waveform` property to `'Complex exponential'` or `'Sine and cosine'`, the object implements a 1/8 sine wave lookup table for each of the sine and cosine parts of the waveform, and uses control logic to select and invert the values to generate both sine and cosine waveforms. This optimization means that dual output mode uses similar hardware resources compared to single output mode.

`PhasePort` — Return current phase
`false` or `0` (default) | `true` or `1`

Set this property to `1` (`true`) to return the current NCO phase in the `phase` output argument. The phase is the output of the quantized accumulator with offset and increment applied. If quantization is disabled, this argument returns the output of the accumulator with offset and increment applied.

`ResetAction` — Enable reset accumulator input argument
`false` or `0` (default) | `true` or `1`

When this property is `1` (`true`), the object accepts a `ResetAccum` input argument. When the `ResetAccum` argument is `1` (`true`), the object resets the accumulator to its initial value.

Data Types

`OverflowAction` — Overflow mode for fixed-point operations
`'Wrap'` (default)

This property is read-only.

Overflow mode for fixed-point operations.

`RoundingMethod` — Rounding mode for fixed-point operations
`'Floor'` (default)

This property is read-only.

Rounding mode for fixed-point operations.

`AccumulatorDataType` — Accumulator data type
`'Binary point scaling'` (default)

This property is read-only.

Accumulator data type description. The object defines the fixed-point data type using the `AccumulatorSigned`, `AccumulatorWL`, and `AccumulatorFL` properties.

`AccumulatorSigned` — Signed or unsigned accumulator data format
`'Signed'` (default)

This property is read-only.

Signed or unsigned accumulator data format. All output is signed format.

`AccumulatorWL` — Accumulator word length
`16` (default) | integer

Accumulator word length, in bits, specified as an integer. This value must include the sign bit.

When the `PhaseQuantization` property is `0`, then `AccumulatorWL` determines the LUT size. For HDL code generation, the LUT size must be between 2 and 2^{17} entries. When you set the `LUTCompress` property to `1` (`true`), `AccumulatorWL` must be an integer in the range [5,21]. When you set the `LUTCompress` property to `0` (`false`), `AccumulatorWL` must be an integer in the range [3,19]. For more information on how this parameter affects the LUT size, see the “Lookup Table Compression” on page 2-68 section.

When you set the `PhaseQuantization` property to `1`, there is no limit to the accumulator word length property value.

`AccumulatorFL` — Accumulator fraction length
`0` (default)

This property is read-only.

Accumulator fraction length, in bits. The accumulator operates on integers. If the phase increment is fixed-point type with a fractional part, the object returns an error.

`PhaseQuantization` — Quantize accumulated phase
`false` or `0` (default) | `true` or `1`

Whether to quantize accumulated phase, specified as `1` (`true`) or `0` (`false`). When this property is enabled, the object quantizes the result of the phase accumulator to a fixed bit-width. The object uses this quantized value to select a waveform value from the lookup table. Quantizing the output of the phase accumulator enables you to reduce the lookup table size without lowering the frequency resolution. Select the size of the lookup table by using the `NumQuantizerAccumulatorBits` property.

When you disable this property, the object uses the full accumulator value as the address of the lookup table.

`NumQuantizerAccumulatorBits` — Number of quantizer accumulator bits
`12` (default) | integer

Number of quantizer accumulator bits, specified as an integer greater than 4 and less than the `AccumulatorWL` property value. For HDL code generation, this parameter value must result in a LUT size between 2 and 2^{17} entries.

When you set the `LUTCompress` property to `1` (`true`), `AccumulatorWL` must be an integer in the range [5,21]. When you set the `LUTCompress` property to `0` (`false`), `AccumulatorWL` must be an integer in the range [3,19]. For more information on how this parameter affects the LUT size, see the “Lookup Table Compression” on page 2-68 section.

When you set the `QuantizePhase` property to `true`, there is no limit to the `NumQuantizerAccumulatorBits` property value.

Dependencies

To enable this property, set the `PhaseQuantization` property to `1` (`true`).

`OutputDataType` — Output data type

`'Binary point scaling'` (default) | `'double'` | `'single'`

Output data type. If you specify `'Binary point scaling'`, the object defines the fixed-point data type using the `OutputSigned`, `OutputWL`, and `OutputFL` properties.

This parameter is ignored if any input is of floating-point type. In that case, the output data type is `double`.

`OutputSigned` — Signed or unsigned output data format

`'Signed'` (default)

This property is read-only.

Signed or unsigned output data format. All output is signed format.

`OutputWL` — Output word length

16 (default) | integer

Output word length, in bits, specified as an integer. This value must include the sign bit.

`OutputFL` — Output fraction length

14 (default) | scalar integer

Output fraction length, in bits, specified as a scalar integer.

Usage

Syntax

```
[Y,ValidOut] = hdlnco(Inc,ValidIn)
[Y,ValidOut] = hdlnco (ValidIn)
[Y,ValidOut] = hdlnco(Inc,Offset,Dither,ValidIn)
[Y,Phase,ValidOut] = hdlnco( ___ )
[ ___ ] = hdlnco( ___ ,ResetAccum,ValidIn)
```

Description

The object returns the waveform value, `Y`, as a sine value, a cosine value, a complex exponential value, or a `[Sine,Cosine]` pair of values, depending on the Waveform property.

`[Y,ValidOut] = hdlnco(Inc,ValidIn)` returns a sinusoidal signal, `Y`, generated by the HDLNCO System object, using the phase increment, `Inc`. When `ValidIn` is `true`, `Inc` is added to the accumulator. The `Inc` argument is optional. Alternatively, you can specify the phase increment as a property.

`[Y,ValidOut] = hdlnco (ValidIn)` returns a waveform, `Y`, using waveform parameters from properties rather than input arguments.

To use this syntax, set the PhaseIncrementSource, PhaseOffsetSource, and DitherSource properties to `'Property'`. These properties are independent of each other. For example:

```
hdlnco = dsphdl.NCO('PhaseIncrementSource','Property', ...
       'PhaseIncrement',phIncr,...
       'PhaseOffset',phOffset,...
       'NumDitherBits',4)
```

`[Y,ValidOut] = hdlnco(Inc,Offset,Dither,ValidIn)` returns a waveform, `Y`, with phase increment, `Inc`, phase offset, `Offset`, and dither, `Dither`.

This syntax applies when you set the PhaseIncrementSource, PhaseOffsetSource, and DitherSource properties to `'Input port'`. These properties are independent of each other. You can mix and match the activation of these arguments. `PhaseIncrementSource` is `'Input port'` by default. For example:

```
hdlnco = dsphdl.NCO('PhaseOffsetSource','Input port',...
                    'DitherSource','Input port')
for k = 1:1/Ts
   y(k) = hdlnco(phIncr,phOffset,ditherBits,true);
end
```

`[Y,Phase,ValidOut] = hdlnco( ___ )` returns a waveform, `Y`, and current phase, `Phase`. The phase is the output of the quantized accumulator.

To use this syntax, set the PhasePort property to `true`. This syntax can include any of the arguments from other syntaxes.

`[ ___ ] = hdlnco( ___ ,ResetAccum,ValidIn)` resets the accumulator value, but does not reset the output samples in the pipeline. If `ValidIn` is `true`, then the object continues to generate the output waveform starting from the reset accumulator value.

To use this syntax, set the ResetAction property to `1` (`true`). This syntax can include any of the arguments from other syntaxes.

Input Arguments

`Inc` — Phase increment

scalar integer

Phase increment, specified as a scalar integer. The object casts this value to match the accumulator word length.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Dependencies

To enable this argument, set the PhaseIncrementSource property to `'Input port'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt([],N,0)`

`ValidIn` — Control signal that enables NCO operation

scalar

Control signal that enables NCO operation, specified as a `logical` scalar. When `ValidIn` is `true`, the object increments the phase and captures any input values. When `ValidIn` is `false`, the object holds the phase accumulator and ignores any input values.

When the `SamplesPerFrame` property value is greater than `1`, this signal enables processing of `SamplesPerFrame` samples.

Data Types: `logical`

`Offset` — Phase offset

scalar integer

Phase offset, specified as a scalar integer.

`double` and `single` data types are supported for simulation but not for HDL code generation.

Dependencies

To enable this argument, set the PhaseOffsetSource property to `'Input port'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt([],N,0)`

`Dither` — Dither

scalar integer | column vector of integers

Dither, specified as an integer or as a column vector of integers. The length of the vector must equal the `SamplesPerFrame` property value.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Dependencies

To enable this argument, set the DitherSource property to `'Input port'`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt([],N,0)`

`ResetAccum` — Resets the accumulator

scalar

Control signal that resets the accumulator, specified as a `logical` scalar . When this signal is `true`, the object resets the accumulator to its initial value. This signal does not reset the output samples in the pipeline.

Dependencies

To enable this argument, set the ResetAction property to `1` (`true`).

Data Types: `logical`

Output Arguments

`Y` — Generated waveform

scalar | [*Sine*,*Cosine*] pair | vector

Generated waveform, returned as a scalar or a vector of length `SamplesPerFrame`. This argument can be a `sin` or `cos` value, an `exp` value representing `cosine + j*sine`, or a pair of arguments in the form `[Sine,Cosine]`.

If any input is of floating-point type, the object returns floating-point values for the waveform and `Phase` arguments, otherwise the object returns values using the type defined by the `OutputDataType` property.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Dependencies

By default, the output waveform is a sine wave. The format of the output waveform depends on the Waveform property.

`ValidOut` — Indicates validity of output data

scalar

Control signal that indicates validity of output data, specified as a `logical` scalar. When `validOut` is `true`, the values of `Y` and `Phase` are valid. When `validOut` is `false`, the values of `Y` and `Phase` are not valid.

When the `SamplesPerFrame` property value is greater than `1`, this signal indicates the validity of all elements in the output vectors.

Data Types: `logical`

`Phase` — Current phase of NCO

scalar | column vector

Current phase of the NCO, returned as a scalar or as a vector of length `SamplesPerFrame`. The phase is the output of the quantized accumulator with offset and increment applied. If quantization is disabled, this port returns the output of the accumulator with offset and increment applied.

The values are of type `fixdt(1,N,0)`, where `N` is the `NumQuantizerAccumulatorBits` property value. If quantization is disabled, then `N` is the `AccumulatorWL` property value.

If any input argument is floating-point type, the object returns the `Phase` argument as a floating point value. Floating-point types are supported for simulation but not for HDL code generation.

Dependencies

To enable this argument, set the PhasePort property to `1` (`true`).

Data Types: `single` | `double` | `fixdt(1,N,0)`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Design a HDL-Compatible NCO Source

This example shows how to design an HDL-compatible NCO source.

Write a function that creates and calls the System object™, based on the waveform requirements. You can generate HDL from this function.

```
function yOut = HDLNCO510(validIn)
%HDLNCO510
% Generates one sample of NCO waveform using the dsphdl.NCO System object(TM)
% validIn is a logical scalar value
% phase increment, phase offset, and dither are fixed.
% You can generate HDL code from this function.

  persistent nco510;

  if isempty(nco510)
  % Since calculation of the object parameters results in constant values, this
  % code is not included in the generated HDL. The generated HDL code for
  % the NCO object is initialized with the constant property values.

   F0 = 510;     % Target output frequency in Hz
   dphi = pi/2;  % Target phase offset
   df = 0.05;    % Frequency resolution in Hz
   minSFDR = 96; % Spurious free dynamic range(SFDR) in dB
   Ts = 1/4000;  % Sample period in seconds

   % Calculate the number of accumulator bits required for the frequency
   % resolution and the number of quantized accumulator bits to satisfy the SFDR
   % requirement.
   Nacc = ceil(log2(1/(df*Ts)));
   % Actual frequency resolution achieved = 1/(Ts*2^Nacc)
   Nqacc = ceil((minSFDR-12)/6);
   % Calculate the phase increment and offset to achieve the target frequency
   % and offset.
   phIncr = round(F0*Ts*2^Nacc);
   phOffset = 2^Nacc*dphi/(2*pi);
   nco510 = dsphdl.NCO('PhaseIncrementSource','Property', ...
        'PhaseIncrement',phIncr,...
        'PhaseOffset',phOffset,...
        'NumDitherBits',4, ...
        'NumQuantizerAccumulatorBits',Nqacc,...
        'AccumulatorWL',Nacc);
  end

  yOut = nco510(validIn);

end
```

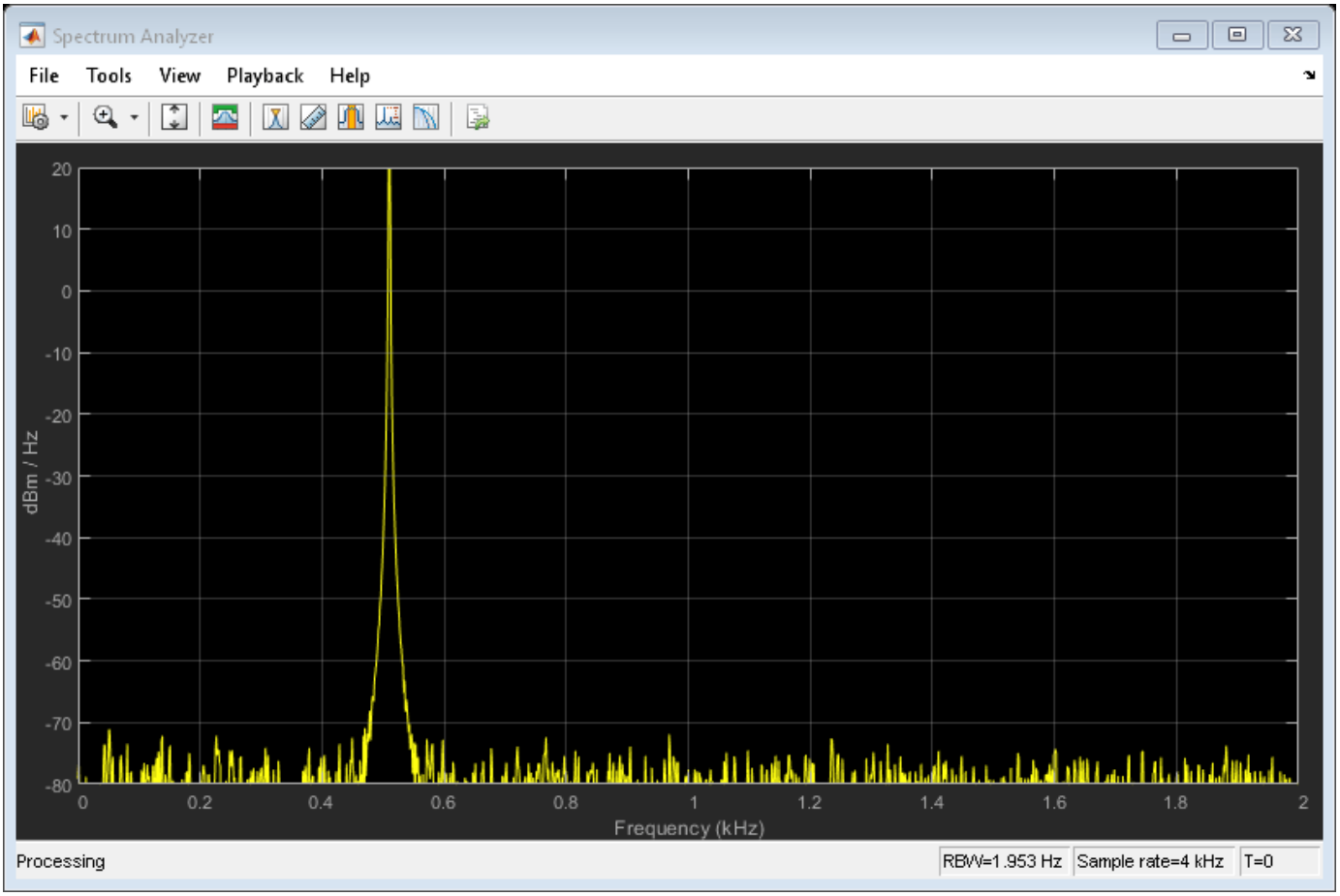
Call the object to generate data points in a sine wave. The input to the object is a valid control signal.

```
Ts = 1/4000;
y = zeros(1,1/Ts);
```

```
for k = 1:1/Ts
   y(k) = HDLNCO510(true);
end
```

Plot the mean-square spectrum of the 510 Hz sine wave generated by the NCO.

```
sa = dsp.SpectrumAnalyzer('SampleRate',1/Ts);
sa.SpectrumType = 'Power density';
sa.PlotAsTwoSidedSpectrum = false;
sa(y')
```



Algorithms

The frequency resolution of the sine wave depends on the size of the accumulator. Given a sample time, T_s, and the desired output frequency resolution Δf, calculate the necessary accumulator word length, N.

$$N = \text{ceil}\left(\log_2\left(\frac{1}{T_s \cdot \Delta f}\right)\right)$$

For a desired output frequency F_o, calculate the *phase increment*.

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

Quantizing the output of the phase accumulator enables you to reduce the lookup table size without lowering the frequency resolution. Calculate the quantized word length to achieve a desired spurious free dynamic range (*SFDR*).

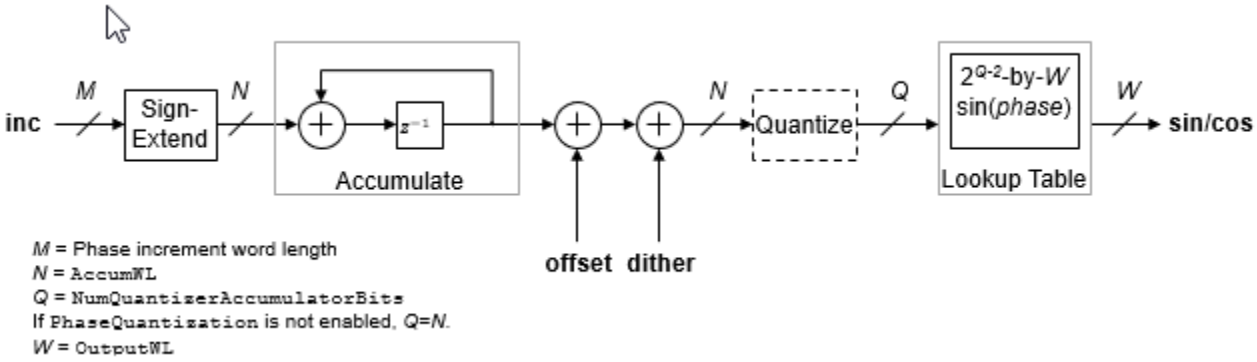
$$Q = \text{ceil}\left(\frac{SFDR - 12}{6}\right)$$

Phase offset and dither are optionally added at the accumulator stage. For a desired phase offset (in radians) of the output waveform, calculate the *phase offset* value that the object adds in the accumulator.

$$phaseoffset = \frac{2^N \cdot desiredphaseoffset}{2\pi}$$

The NCO implementation depends on whether you enable the `LUTCompress` property.

Without lookup table compression, the object uses the same quarter-sine lookup table as the `dsp.NCO` object. The size of the LUT is $2^{Q-2} \times W$ bits, where Q is `NumQuantizerAccumulatorBits` and W is `OutputWL`.



The object casts the phase increment value to match the accumulator word length.

If you do not enable `PhaseQuantization`, then Q=N, where N is `AccumulatorWL`. Consider the impact on simulator memory and hardware resources when you select these parameters.

When you set the `Waveform` property to `'Complex exponential'` or `'Sine and cosine'`, the object implements a 1/8 sine wave lookup table for each of the sine and cosine parts of the waveform, and uses control logic to select and invert the values to generate both sine and cosine waveforms. This optimization means that dual output mode uses similar hardware resources compared to single output mode.

For an example of how to generate a sine wave using this System object, see “Design a HDL-Compatible NCO Source” on page 2-66.

Lookup Table Compression

When you select lookup table (LUT) compression, the object applies the Sunderland compression method. Sunderland techniques use trigonometric identities to divide each phase of the quarter sine wave into three components and express it as:

$$\sin(A + B + C) = \sin(A + B)\cos(C) + \cos(A)\cos(B)\sin(C) - \sin(A)\sin(B)\sin(C)$$

If the quarter-sine phase has `Q-2` bits, then the phase components *A* and *B* have a word length of *LA*=*LB*=`ceil((Q-2)/3)`. Phase component *C* contains the remaining phase bits. If the phase has 12 bits, then the quarter sine phase has 10 bits, and the components are defined as:

- *A*, the four most significant bits

  $(0 \le A \le \frac{\pi}{2})$

- *B*, the next four bits

  $(0 \le B \le \frac{\pi}{2} \times 2^{-4})$

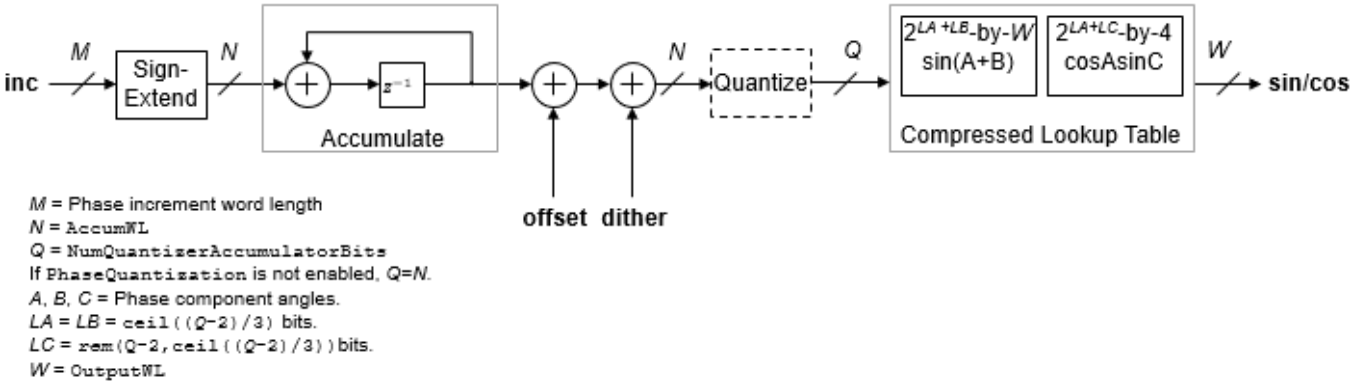
- *C*, the remaining two least significant bits

  $(0 \le C \le \frac{\pi}{2} \times 2^{-8})$

Given the relative sizes of A, B, and C, the equation can be approximated by:

$$\sin(A + B + C) \approx \sin(A + B) + \cos A \sin C$$

The object implements this equation with one LUT for $\sin(A + B)$ and one LUT for $\cos(A)\sin(C)$. The second term is a fine correction factor that you can truncate to fewer bits without losing precision. Therefore, the second LUT returns a four-bit result.



With the default accumulator size of 16 bits, and the default quantized phase width of 12 bits, the LUTs use $2^8 \times 16$ plus $2^6 \times 4$ bits (4.5 kb). For comparison, a quarter-sine lookup table without compression uses $2^{10} \times 16$ bits (16 kb). The compression approximation is accurate within one LSB, resulting in an SNR of at least 60 dB on the output. See [1].

When you set the `Waveform` property to `'Complex exponential'` or `'Sine and cosine'`, the object implements a compressed lookup table for each of the sine and cosine parts of the waveform. The hardware resource use is still smaller than dual output mode with an uncompressed table.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this System object was named `dsp.HDLNCO`.

Resource optimization for dual output mode

When you set the `Waveform` property to `'Complex exponential'` or `'Sine and cosine'`, this System object implements a 1/8 sine wave lookup table for each of the sine and cosine parts of the waveform, and uses control logic to select and invert the values to generate both sine and cosine waveforms. This optimization means that dual output mode uses similar hardware resources compared to single output mode. In previous releases, the object implemented one lookup table for each output waveform.

HDL-optimized NCO requires valid input argument
Behavior changed in R2020a

In previous releases, the input `validIn` argument of the `dsp.HDLNCO` System object was optional. It is now required. If you are using no other input ports, the object uses the `validIn` argument as an enable signal.

HDL-optimized NCO with floating-point inputs applies phase quantization
Behavior changed in R2020a

The output waveform returned from floating-point input values has changed. The output waveform now matches that returned from the same input values specified in fixed-point types.

Prior to R2020a, when using floating-point input types, the `dsp.HDLNCO` System object did not quantize the phase internally. The object expected floating-point phase increment and phase offset inputs specified in radians. Now, the object quantizes the phase internally, and you must specify the input phase increment and offset in terms of the quantized size, for both floating-point and fixed-point input types.

For example, prior to R2020a, for a floating-point HDL NCO to generate output samples with a desired output frequency of F_0 and sample frequency of F_s, you had to specify the phase increment as $2\pi(F_0/F_s)$ and phase offset as $\pi/2$.

Starting in R2020a, you must specify the phase increment and phase offset in terms of the quantized size, N. These input values are the same as the input values you use with fixed-point types. Specify the phase increment as $(F_0*2^N)/F_s$, and the phase offset as $(\pi/2)*2^N/2\pi$, or $2^N/4$.

References

[1] Cordesses, L., "Direct Digital Synthesis: A Tool for Periodic Wave Generation (Part 1)." *IEEE Signal Processing Magazine*. Volume 21, Issue 4, July 2004, pp. 50–54.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Objects
`dsp.NCO`

Blocks
NCO

Introduced in R2013a

dsphdl.ChannelSynthesizer

Package: `dsphdl`

Combine narrowband signals into multichannel signal

Description

The `dsphdl.ChannelSynthesizer` System object combines narrowband signals into a multi-channel signal using the polyphase filter bank technique. The filter bank uses a prototype lowpass filter and is implemented using a polyphase structure. You can specify the filter coefficients directly or through design parameters. The System object provides an architecture suitable for HDL code generation and hardware deployment.

The System object supports real and complex fixed-point inputs.

To combine multiple narrowband signals into a broadband signal:

1. Create the `dsphdl.ChannelSynthesizer` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
ChannelSynthesizer = dsphdl.ChannelSynthesizer
ChannelSynthesizer = dsphdl.ChannelSynthesizer(Name,Value)
```

Description

`ChannelSynthesizer = dsphdl.ChannelSynthesizer` creates a polyphase FFT synthesis filter bank System object, which combines multiple narrowband input signals into a broadband output signal.

`ChannelSynthesizer = dsphdl.ChannelSynthesizer(Name,Value)` creates a polyphase FFT synthesis filterbank object with each specified property set to the specified value. You can specify additional name-value arguments in any order as (`Name1,Value1,...,NameN,ValueN`).

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Main

`FilterCoefficients` — Polyphase filter coefficients
`[-0.0329 0.1218 0.3183 0.4829 0.5469 0.4829 0.5469 0.4829 0.3183 0.1218 -0.0329]` (default) | real- or complex-valued vector

Polyphase filter coefficients, specified as a real- or complex-valued vector. If the number of coefficients is not a multiple of the number of frequency bands or the IFFT length, the object pads this vector with zeros. The default filter specification is a raised-cosine FIR filter, `rcosdesign(0.25,2,4,'sqrt')`. You can specify a vector of coefficients or a call to a filter design function that returns the coefficient values. By default, the object casts the coefficients to the same data type as the input.

`FilterStructure` — HDL filter architecture
`'Direct form transposed'` (default) | `'Direct form systolic'`

HDL filter architecture, specified as one of these structures:

- `Direct form transposed` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. For architecture details, see “Fully Parallel Transposed Architecture”.
- `Direct form systolic` — This architecture provides a fully parallel filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For architecture details, see “Fully Parallel Systolic Architecture”.

All implementations share multipliers for symmetric and antisymmetric coefficients and remove multipliers for zero-valued coefficients.

`ComplexMultiplication` — HDL implementation of complex multipliers
`'Use 4 multipliers and 2 adders'` (default) | `'Use 3 multipliers and 5 adders'`

HDL implementation of complex multipliers, specified as either `'Use 4 multipliers and 2 adders'` or `'Use 3 multipliers and 5 adders'`. The performance of the HDL implementation depends on your synthesis tool and target device.

`Normalize` — IFFT scaling
`true` or `1` (default) | `false` or `0`

IFFT output scaling, specified as either:

- `true` — The IFFT implements an overall 1/*N* scale factor by scaling the result of each pipeline stage by 2, , where *N* is the IFFT length. This adjustment keeps the output of the IFFT in the same amplitude range as its input.
- `false` — The IFFT avoids overflow by increasing the word length by one bit at each stage.

Data Types

`RoundingMethod` — Rounding mode used for internal fixed-point calculations
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method for type-casting the output, specified as `'Floor'`, `'Ceiling'`, `'Convergent'`, `'Nearest'`, `'Round'`, or `'Zero'`. The object uses this property when casting the output to the data type specified by the `OutputDataType` property. When the input data type is floating point, the object ignores this property. For more details, see “Rounding Modes”.

`OverflowAction` — Overflow handling for type-casting the output

`'Wrap'` (default) | `'Saturate'`

Overflow handling for type-casting the output, specified as `'Wrap'` or `'Saturate'`. The object uses this property when casting the output to the data type specified by the `OutputDataType` property. When the input data type is floating point, the object ignores this property. For more details, see "Overflow Handling".

The IFFT algorithm avoids overflow by either scaling the output of each stage (Normalize enabled), or by increasing the word length by 1 bit at each stage (`Normalize` disabled).

`CoefficientsDataType` — Data type of filter coefficients

`'Same word length as input'` (default) | `numerictype` object

Data type of filter coefficients, specified as `'Same word length as input'` or a `numerictype` object. To specify a `numerictype` object, call `numerictype(`*s*`,`*w*`,`*f*`)`, where:

- *s* is `1` for signed and `0` for unsigned.
- *w* is the word length in bits.
- *f* is the number of fractional bits.

The object casts the filter coefficients to the specified data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the object ignores this property.

`OutputDataType` — Output data type

`'Full precision'` (default) | `'Same as input'` | `numerictype` object

Output data type, specified as `'Same word length as input'`, `'Full precision'`, or a `numerictype` object. To specify a `numerictype` object, call `numerictype(`*s*`,`*w*`,`*f*`)`, where:

- *s* is `1` for signed and `0` for unsigned.
- *w* is the word length in bits.
- *f* is the number of fractional bits.

The object casts the output of the filter to the specified data type. The quantization uses the settings of the `RoundingMethod` and `OverflowAction` properties. When the input data type is floating point, the object ignores this property.

When you set this property to `'Full precision'`, the System object selects a best-precision binary point by considering the values of your filter coefficients and the range of your input data type. When you set this property to `'Same as input'`, the System object casts the output of the polyphase filter to the input data type using the rounding and overflow settings you specify.

Control Arguments

`ResetInputPort` — Option to enable reset input argument

`false` (default) | `true`

Option to enable reset input argument, specified as `true` or `false`. When you set this property to `true`, the object expects a value for the `reset` input argument. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Usage

Syntax

```
[dataOut,validOut] = channelsynthesizer(dataIn,validIn)
[dataOut,validOut] = channelsynthesizer(dataIn,validIn,reset)
```

Description

`[dataOut,validOut] = channelsynthesizer(dataIn,validIn)` combines the narrowband row input `dataIn` vector signals and returns a broadband signal, `dataOut`, when `validIn` is `1` (true). The `validIn` and `validOut` arguments are logical scalars that indicate the validity of the input and output signals, respectively.

`[dataOut,validOut] = channelsynthesizer(dataIn,validIn,reset)` combines the narrowband row input `dataIn` vector signals returns a broadband signal, `dataOut`, when `validIn` is `1` (true) and `reset` is `0` (false). When `reset` is `1` (true), the object stops the current calculation and clears all internal state.

To use this syntax, set the ResetInputPort property to `true`. For example:

```
synthesizer = dsphdl.ChannelSynthesizer(...,'ResetInputPort',true);
...
[dataOut,validOut] = synthesizer(dataIn,validIn,reset)
```

Input Arguments

`dataIn` — Input data

real-valued row vector | complex-valued row vector

Input data, specified as a real-valued or complex-valued row vector.

The vector length must be a power of 2 and in the range [4, 64].

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

`validIn` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

`reset` — Clears internal states

logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the object captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set `ResetInputPort` to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — Synthesized output data

complex-valued column vector

Synthesized output data, returned as a complex-valued column vector.

When the input data type is a floating-point type, the output data inherits the data type of the input data. When the input data type is an integer type or a fixed-point type, the `OutputDataType` property controls the data type of output data.

The output size is same as the input size and it is equal to the number of frequency bands or IFFT length.

The output order is bit natural. The output data type is a result of the bit growth in the IFFT necessary to avoid overflow and the data type set in the `OutputDataType` property.

Data Types: `single` | `double` | `fi`

`validOut` — Indicates valid output data

scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to dsphdl.ChannelSynthesizer

getLatency Latency of channel synthesizer calculation

Common to All System Objects

step Run System object algorithm

release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create Synthesizer for HDL Generation

Create a function that contains a channel synthesizer object and supports HDL code generation.

Create an input sine wave signal with the specified number of frequency bands, frequency vector, and loop count.

```
numOfFrequencyBands = 8;
frequency = [-250,-180,-120,50,120,175,230,300];
loopCount = 100;
sinewave = dsp.SineWave('ComplexOutput',true,'Frequency',...
    frequency,'SamplesPerFrame',loopCount);
spectrumAnalyzer = dsp.SpectrumAnalyzer('ShowLegend',true,...
    'SampleRate',sinewave.SampleRate*numOfFrequencyBands);
```

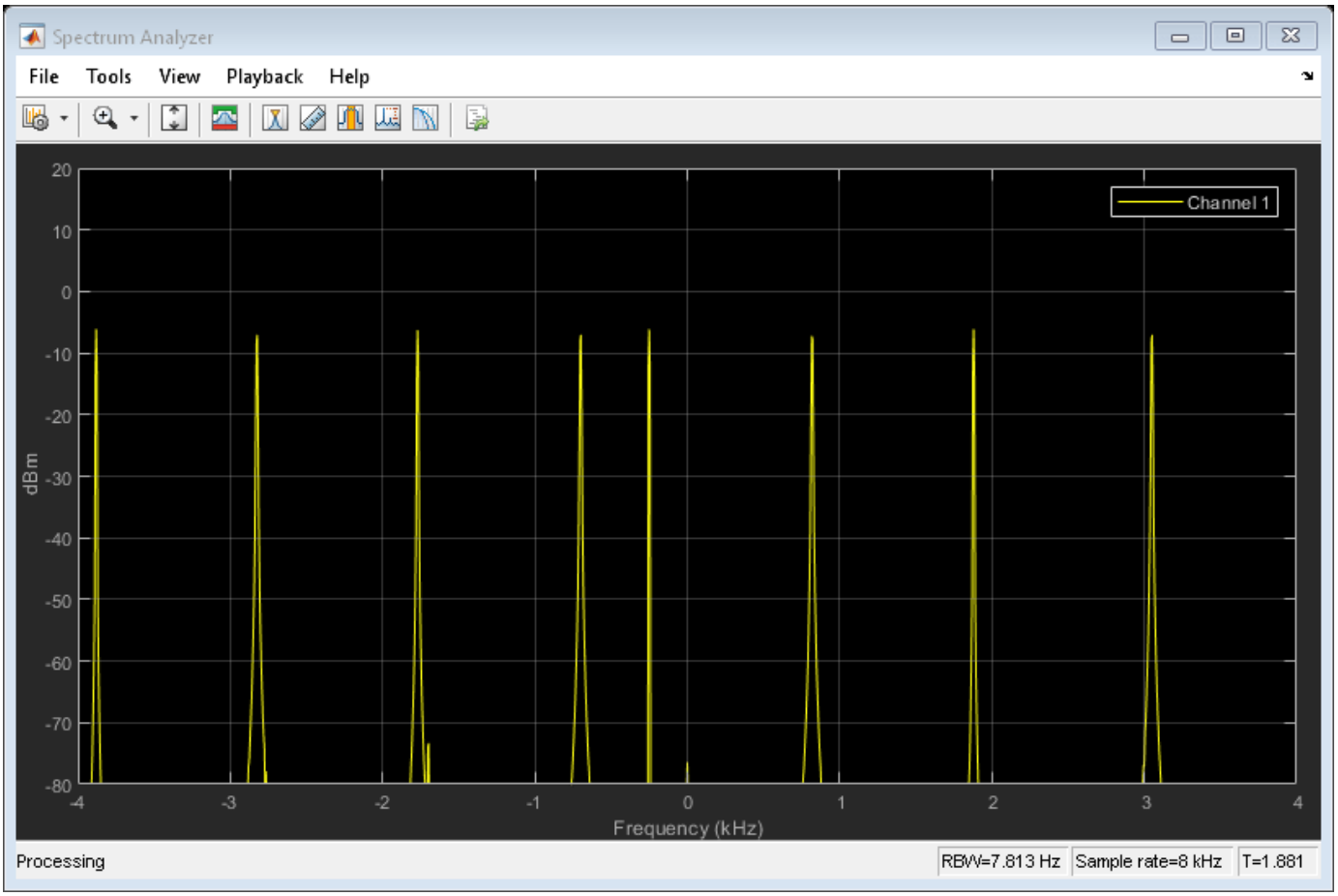
Call the function that contains the `dsphdl.ChannelSynthesizer` object. You can generate HDL code using this function.

```
function [yOut,validOut] = HDLSynthesizer8(yIn,validIn)
%HDLSynthesizer8
% Combines narrow band signals to broadband signal using the
% |dsphdl.ChannelSynthesizer| System object(TM).

% |yIn| is a fixed-point row vector and |validIn| is a logical scalar value.
% You can generate HDL code from this function.

  persistent synthesizer8;
  coder.extrinsic('tf');
  coder.extrinsic('dsp.Channelizer');

  if isempty(synthesizer8)
    % Use filter coefficients from non-HDL channelizer. You can also
    % provide your own filter coefficients.
    FilterCoefficients = tf(dsp.Channelizer('NumFrequencyBands',...
        8));
    synthesizer8 = dsphdl.ChannelSynthesizer('FilterCoefficients',...
        FilterCoefficients);
  end
  [yOut,validOut] = synthesizer8(yIn,validIn);
end
```

Synthesize the input data by calling the function.

```
y = zeros(numOfFrequencyBands,loopCount);
validOut = false(loopCount,1);

for i  = 1:20
    x = fi(sinewave(),1,16); % Each column is a sine wave signal
```

```
    for j = 1:loopCount
        [y(:,j),validOut(j)] = HDLSynthesizer8(x(j,:),true);
    end
    yValid = y(:,validOut == true);
    spectrumAnalyzer(yValid(:));
end
```



Explore Latency of Channel Synthesizer Object

The latency of the `dsphdl.ChannelSynthesizer` object varies with the IFFT length and filter structure.

Create a `dsphdl.ChannelSynthesizer` object with a direct form transposed filter structure and 16 frequency bands, and then calculate the latency.

```
synthesizerDT = dsphdl.ChannelSynthesizer('FilterStructure','Direct form transposed');
latencyDT = getLatency(synthesizerDT,16)
```

```
latencyDT = 20
```

Calculate the latency information for `dsphdl.ChannelSynthesizer` object with a direct form systolic filter structure and and 8 frequency bands.

```
synthesizerDS = dsphdl.ChannelSynthesizer('FilterStructure','Direct form systolic');
latencyDS = getLatency(synthesizerDS,8)
```

```
latencyDS = 21
```

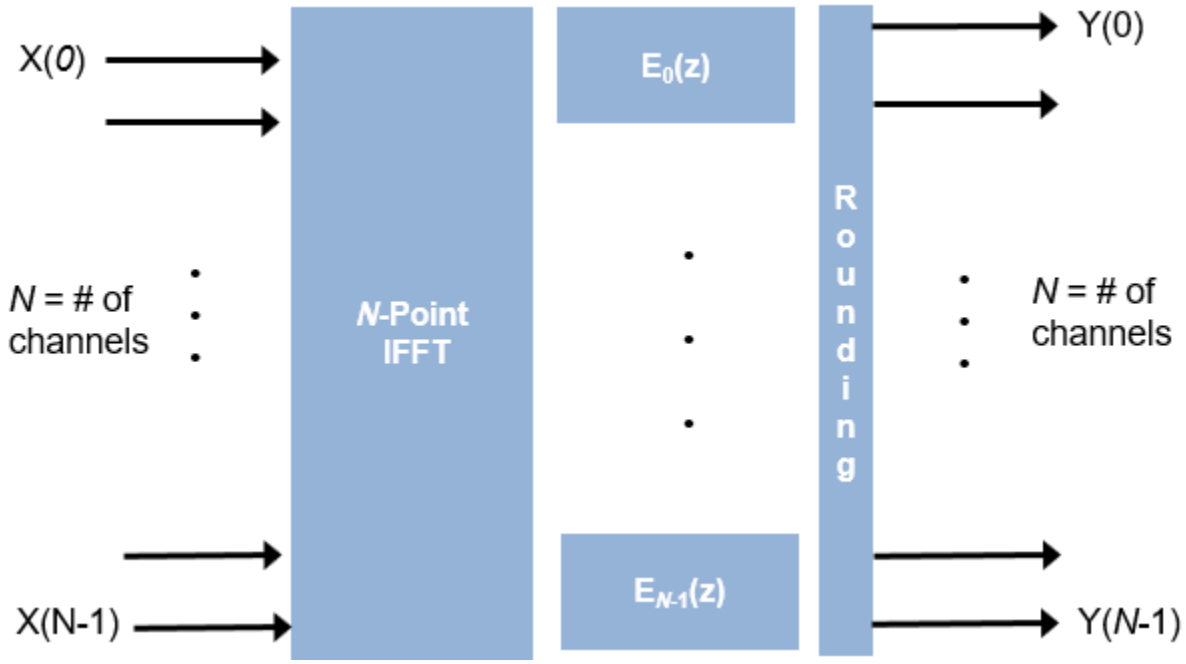
Enable scaling at each stage of the IFFT. The latency does not change.

```
synthesizerDT.Normalize = true;
latencyDTn = getLatency(synthesizerDT,16)
```

```
latencyDTn = 20
```

Algorithms

The polyphase filter algorithm requires a subfilter for each FFT channel. For more information on the polyphase filter architecture, see the Channelizer (DSP System Toolbox) block reference page.

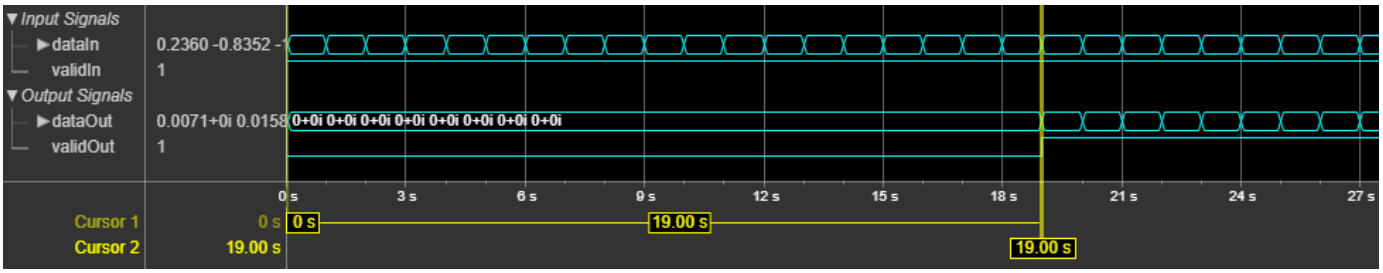


If the FFT length is N, the object implements N subfilters in the hardware. Each subfilter is an FIR filter direct form transposed or direct form systolic with Num_{Coeffs}/N taps. The object casts the output of the subfilters to the specified **OutputDataType** property by using the rounding and overflow settings you select and then pipelines filter tap in the subfilter to target the DSP sections of an FPGA.

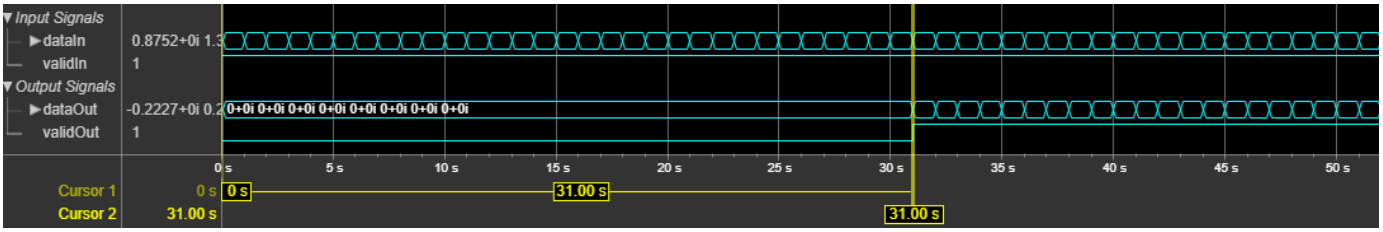
Latency

The latency varies with the input size and the filter structure. Use the `getLatency` function to find the latency of a particular configuration. Latency is the number of cycles between the first valid input and the first valid output, assuming that the input is continuous. The filter coefficients and complex multiplication do not affect the latency.

This figure shows the output of the object for a vector input length 8 when you set the `Filter structure` property to `'Direct form transposed'` and with other properties set to default values. The latency of the object is 19 clock cycles.



This figure shows the output of the object for a vector input of length 8 when you set the `Filter structure` property to `'Direct form systolic'` and with other properties set with default values. The latency of the object is 31 clock cycles.



Performance

These resource and performance data are the place-and-route results from the generated HDL targeted to the Xilinx Zynq- 7000 ZC706 evaluation board. The two examples in the tables use this common configuration:

- 1-by-8 vector
- 16-bit complex input data
- Filter structure — Direct form transposed
- Filter length — 96 coefficients
- Coefficient data type — Same word length as input
- Output data type — Same as input
- Complex multiplication (default) — Use 4 multipliers and 2 adders
- Output scaling — Enabled

The performance of the synthesized HDL code varies with your target and synthesis options.

When you set the `Filter structure` property to `'Direct form transposed'`, the design achieves a clock frequency of 382 MHz. The design uses these resources.

Resource	Number Used
LUT	1953
FFS	3833
Xilinx LogiCORE DSP48	208

When you set the `Filter structure` property to `'Direct form systolic'`, the design achieves a clock frequency of 381 MHz. The design uses these resources.

Resource	Number Used
LUT	2026
FFS	3519
Xilinx LogiCORE DSP48	208

References

[1] Harris, Fredric J. *Multirate Signal Processing for Communication Systems*. Upper Saddle River, N.J: Prentice Hall PTR, 2004.

[2] Harris, Frederic J., Chris Dick, and Michael Rice. "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications." *IEEE Transactions on Microwave Theory and Techniques*. 51, no 4, (April 2003): 1395–1412. https://doi.org/10.1109/TMTT.2003.809176.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Blocks

Channel Synthesizer | Channelizer

Objects

`dsphdl.Channelizer`

Introduced in R2021b

dsphdl.FFT

Package: `dsphdl`

Compute fast Fourier transform (FFT)

Description

The `dsphdl.FFT` System object provides two architectures to optimize either throughput or area. Use the streaming Radix 2^2 architecture for high-throughput applications. This architecture supports scalar or vector input data. You can achieve gigasamples-per-second (GSPS) throughput using vector input. Use the burst Radix 2 architecture for a minimum resource implementation, especially with large FFT sizes. Your system must be able to tolerate bursty data and higher latency. This architecture supports only scalar input data. The object accepts real or complex data, provides hardware-friendly control signals, and has optional output frame control signals.

To calculate the fast Fourier transform:

1. Create the `dsphdl.FFT` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
FFT_N = dsphdl.FFT
FFT_N = dsphdl.FFT(Name,Value)
```

Description

`FFT_N = dsphdl.FFT` returns an HDL FFT System object, `FFT_N`, that performs a fast Fourier transform.

`FFT_N = dsphdl.FFT(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Example: `fft128 = dsphdl.FFT('FFTLength',128)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Architecture` — Hardware implementation
`'Streaming Radix 2^2'` (default) | `'Burst Radix 2'`

Hardware implementation, specified as either:

- `'Streaming Radix 2^2'` — Low-latency architecture. Supports gigasamples-per-second (GSPS) throughput when you use vector input.
- `'Burst Radix 2'`— Minimum resource architecture. Vector input is not supported when you select this architecture. When you use this architecture, your input data must comply with the `ready` backpressure signal.

`ComplexMultiplication` — HDL implementation of complex multipliers
`'Use 4 multipliers and 2 adders'` (default) | `'Use 3 multipliers and 5 adders'`

HDL implementation of complex multipliers, specified as either `'Use 4 multipliers and 2 adders'` or `'Use 3 multipliers and 5 adders'`. Depending on your synthesis tool and target device, one option may be faster or smaller.

`BitReversedOutput` — Order of the output data
`true` (default) | `false`

Order of the output data, specified as either:

- `true` — The output channel elements are bit reversed relative to the input order.
- `false` — The output channel elements are in linear order.

The FFT algorithm calculates output in the reverse order to the input. When you request output in the same order as the input, the algorithm performs an extra reversal operation. For more information on ordering of the output, see “Linear and Bit-Reversed Output Order”.

`BitReversedInput` — Expected order of the input data
`false` (default) | `true`

Expected order of the input data, specified as either:

- `true` — The input channel elements are in bit-reversed order.
- `false` — The input channel elements are in linear order.

The FFT algorithm calculates output in the reverse order to the input. When you request output in the same order as the input, the algorithm performs an extra reversal operation. For more information on ordering of the output, see “Linear and Bit-Reversed Output Order”.

`Normalize` — Output scaling
`false` (default) | `true`

Output scaling, specified as either:

- `true` — The object implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by 2. This adjustment keeps the output of the FFT in the same amplitude range as its input.
- `false` — The object avoids overflow by increasing the word length by one bit after each butterfly multiplication. The bit growth is the same for both architectures.

FFTLength — Number of data points used for one FFT calculation
1024 (default) | integer power of 2 between 2^2 and 2^{16}

Number of data points used for one FFT calculation, specified as an integer power of 2 between 2^2 and 2^{16}. The object accepts FFT lengths outside this range, but they are not supported for HDL code generation.

ResetInputPort — Enable reset argument
`false` (default) | `true`

Enable `reset` input argument to the object. When `reset` is `1` (true), the object stops the current calculation and clears internal states. When the `reset` is `0` (false) and the input `valid` is `1` (true), the object captures data for processing.

StartOutputPort — Enable start output argument
`false` (default) | `true`

Enable `startOut` output argument of the object. When enabled, the object returns an additional output signal that is `true` on the first cycle of each valid output frame.

EndOutputPort — Enable end output argument
`false` (default) | `true`

Enable `endOut` output argument of the object. When enabled, the object returns an additional output signal that is `true` on the first cycle of each valid output frame.

RoundingMethod — Rounding mode used for fixed-point operations
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding mode used for fixed-point operations. When the input is any integer or fixed-point data type, the FFT algorithm uses fixed-point arithmetic for internal calculations. This option does not apply when the input is single or double type. Rounding applies to twiddle factor multiplication and scaling operations.

Usage

Syntax

```
[Y,validOut] = FFT_N(X,validIn)
[Y,validOut,ready] = FFT_N(X,validIn)
[Y,startOut,endOut,validOut] = FFT_N(X,validIn)
[Y,validOut] = FFT_N(X,validIn,resetIn)
[Y,startOut,endOut,validOut] = FFT_N(X,validIn,resetIn)
```

Description

`[Y,validOut] = FFT_N(X,validIn)` returns the FFT, `Y`, of the input, `X`, when `validIn` is `true`. `validIn` and `validOut` are logical scalars that indicate the validity of the input and output signals, respectively.

`[Y,validOut,ready] = FFT_N(X,validIn)` returns the fast Fourier transform (FFT) when using the burst Radix 2 architecture. The `ready` signal indicates when the object has memory available for new input samples. You must apply input `data` and `valid` signals only when `ready` is `1` (true). The object ignores any input `data` and `valid` signals when `ready` is `0` (false).

To use this syntax, set the `Architecture` property to `'Burst Radix 2'`. For example:

```
FFT_N = dsphdl.FFT(___,'Architecture','Burst Radix 2');
...
[y,validOut,ready] = FFT_N(x,validIn)
```

`[Y,startOut,endOut,validOut] = FFT_N(X,validIn)` also returns frame control signals `startOut` and `endOut`. `startOut` is `true` on the first sample of a frame of output data. `endOut` is `true` for the last sample of a frame of output data.

To use this syntax, set the `StartOutputPort` and `EndOutputPort` properties to `true`. For example:

```
FFT_N = dsphdl.FFT(___,'StartOutputPort',true,'EndOutputPort',true);
...
[y,startOut,endOut,validOut] = FFT_N(x,validIn)
```

`[Y,validOut] = FFT_N(X,validIn,resetIn)` returns the FFT when `validIn` is `true` and `resetIn` is `false`. When `resetIn` is `true`, the object stops the current calculation and clears all internal state.

To use this syntax set the `ResetInputPort` property to `true`. For example:

```
FFT_N = dsphdl.FFT(___,'ResetInputPort',true);
...
[y,validOut] = FFT_N(x,validIn,resetIn)
```

`[Y,startOut,endOut,validOut] = FFT_N(X,validIn,resetIn)` returns the FFT, `Y`, using all optional control signals. You can use any combination of the optional port syntaxes.

Input Arguments

X — Input data

scalar or column vector of real or complex values

Input data, specified as a scalar or column vector of real or complex values, in fixed-point or integer format. Vector input is supported with `'Streaming Radix 2^2'` architecture only. The vector size must be a power of 2 between 1 and 64, and not greater than the FFT length.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `single` | `double`

validIn — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

resetIn — Clears internal states

logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set `ResetInputPort` to `true`.

Data Types: `logical`

Output Arguments

Y — Output data
scalar or column vector of real or complex values

Output data, returned as a scalar or column vector of real or complex values. The output format matches the format of the input data.

`ready` — Indicates object is ready for new input data
logical scalar

Control signal that indicates that the object is ready for new input data sample on the next cycle. When `ready` is `1` (`true`), you can specify the `data` and `valid` inputs for the next time step. When `ready` is `0` (`false`), the object ignores any input data in the next time step. This output is returned when you select `'Burst Radix 2'` architecture.

Data Types: `logical`

`startOut` — First sample of output frame
logical scalar

First sample of output frame, returned as a logical scalar. To enable this argument, set the `StartOutputPort` property to `true`.

Data Types: `logical`

`endOut` — Last sample of output frame
logical scalar

Last sample of output frame, returned as a logical scalar. To enable this argument, set the `EndOutputPort` property to `true`.

Data Types: `logical`

`validOut` — Indicates valid output data
scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to dsphdl.FFT

getLatency	Latency of FFT calculation

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create FFT for HDL Generation

Create the specifications and input signal.

```
N = 128;
Fs = 40;
t = (0:N-1)'/Fs;
x = sin(2*pi*15*t) + 0.75*cos(2*pi*10*t);
y = x + .25*randn(size(x));
y_fixed = sfi(y,32,24);
```

Write a function that creates and calls the System object™. You can generate HDL from this function.

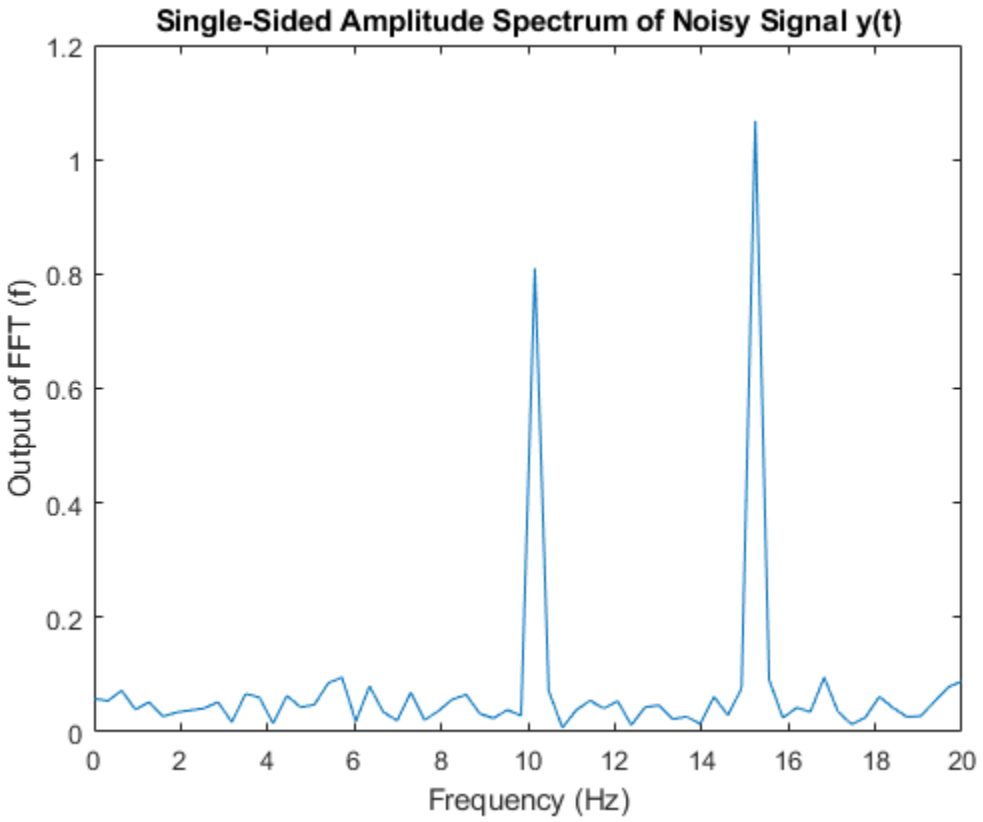
```
function [yOut,validOut] = HDLFFT128(yIn,validIn)
%HDLFFT128
% Processes one sample of FFT data using the dsphdl.FFT System object(TM)
% yIn is a fixed-point scalar or column vector.
% validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent fft128;
  if isempty(fft128)
    fft128 = dsphdl.FFT('FFTLength',128);
  end
  [yOut,validOut] = fft128(yIn,validIn);
end
```

Compute the FFT by calling the function for each data sample.

```
Yf = zeros(1,3*N);
validOut = false(1,3*N);
for loop = 1:1:3*N
    if (mod(loop, N) == 0)
        i = N;
    else
        i = mod(loop, N);
    end
    [Yf(loop),validOut(loop)] = HDLFFT128(complex(y_fixed(i)),(loop <= N));
end
```

Discard invalid data samples. Then plot the frequency channel results from the FFT.

```
Yf = Yf(validOut == 1);
Yr =  bitrevorder(Yf);
plot(Fs/2*linspace(0,1,N/2), 2*abs(Yr(1:N/2)/N))
title('Single-Sided Amplitude Spectrum of Noisy Signal y(t)')
xlabel('Frequency (Hz)')
ylabel('Output of FFT (f)')
```



Create Vector-Input FFT for HDL Generation

Create specifications and input signal. This example uses a 128-point FFT and computes the transform over 16 samples at a time.

```
N = 128;
V = 16;
Fs = 40;
t = (0:N-1)'/Fs;
x = sin(2*pi*15*t) + 0.75*cos(2*pi*10*t);
y = x + .25*randn(size(x));
y_fixed = sfi(y,32,24);
y_vect = reshape(y_fixed,V,N/V);
```

Write a function that creates and calls the System object™. The function does not need to know the vector size. The object saves the size of the input signal the first time you call it.

```
function [yOut,validOut] = HDLFFT128V16(yIn,validIn)
%HDLFFT128V16
% Processes 16-sample vectors of FFT data
% yIn is a fixed-point column vector.
% validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent fft128v16;
  if isempty(fft128v16)
    fft128v16 = dsphdl.FFT('FFTLength',128);
  end
  [yOut,validOut] = fft128v16(yIn,validIn);
end
```

Compute the FFT by passing 16-element vectors to the object. Use the `getLatency` function to find out when the first output data sample will be ready. Then, add the frame length to determine how many times to call the object. Because the object variable is inside the function, use a second object to call `getLatency`. Use the loop counter to flip `validIn` to `false` after *N* input samples.

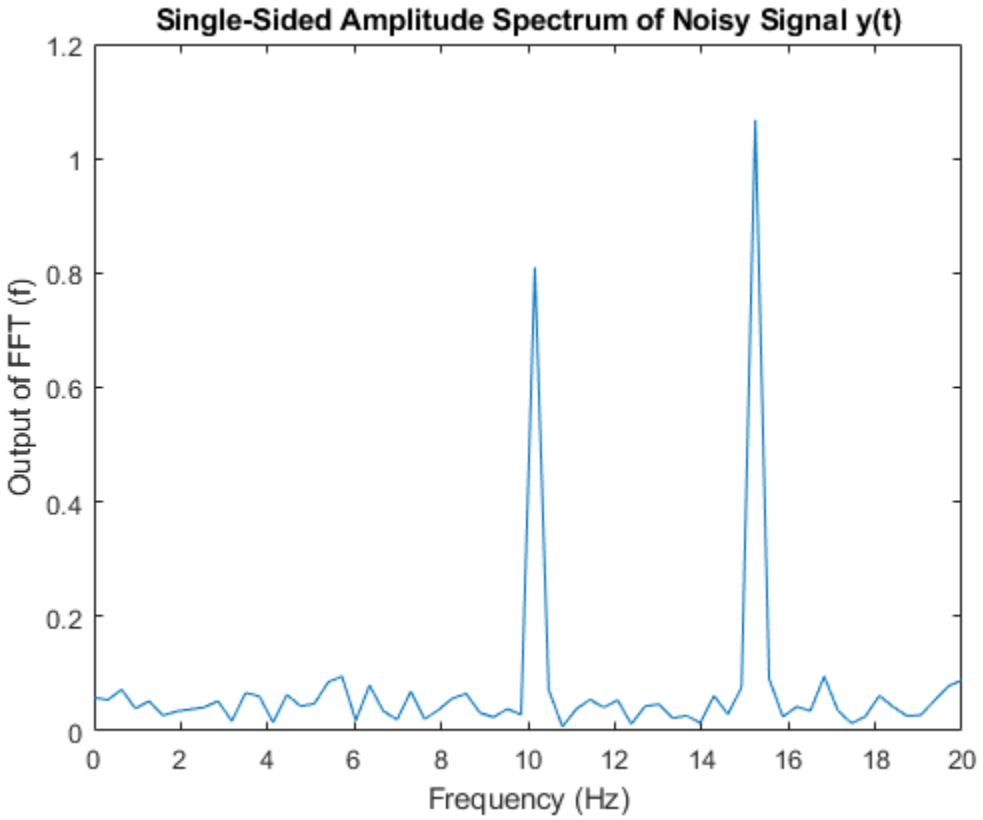
```
tempfft = dsphdl.FFT;
loopCount = getLatency(tempfft,N,V)+N/V;
Yf = zeros(V,loopCount);
validOut = false(V,loopCount);
for loop = 1:1:loopCount
    if ( mod(loop,N/V) == 0 )
        i = N/V;
    else
        i = mod(loop,N/V);
    end
    [Yf(:,loop),validOut(loop)] = HDLFFT128V16(complex(y_vect(:,i)),(loop<=N/V));
end
```

Discard invalid output samples.

```
C = Yf(:,validOut==1);
Yf_flat = C(:);
```

Plot the frequency channel data from the FFT. The FFT output is in bit-reversed order. Reorder it before plotting.

```
Yr =  bitrevorder(Yf_flat);
plot(Fs/2*linspace(0,1,N/2),2*abs(Yr(1:N/2)/N))
title('Single-Sided Amplitude Spectrum of Noisy Signal y(t)')
xlabel('Frequency (Hz)')
ylabel('Output of FFT (f)')
```



Explore Latency of HDL FFT Object

The latency of the object varies with the FFT length and the vector size. Use the `getLatency` function to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous.

Create a new `dsphdl.FFT` object and request the latency.

```
hdlfft = dsphdl.FFT('FFTLength',512);
L512 = getLatency(hdlfft)
```

```
L512 = 599
```

Request hypothetical latency information about a similar object with a different FFT length. The properties of the original object do not change.

```
L256 = getLatency(hdlfft,256)
```

```
L256 = 329
```

```
N = hdlfft.FFTLength
```

```
N = 512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(hdlfft,256,8)
```

```
L256v8 = 93
```

Enable scaling at each stage of the FFT. The latency does not change.

```
hdlfft.Normalize = true;
L512n = getLatency(hdlfft)
```

```
L512n = 599
```

Request the same output order as the input order. The latency increases because the object must collect the output before reordering.

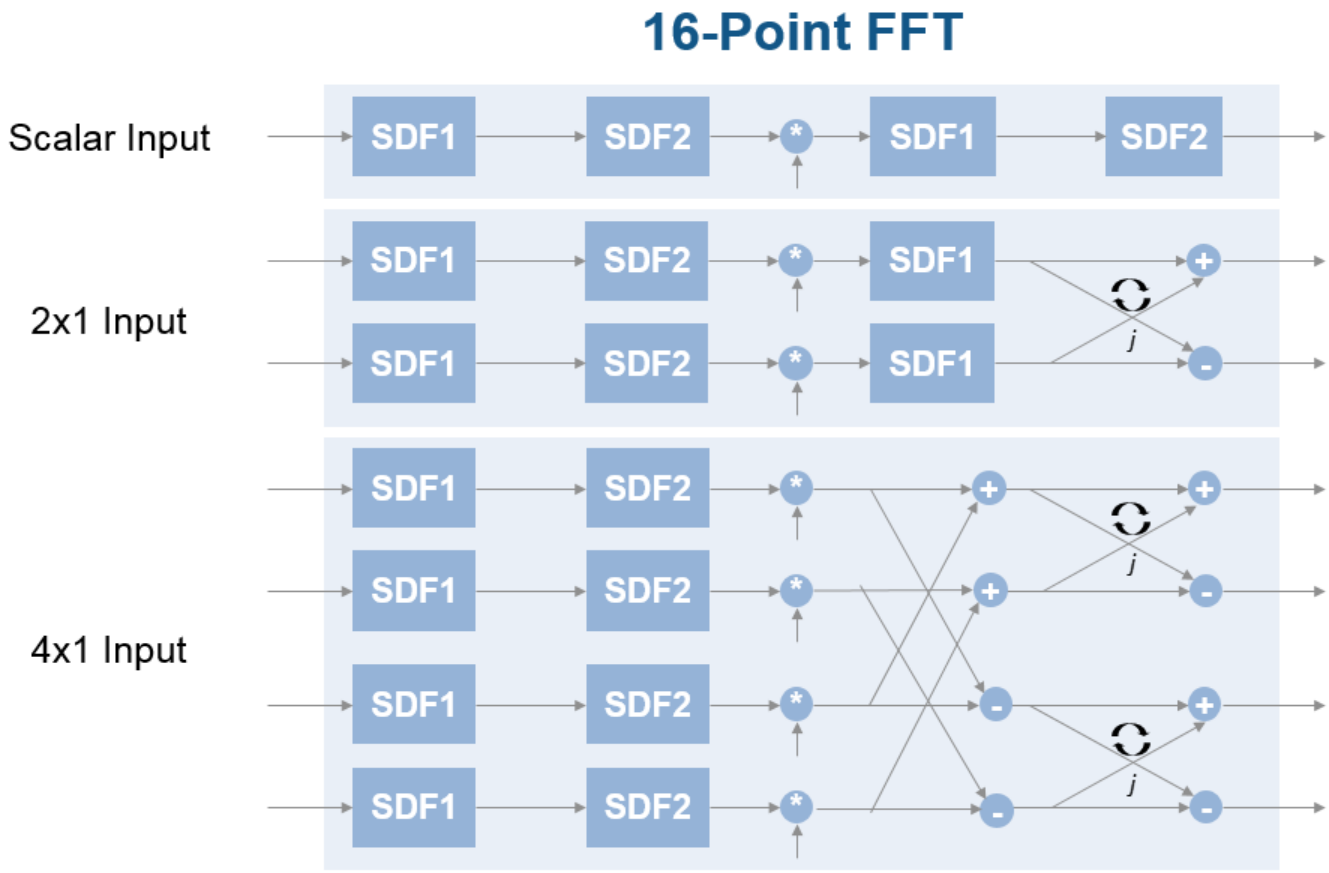
```
hdlfft.BitReversedOutput = false;
L512r = getLatency(hdlfft)
```

```
L512r = 1078
```

Algorithms

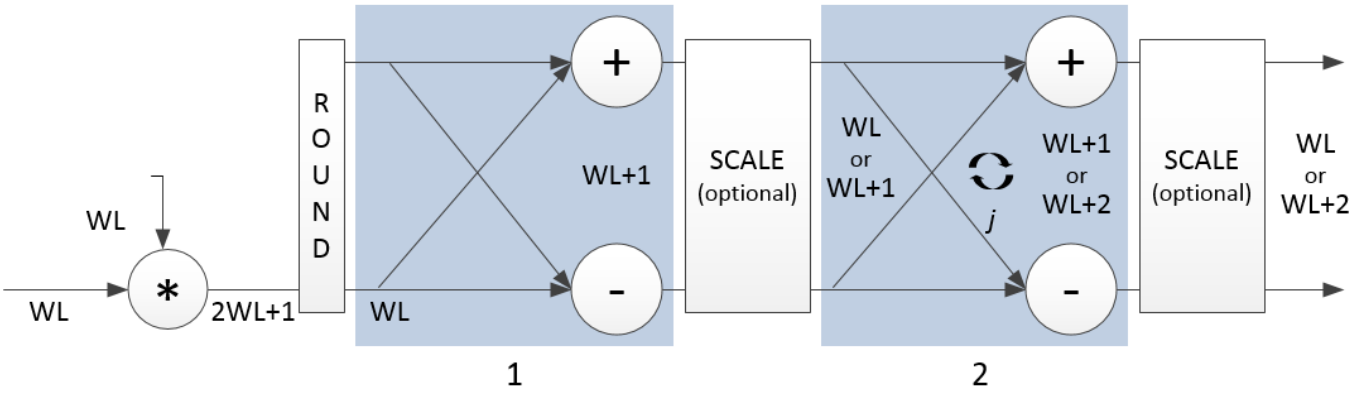
Streaming Radix 2^2

The streaming Radix 2^2 architecture implements a low-latency architecture. It saves resources compared to a streaming Radix 2 implementation by factoring and grouping the FFT equation. The architecture has $\log_4(N)$ stages. Each stage contains two single-path delay feedback (SDF) butterflies with memory controllers. When you use vector input, each stage operates on fewer input samples, so some stages reduce to a simple butterfly, without SDF.



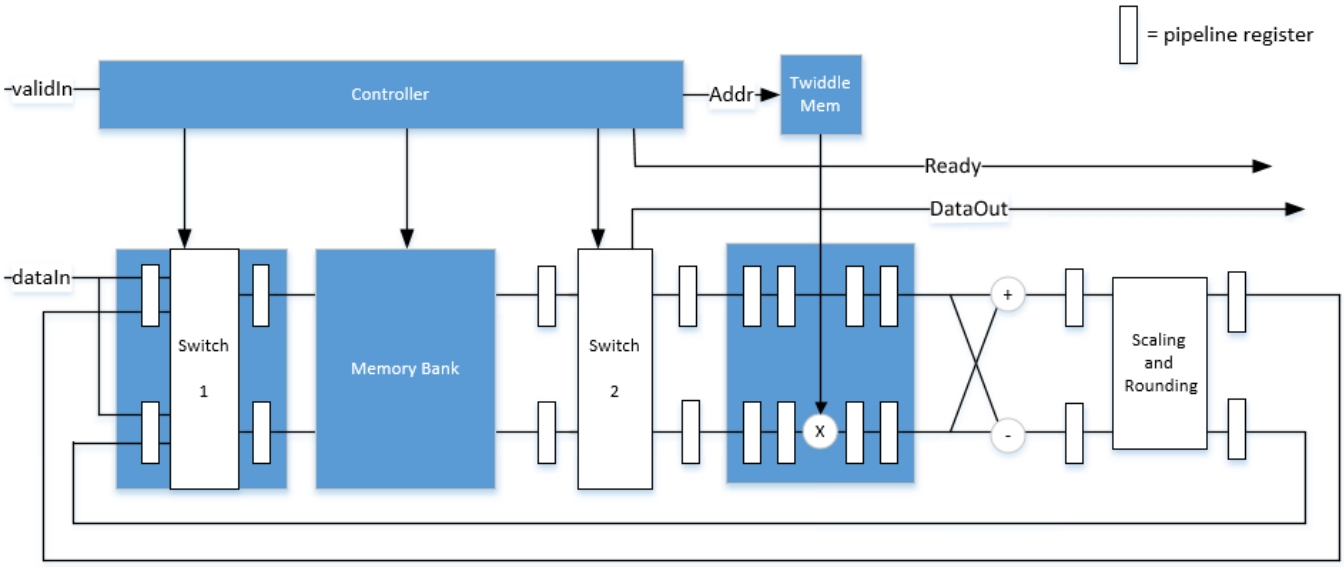
The first SDF stage is a regular butterfly. The second stage multiplies the outputs of the first stage by –*j*. To avoid a hardware multiplier, the block swaps the real and imaginary parts of the inputs, and again swaps the imaginary parts of the resulting outputs. Each stage rounds the result of the twiddle factor multiplication to the input word length. The twiddle factors have two integer bits, and the rest of the bits are used for fractional bits. The twiddle factors have the same bit width as the input data, *WL*. The twiddle factors have two integer bits, and *WL*-2 fractional bits.

If you enable scaling, the algorithm divides the result of each butterfly stage by 2. Scaling at each stage avoids overflow, keeps the word length the same as the input, and results in an overall scale factor of 1/*N*. If scaling is disabled, the algorithm avoids overflow by increasing the word length by 1 bit at each stage. The diagram shows the butterflies and internal word lengths of each stage, not including the memory.



Burst Radix 2

The burst Radix 2 architecture implements the FFT by using a single complex butterfly multiplier. The algorithm cannot start until it has stored the entire input frame, and it cannot accept the next frame until computations are complete. The output **ready** port indicates when the algorithm is ready for new data. The diagram shows the burst architecture, with pipeline registers.



When you use this architecture, your input data must comply with the **ready** backpressure signal.

Control Signals

The algorithm processes input data only when the input **valid** port is 1. Output data is valid only when the output **valid** port is 1.

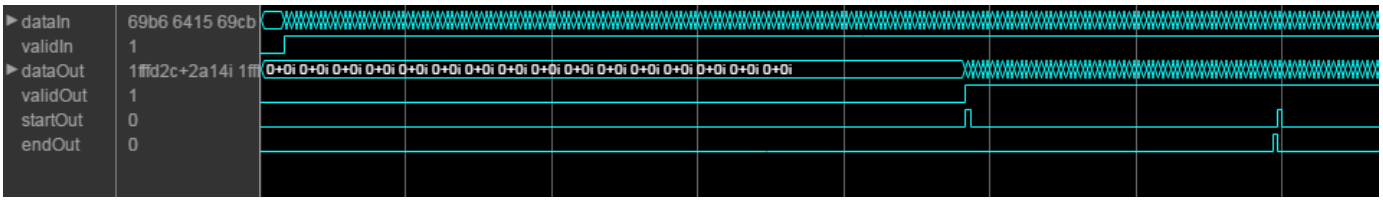
When the optional input **reset** port is 1, the algorithm stops the current calculation and clears all internal states. The algorithm begins new calculations when **reset** port is 0 and the input **valid** port starts a new frame.

Timing Diagram

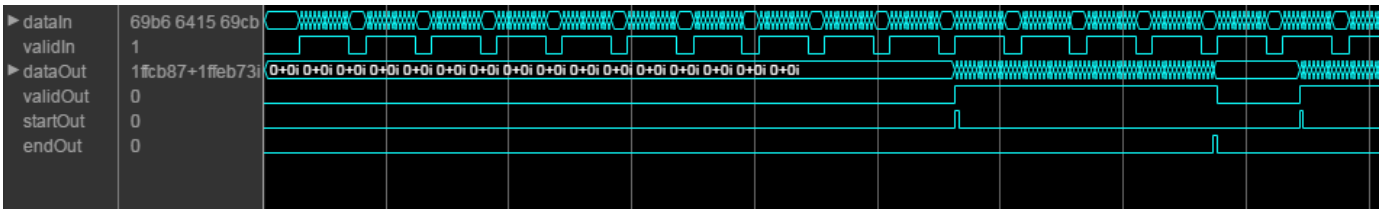
This diagram shows the input and output **valid** port values for contiguous scalar input data, streaming Radix 2^2 architecture, an FFT length of 1024, and a vector size of 16.

The diagram also shows the optional **start** and **end** port values that indicate frame boundaries. If you enable the **start** port, the **start** port value pulses for one cycle with the first valid output of the frame. If you enable the **end** port, the **start** port value pulses for one cycle with the last valid output of the frame.

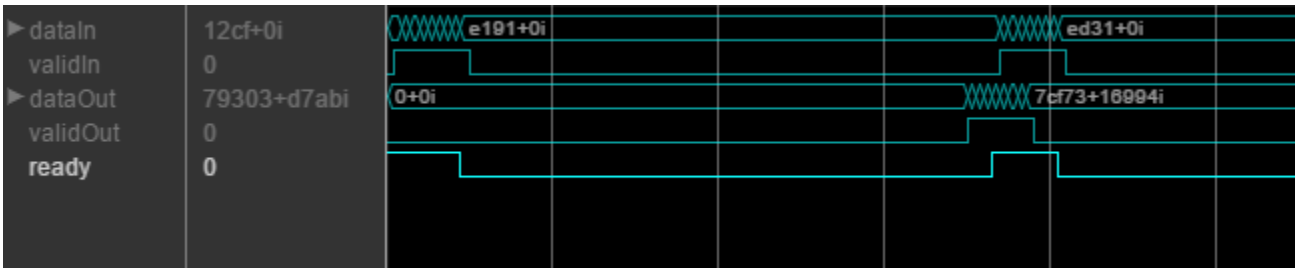
If you apply continuous input frames, the output will also be continuous after the initial latency.



The input **valid** port can be noncontiguous. Data accompanied by an input **valid** port is processed as it arrives, and the resulting data is stored until a frame is filled. Then the algorithm returns contiguous output samples in a frame of *N* (**FFT length**) cycles. This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16.



When you use the burst architecture, 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. You must apply input **data** and **valid** signals only when **ready** is `1` (true). The algorithm ignores any input **data** and **valid** signals when **ready** is `0` (false).

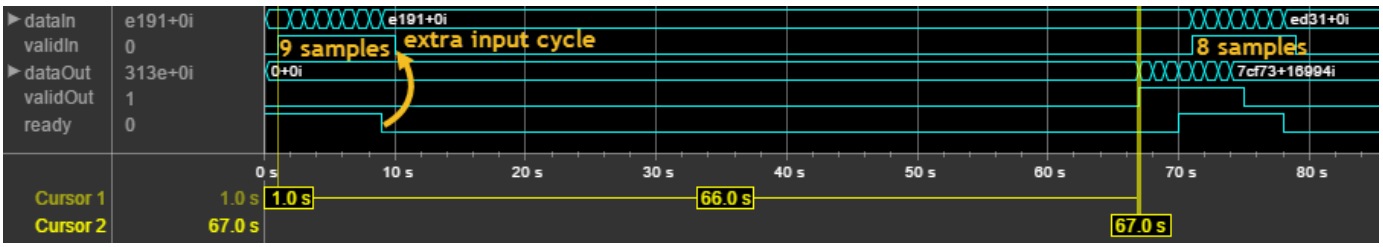


Latency

The latency varies with the FFT length and the vector size. Use the `getLatency` function to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous.

When using the burst architecture with a contiguous input, if your design waits for **ready** to output `0` before de-asserting the input **valid**, then one extra cycle of data arrives at the input. This data sample

is the first sample of the next frame. The algorithm can save one sample while processing the current frame. Due to this one sample advance, the observed latency of the later frames (from input **valid** to output **valid**) is one cycle shorter than the reported latency. The latency is measured from the first cycle, when input **valid** is 1 to the first cycle when output **valid** is 1. The number of cycles between when **ready** port is 0 and the output **valid** port is 1 is always *latency* – *FFTLength*.



Performance

This resource and performance data is the synthesis result from the generated HDL targeted to a Xilinx Virtex-6 (XC6VLX75T-1FF484) FPGA. The examples in the tables have this configuration:

- 1024 FFT length (default)
- Complex multiplication using 4 multipliers, 2 adders
- Output scaling enabled
- Natural order input, Bit-reversed output
- 16-bit complex input data
- Clock enables minimized (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options. For instance, reordering for a natural-order output uses more RAM than the default bit-reversed output, and real input uses less RAM than complex input.

For a scalar input Radix 2^2 configuration, the design achieves 326 MHz clock frequency. The latency is 1116 cycles. The design uses these resources.

Resource	Number Used
LUT	4597
FFS	5353
Xilinx LogiCORE DSP48	12
Block RAM (16K)	6

When you vectorize the same Radix 2^2 implementation to process two 16-bit input samples in parallel, the design achieves 316 MHz clock frequency. The latency is 600 cycles. The design uses these resources.

Resource	Number Used
LUT	7653
FFS	9322
Xilinx LogiCORE DSP48	24

Resource	Number Used
Block RAM (16K)	8

The block supports scalar input data only when implementing burst Radix 2 architecture. The burst design achieves 309 MHz clock frequency. The latency is 5811 cycles. The design uses these resources.

Resource	Number Used
LUT	971
FFS	1254
Xilinx LogiCORE DSP48	3
Block RAM (16K)	6

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this System object was named `dsp.HDLFFT` and was part of the DSP System Toolbox product.

FFT length of 4
Behavior changed in R2022a

This System object now supports an FFT length of 4. In previous releases the FFT length had to be a power of 2 from 2^3 to 2^{16}.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Objects
`dsp.HDLChannelizer` | `dsp.HDLIFFT` | `dsp.FFT`

Blocks
FFT | IFFT

Introduced in R2014b

dsphdl.IFFT

Package: `dsphdl`

Compute inverse fast Fourier transform (IFFT)

Description

The `dsphdl.IFFT` System object provides two architectures to optimize either throughput or area. Use the streaming Radix 2^2 architecture for high-throughput applications. This architecture supports scalar or vector input data. You can achieve gigasamples-per-second (GSPS) throughput using vector input. Use the burst Radix 2 architecture for a minimum resource implementation, especially with large FFT sizes. Your system must be able to tolerate bursty data and higher latency. This architecture supports only scalar input data. The object accepts real or complex data, provides hardware-friendly control signals, and has optional output frame control signals.

To calculate the inverse fast Fourier transform:

1. Create the `dsphdl.IFFT` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
IFFT_N = dsphdl.IFFT
IFFT_N = dsphdl.IFFT(Name,Value)
```

Description

`IFFT_N = dsphdl.IFFT` returns an HDL IFFT System object, `IFFT_N`, that performs a fast Fourier transform.

`IFFT_N = dsphdl.IFFT(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Example: `ifft128 = dsphdl.IFFT('FFTLength',128)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Architecture` — Hardware implementation
`'Streaming Radix 2^2'` (default) | `'Burst Radix 2'`

Hardware implementation, specified as either:

- `'Streaming Radix 2^2'` — Low-latency architecture. Supports gigasamples-per-second (GSPS) throughput when you use vector input.
- `'Burst Radix 2'`— Minimum resource architecture. Vector input is not supported when you select this architecture. When you use this architecture, your input data must comply with the `ready` backpressure signal.

`ComplexMultiplication` — HDL implementation of complex multipliers
`'Use 4 multipliers and 2 adders'` (default) | `'Use 3 multipliers and 5 adders'`

HDL implementation of complex multipliers, specified as either `'Use 4 multipliers and 2 adders'` or `'Use 3 multipliers and 5 adders'`. Depending on your synthesis tool and target device, one option may be faster or smaller.

`BitReversedOutput` — Order of the output data
`true` (default) | `false`

Order of the output data, specified as either:

- `true` — The output channel elements are bit reversed relative to the input order.
- `false` — The output channel elements are in linear order.

The IFFT algorithm calculates output in the reverse order to the input. When you request output in the same order as the input, the algorithm performs an extra reversal operation. For more information on ordering of the output, see “Linear and Bit-Reversed Output Order”.

`BitReversedInput` — Expected order of the input data
`false` (default) | `true`

Expected order of the input data, specified as either:

- `true` — The input channel elements are in bit-reversed order.
- `false` — The input channel elements are in linear order.

The IFFT algorithm calculates output in the reverse order to the input. When you request output in the same order as the input, the algorithm performs an extra reversal operation. For more information on ordering of the output, see “Linear and Bit-Reversed Output Order”.

`Normalize` — Output scaling
`true` (default) | `false`

Output scaling, specified as either:

- `true` — The object implements an overall 1/*N* scale factor by dividing the output of each butterfly multiplication by 2. This adjustment keeps the output of the IFFT in the same amplitude range as its input.
- `false` — The object avoids overflow by increasing the word length by one bit after each butterfly multiplication. The bit growth is the same for both architectures.

FFTLength — Number of data points used for one FFT calculation
1024 (default) | integer power of 2 between 2^2 and 2^{16}

Number of data points used for one FFT calculation, specified as an integer power of 2 between 2^2 and 2^{16}. The object accepts FFT lengths outside this range, but they are not supported for HDL code generation.

ResetInputPort — Enable reset argument
`false` (default) | `true`

Enable `reset` input argument to the object. When `reset` is `1` (true), the object stops the current calculation and clears internal states. When the `reset` is `0` (false) and the input `valid` is `1` (true), the object captures data for processing.

StartOutputPort — Enable start output argument
`false` (default) | `true`

Enable `startOut` output argument of the object. When enabled, the object returns an additional output signal that is `true` on the first cycle of each valid output frame.

EndOutputPort — Enable end output argument
`false` (default) | `true`

Enable `endOut` output argument of the object. When enabled, the object returns an additional output signal that is `true` on the first cycle of each valid output frame.

RoundingMethod — Rounding mode used for fixed-point operations
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding mode used for fixed-point operations. When the input is any integer or fixed-point data type, the IFFT algorithm uses fixed-point arithmetic for internal calculations. This option does not apply when the input is single or double type. Rounding applies to twiddle factor multiplication and scaling operations.

Usage

Syntax

```
[Y,validOut] = IFFT_N(X,validIn)
[Y,validOut,ready] = IFFT_N(X,validIn)
[Y,startOut,endOut,validOut] = IFFT_N(X,validIn)
[Y,validOut] = IFFT_N(X,validIn,resetIn)
[Y,startOut,endOut,validOut] = IFFT_N(X,validIn,resetIn)
```

Description

`[Y,validOut] = IFFT_N(X,validIn)` returns the inverse fast Fourier transform (IFFT), `Y`, of the input, `X`, when `validIn` is `true`. `validIn` and `validOut` are logical scalars that indicate the validity of the input and output signals, respectively.

`[Y,validOut,ready] = IFFT_N(X,validIn)` returns the inverse fast Fourier transform (IFFT) when using the burst Radix 2 architecture. The `ready` signal indicates when the object has memory available to accept new input samples. You must apply input `data` and `valid` signals only when `ready` is `1` (true). The object ignores the input `data` and `valid` signals when `ready` is `0` (false).

To use this syntax, set the `Architecture` property to `'Burst Radix 2'`. For example:

```
IFFT_N = dsphdl.IFFT(___,'Architecture','Burst Radix 2');
...
[y,validOut,ready] = IFFT_N(x,validIn)
```

`[Y,startOut,endOut,validOut] = IFFT_N(X,validIn)` also returns frame control signals `startOut` and `endOut`. `startOut` is `true` on the first sample of a frame of output data. `endOut` is `true` for the last sample of a frame of output data.

To use this syntax, set the `StartOutputPort` and `EndOutputPort` properties to `true`. For example:

```
IFFT_N = dsphdl.IFFT(___,'StartOutputPort',true,'EndOutputPort',true);
...
[y,startOut,endOut,validOut] = IFFT_N(x,validIn)
```

`[Y,validOut] = IFFT_N(X,validIn,resetIn)` returns the IFFT, `Y`, when `validIn` is `true` and `resetIn` is `false`. When `resetIn` is `true`, the object stops the current calculation and clears all internal state.

To use this syntax, set the `ResetInputPort` property to `true`. For example:

```
IFFT_N = dsphdl.IFFT(___,'ResetInputPort',true);
...
[y,validOut] = IFFT_N(x,validIn,resetIn)
```

`[Y,startOut,endOut,validOut] = IFFT_N(X,validIn,resetIn)` returns the IFFT, `Y`, using all optional control signals. You can use any combination of the optional port syntaxes.

Input Arguments

X — Input data

scalar or column vector of real or complex values

Input data, specified as a scalar or column vector of real or complex values, in fixed-point or integer format. Vector input is supported with `'Streaming Radix 2^2'` architecture only. The vector size must be a power of 2 between 1 and 64 that is not greater than the FFT length.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `single` | `double`

validIn — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

resetIn — Clears internal states

logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set `ResetInputPort` to `true`.

Data Types: `logical`

Output Arguments

Y — Output data
scalar or column vector of real or complex values

Output data, returned as a scalar or column vector of real or complex values. The output format matches the format of the input data.

`ready` — Indicates object is ready for new input data
logical scalar

Control signal that indicates that the object is ready for new input data sample on the next cycle. When `ready` is `1` (`true`), you can specify the `data` and `valid` inputs for the next time step. When `ready` is `0` (`false`), the object ignores any input data in the next time step. This output is returned when you select `'Burst Radix 2'` architecture.

Data Types: `logical`

`startOut` — First sample of output frame
logical scalar

First sample of output frame, returned as a logical scalar. To enable this argument, set the `StartOutputPort` property to `true`.

Data Types: `logical`

`endOut` — Last sample of output frame
logical scalar

Last sample of output frame, returned as a logical scalar. To enable this argument, set the EndOutputPort property to `true`.

Data Types: `logical`

`validOut` — Indicates valid output data
scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to dsphdl.IFFT

getLatency Latency of FFT calculation

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create IFFT for HDL Code Generation

Create the specifications and input signal. This example uses a 128-point FFT.

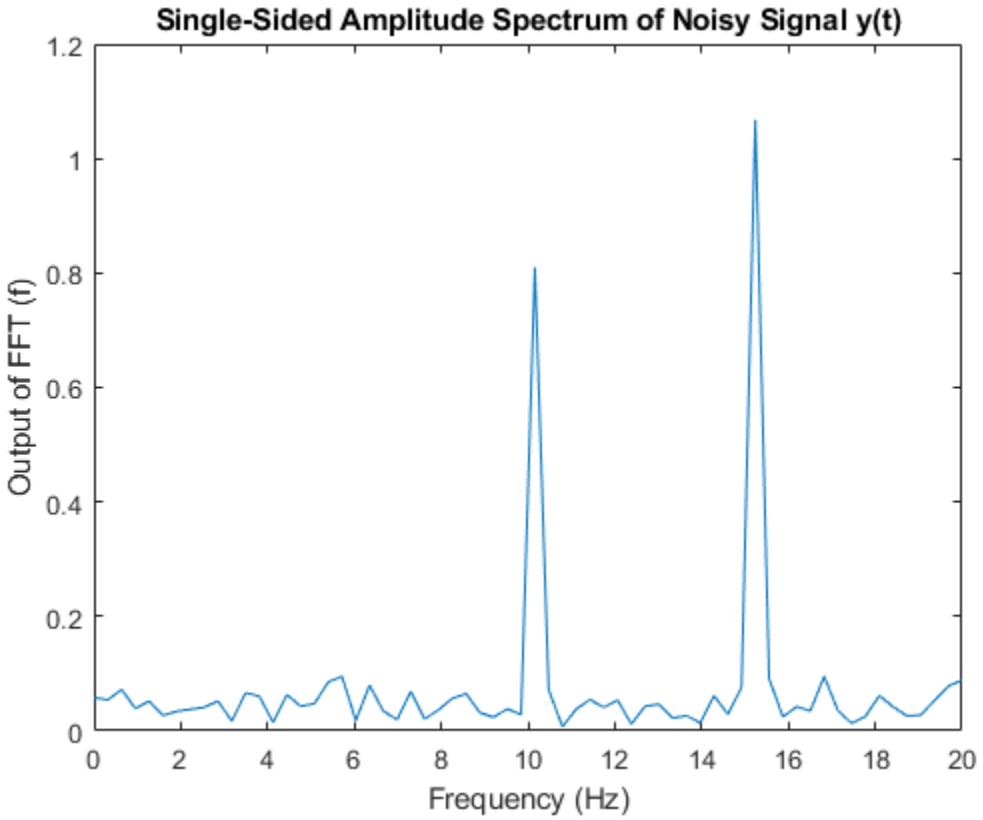
```
N = 128;
Fs = 40;
t = (0:N-1)'/Fs;
x = sin(2*pi*15*t) + 0.75*cos(2*pi*10*t);
y = x + .25*randn(size(x));
y_fixed = sfi(y,32,16);
noOp = zeros(1,'like',y_fixed);
```

Compute the FFT of the signal to use as the input to the IFFT object.

```
hdlfft = dsphdl.FFT('FFTLength',N,'BitReversedOutput',false);
Yf = zeros(1,4*N);
validOut = false(1,4*N);
for loop = 1:1:N
         [Yf(loop),validOut(loop)] = hdlfft(complex(y_fixed(loop)),true);
end
for loop = N+1:1:4*N
      [Yf(loop),validOut(loop)] = hdlfft(complex(noOp),false);
end
Yf = Yf(validOut == 1);
```

Plot the single-sided amplitude spectrum.

```
plot(Fs/2*linspace(0,1,N/2),2*abs(Yf(1:N/2)/N))
title('Single-Sided Amplitude Spectrum of Noisy Signal y(t)')
xlabel('Frequency (Hz)')
ylabel('Output of FFT (f)')
```



Select frequencies that hold the majority of the energy in the signal. The `cumsum` function does not accept fixed-point arguments, so convert the data back to `double`.

```
[Ysort,i] = sort(abs(double(transpose(Yf(1:N)))),1,'descend');
Ysort_d = double(Ysort);
CumEnergy = sqrt(cumsum(Ysort_d.^2))/norm(Ysort_d);
j = find(CumEnergy > 0.9, 1);
     disp(['Number of FFT coefficients that represent 90% of the ', ...
     'total energy in the sequence: ', num2str(j)])
Yin = zeros(N,1);
Yin(i(1:j)) = Yf(i(1:j));
```

```
Number of FFT coefficients that represent 90% of the total energy in the sequence: 4
```

Write a function that creates and calls the IFFT System object™. You can generate HDL from this function.

```
function [yOut,validOut] = HDLIFFT128(yIn,validIn)
%HDLIFFT128
% Processes one sample of data using the dsphdl.IFFT System object(TM)
% yIn is a fixed-point scalar or column vector.
% validIn is a logical scalar.
% You can generate HDL code from this function.

  persistent ifft128;
  if isempty(ifft128)
    ifft128 = dsphdl.IFFT('FFTLength',128);
```

```
  end
  [yOut,validOut] = ifft128(yIn,validIn);
end
```

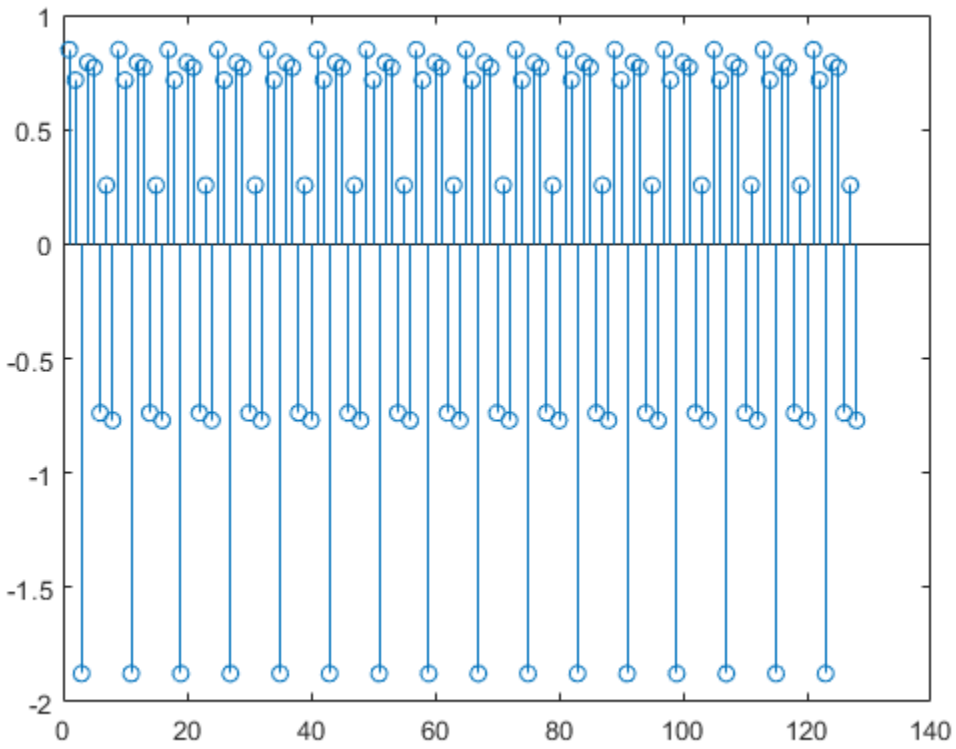
Compute the IFFT by calling the function for each data sample.

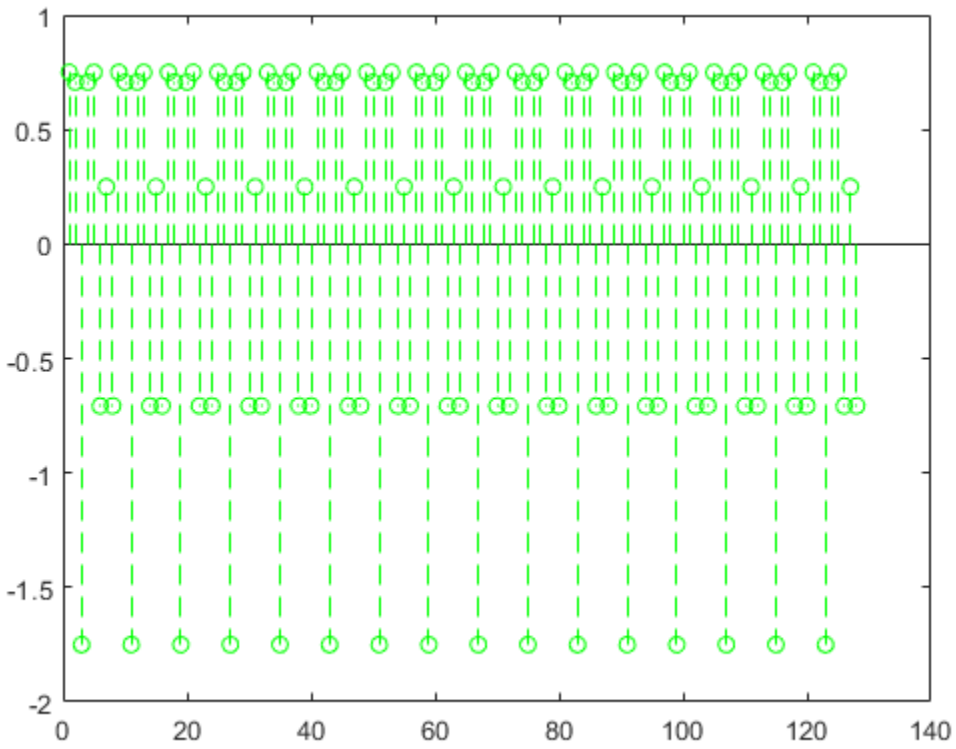
```
Xt = zeros(1,3*N);
validOut = false(1,3*N);
for loop = 1:1:N
    [Xt(loop),validOut(loop)] = HDLIFFT128(complex(Yin(loop)),true);
end
for loop = N+1:1:3*N
    [Xt(loop),validOut(loop)] = HDLIFFT128(complex(0),false);
end
```

Discard invalid output samples. Then inspect the output and compare it with the input signal. The original input is in green.

```
Xt = Xt(validOut==1);
Xt = bitrevorder(Xt);
norm(x-transpose(Xt(1:N)))
figure
stem(real(Xt))
figure
stem(real(x),'--g')
```

```
ans =

    0.7863
```


1
0.5
0
-0.5
-1
-1.5
-2
0
20
40
60
80
100
120
140



Create a Vector-Input IFFT for HDL Code Generation

Create the specifications and input signal. This example uses a 128-point FFT and computes the transform over 16 samples at a time.

```
N = 128;
V = 16;
Fs = 40;
t = (0:N-1)'/Fs;
x = sin(2*pi*15*t) + 0.75*cos(2*pi*10*t);
y = x + .25*randn(size(x));
y_fixed = sfi(y,32,24);
y_vect = reshape(y_fixed,V,N/V);
```

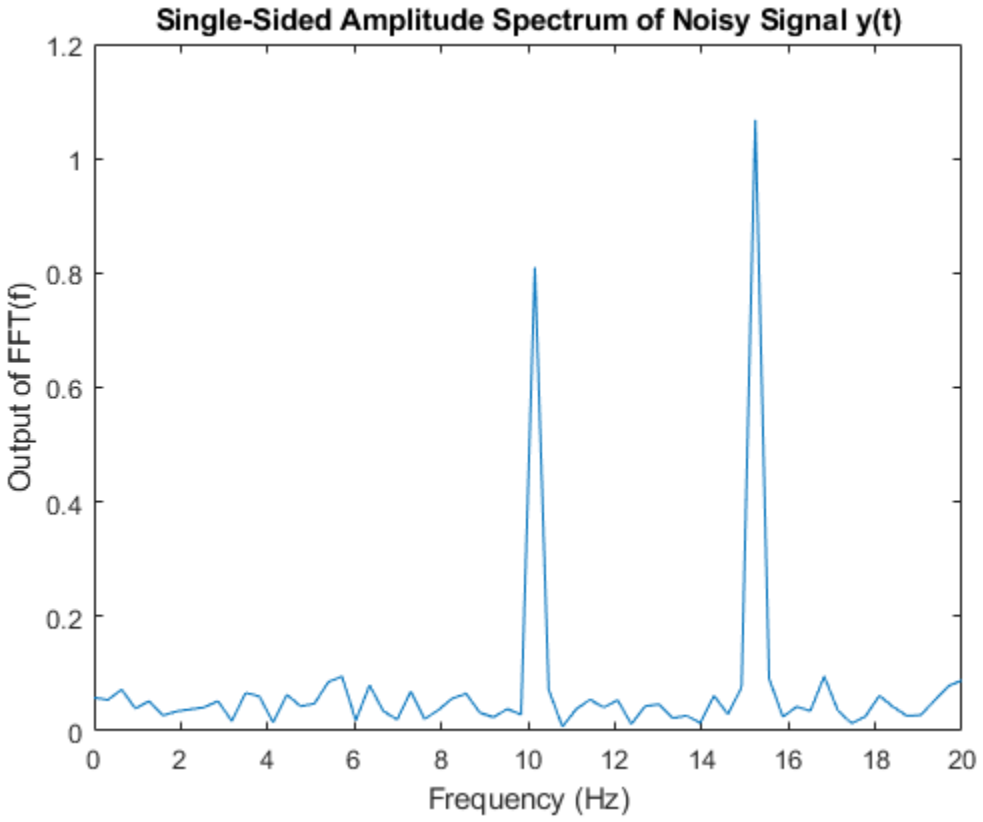
Compute the FFT of the signal, to use as the input to the IFFT object.

```
hdlfft = dsphdl.FFT('FFTLength',N);
loopCount = getLatency(hdlfft,N,V)+N/V;
Yf = zeros(V,loopCount);
validOut = false(V,loopCount);
for loop = 1:1:loopCount
    if ( mod(loop,N/V) == 0 )
        i = N/V;
    else
        i = mod(loop,N/V);
    end
```

```
        [Yf(:,loop),validOut(loop)] = hdlfft(complex(y_vect(:,i)),(loop<=N/V));
end
```

Plot the single-sided amplitude spectrum.

```
C = Yf(:,validOut==1);
Yf_flat = C(:);
Yr = bitrevorder(Yf_flat);
plot(Fs/2*linspace(0,1,N/2),2*abs(Yr(1:N/2)/N))
title('Single-Sided Amplitude Spectrum of Noisy Signal y(t)')
xlabel('Frequency (Hz)')
ylabel('Output of FFT(f)')
```



Select frequencies that hold the majority of the energy in the signal. The `cumsum` function doesn't accept fixed-point arguments, so convert the data back to `double`.

```
[Ysort,i] = sort(abs(double(Yr(1:N))),1,'descend');
CumEnergy = sqrt(cumsum(Ysort.^2))/norm(Ysort);
j = find(CumEnergy > 0.9, 1);
     disp(['Number of FFT coefficients that represent 90% of the ', ...
     'total energy in the sequence: ', num2str(j)])
Yin = zeros(N,1);
Yin(i(1:j)) = Yr(i(1:j));
YinVect = reshape(Yin,V,N/V);
```

```
Number of FFT coefficients that represent 90% of the total energy in the sequence: 4
```

Write a function that creates and calls the IFFT System object™. You can generate HDL from this function.

```
function [yOut,validOut] = HDLIFFT128V16(yIn,validIn)
%HDLFFT128V16
% Processes 16-sample vectors of FFT data
% yIn is a fixed-point column vector.
% validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent ifft128v16;
  if isempty(ifft128v16)
    ifft128v16 = dsphdl.IFFT('FFTLength',128)
  end
  [yOut,validOut] = ifft128v16(yIn,validIn);
end
```

Compute the IFFT by calling the function for each data sample.

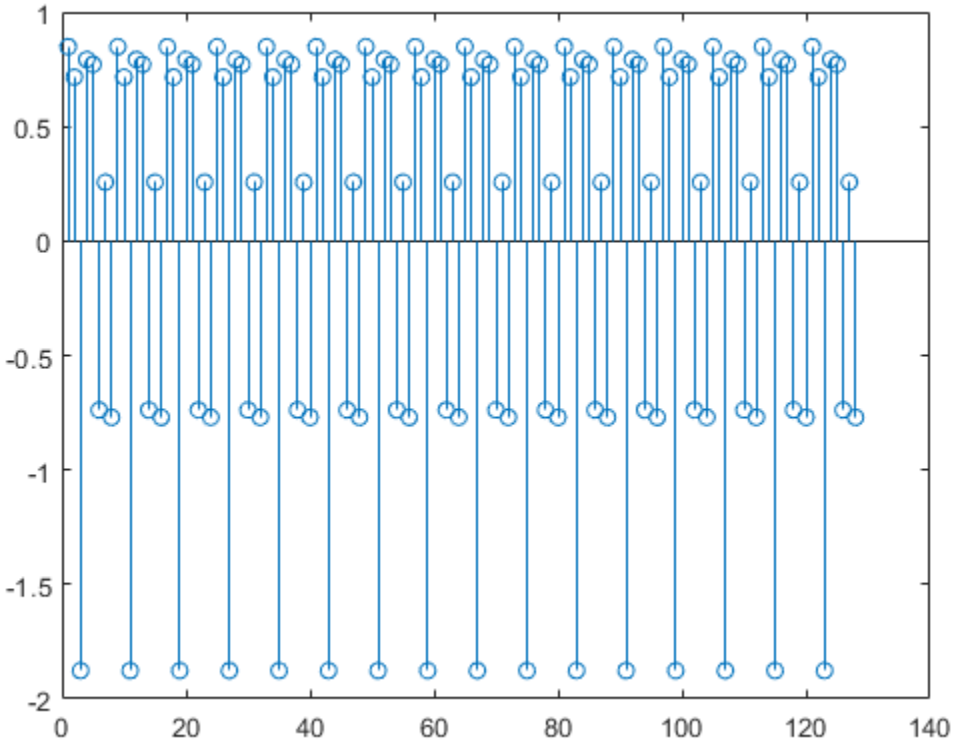
```
Xt = zeros(V,loopCount);
validOut = false(V,loopCount);
for loop = 1:1:loopCount
    if ( mod(loop,N/V) == 0 )
        i = N/V;
    else
        i = mod(loop,N/V);
    end
       [Xt(:,loop),validOut(loop)] = HDLIFFT128V16(complex(YinVect(:,i)),(loop<=N/V));
end
```

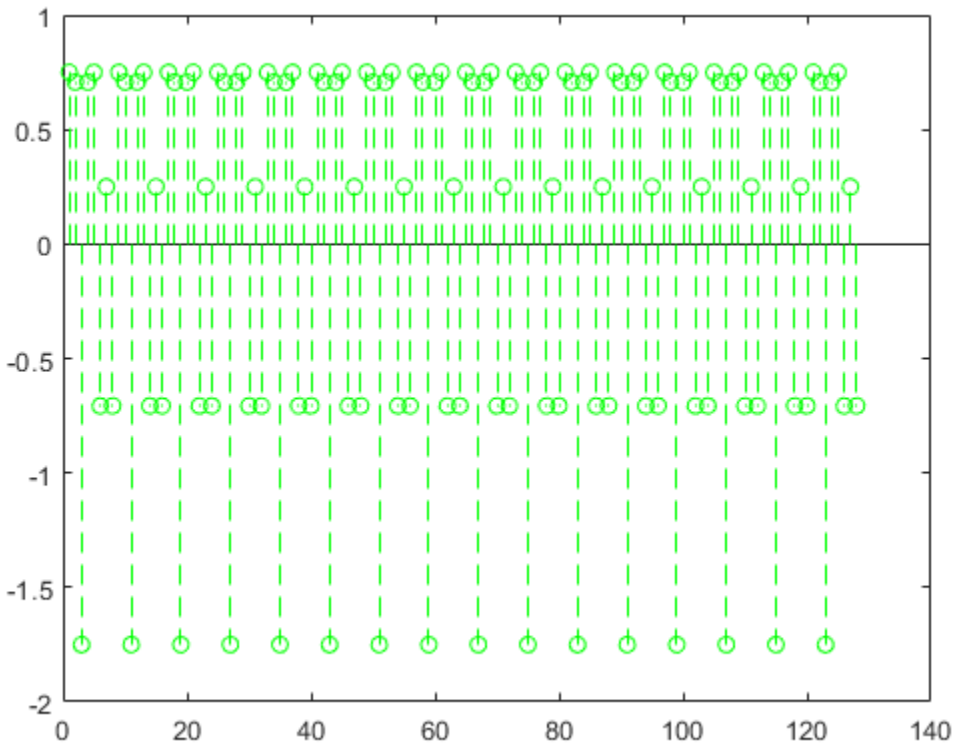
```
ifft128v16 =

  dsphdl.IFFT with properties:

               FFTLength: 128
            Architecture: 'Streaming Radix 2^2'
    ComplexMultiplication: 'Use 4 multipliers and 2 adders'
        BitReversedOutput: true
         BitReversedInput: false
                Normalize: true

  Use get to show all properties
```

Discard invalid output samples. Then inspect the output and compare it with the input signal. The original input is in green.

```
C = Xt(:,validOut==1);
Xt = C(:);
Xt = bitrevorder(Xt);
norm(x-Xt(1:N))
figure
stem(real(Xt))
figure
stem(real(x),'--g')
```

```
ans =

    0.7863
```





Explore Latency of HDL IFFT Object

The latency of the object varies with the FFT length and the vector size. Use the `getLatency` function to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous.

Create a new `dsphdl.IFFT` object and request the latency.

```
hdlifft = dsphdl.IFFT('FFTLength',512);
L512 = getLatency(hdlifft)
```

```
L512 = 599
```

Request hypothetical latency information about a similar object with a different FFT length. The properties of the original object do not change. When you do not specify a vector length, the function assumes scalar input data.

```
L256 = getLatency(hdlifft,256)
```

```
L256 = 329
```

```
N = hdlifft.FFTLength
```

```
N = 512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(hdlifft,256,8)
```

```
L256v8 = 93
```

Enable scaling at each stage of the IFFT. The latency does not change.

```
hdlifft.Normalize = true;
L512n = getLatency(hdlifft)
```

```
L512n = 599
```

Request the same output order as the input order. This setting increases the latency because the object must collect the output before reordering.

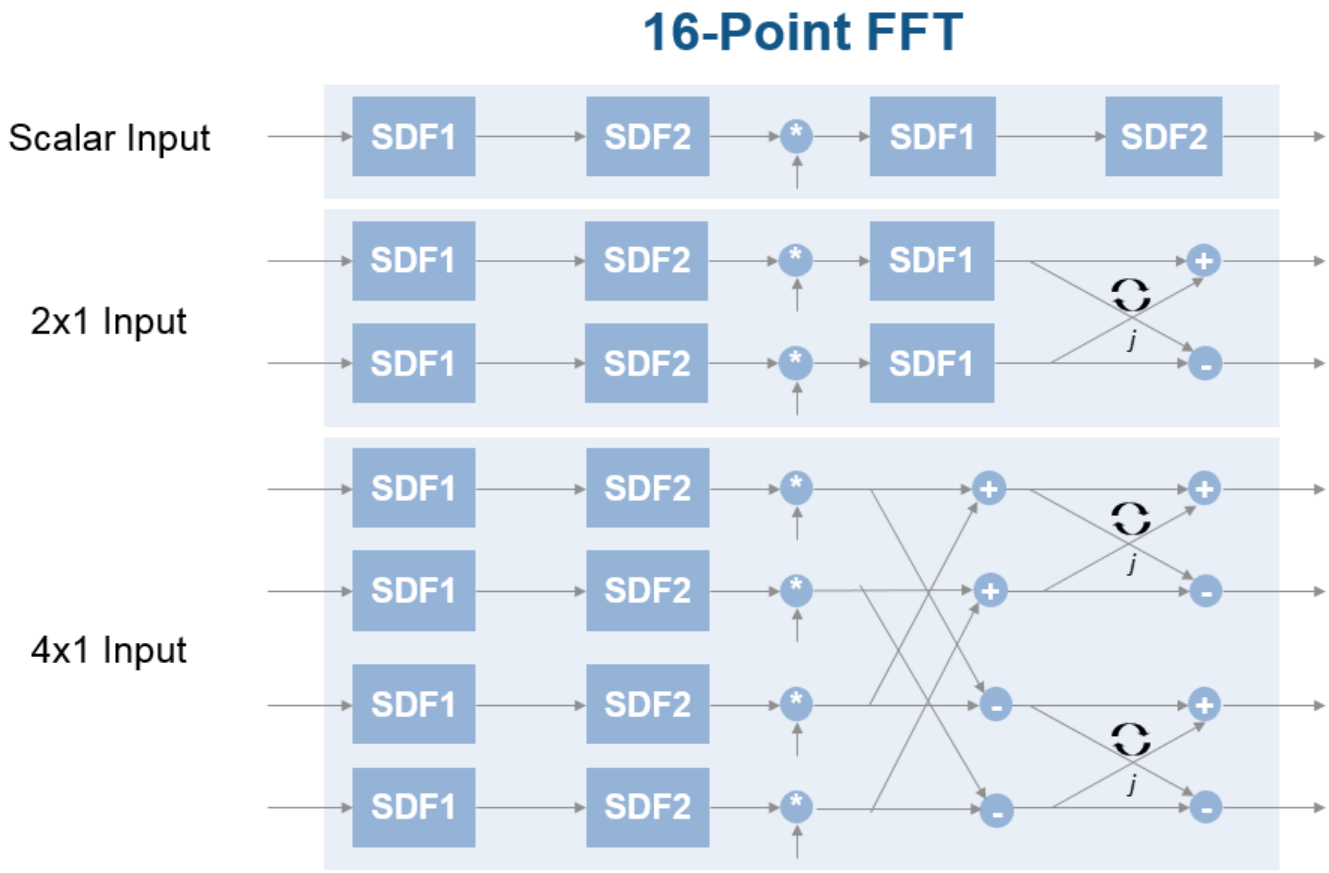
```
hdlifft.BitReversedOutput = false;
L512r = getLatency(hdlifft)
```

```
L512r = 1078
```

Algorithms

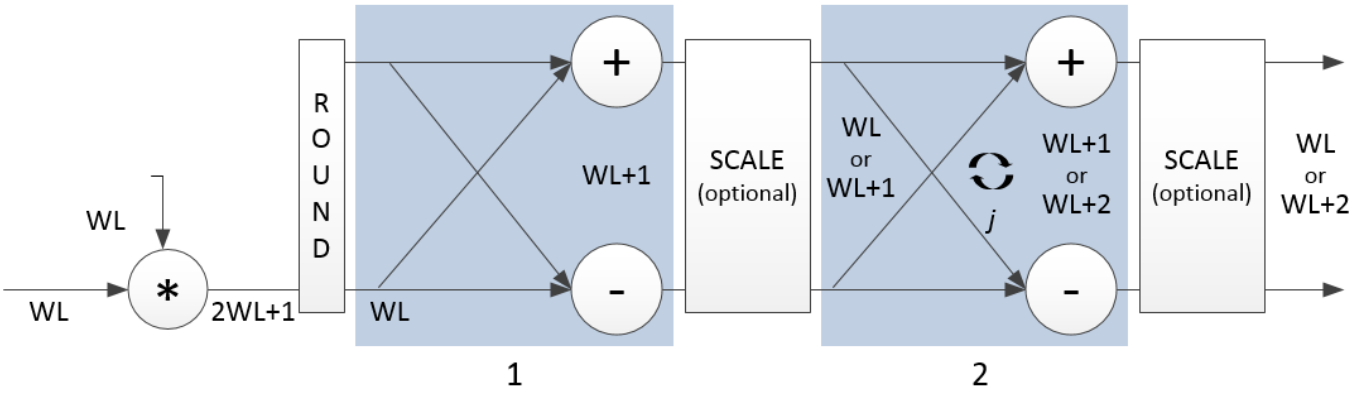
Streaming Radix 2^2

The streaming Radix 2^2 architecture implements a low-latency architecture. It saves resources compared to a streaming Radix 2 implementation by factoring and grouping the FFT equation. The architecture has $\log_4(N)$ stages. Each stage contains two single-path delay feedback (SDF) butterflies with memory controllers. When you use vector input, each stage operates on fewer input samples, so some stages reduce to a simple butterfly, without SDF.



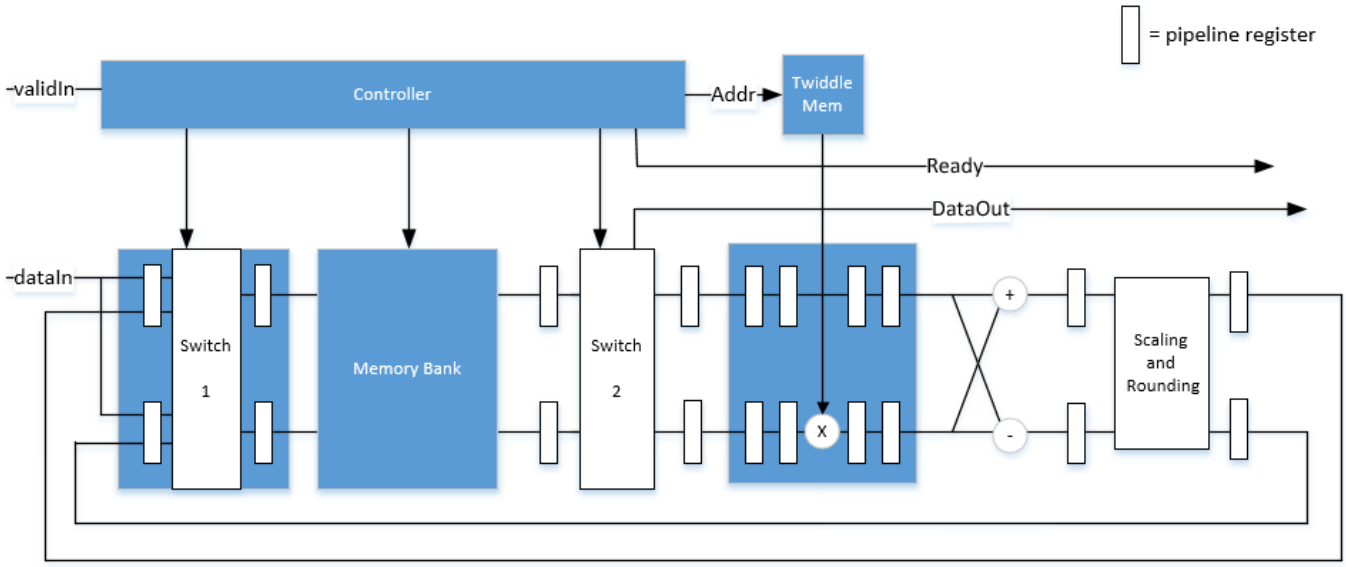
The first SDF stage is a regular butterfly. The second stage multiplies the outputs of the first stage by –*j*. To avoid a hardware multiplier, the block swaps the real and imaginary parts of the inputs, and again swaps the imaginary parts of the resulting outputs. Each stage rounds the result of the twiddle factor multiplication to the input word length. The twiddle factors have two integer bits, and the rest of the bits are used for fractional bits. The twiddle factors have the same bit width as the input data, *WL*. The twiddle factors have two integer bits, and *WL*-2 fractional bits.

If you enable scaling, the algorithm divides the result of each butterfly stage by 2. Scaling at each stage avoids overflow, keeps the word length the same as the input, and results in an overall scale factor of 1/*N*. If scaling is disabled, the algorithm avoids overflow by increasing the word length by 1 bit at each stage. The diagram shows the butterflies and internal word lengths of each stage, not including the memory.



Burst Radix 2

The burst Radix 2 architecture implements the FFT by using a single complex butterfly multiplier. The algorithm cannot start until it has stored the entire input frame, and it cannot accept the next frame until computations are complete. The output **ready** port indicates when the algorithm is ready for new data. The diagram shows the burst architecture, with pipeline registers.



When you use this architecture, your input data must comply with the **ready** backpressure signal.

Control Signals

The algorithm processes input data only when the input **valid** port is 1. Output data is valid only when the output **valid** port is 1.

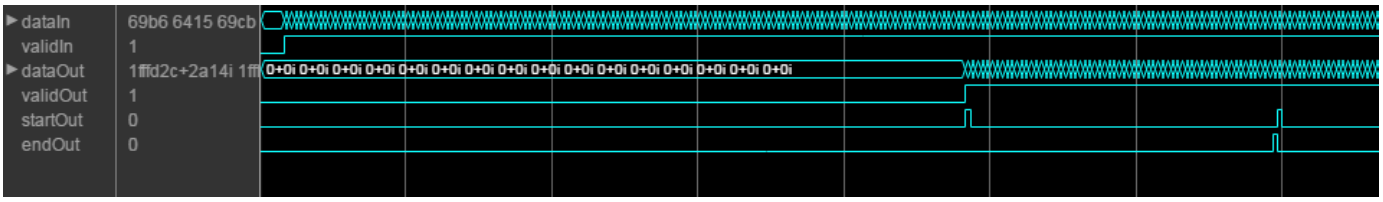
When the optional input **reset** port is 1, the algorithm stops the current calculation and clears all internal states. The algorithm begins new calculations when **reset** port is 0 and the input **valid** port starts a new frame.

Timing Diagram

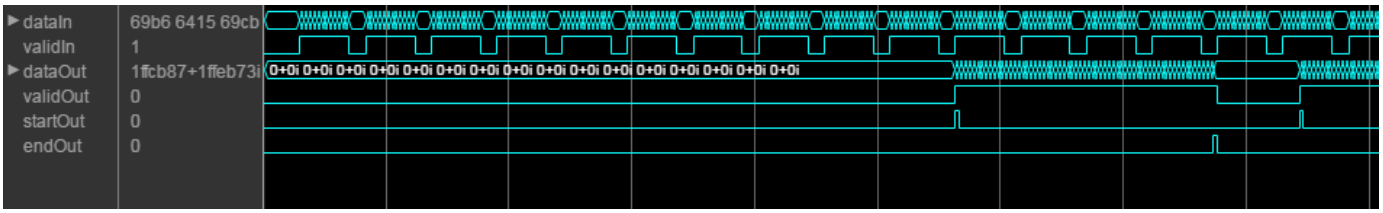
This diagram shows the input and output **valid** port values for contiguous scalar input data, streaming Radix 2^2 architecture, an FFT length of 1024, and a vector size of 16.

The diagram also shows the optional **start** and **end** port values that indicate frame boundaries. If you enable the **start** port, the **start** port value pulses for one cycle with the first valid output of the frame. If you enable the **end** port, the **start** port value pulses for one cycle with the last valid output of the frame.

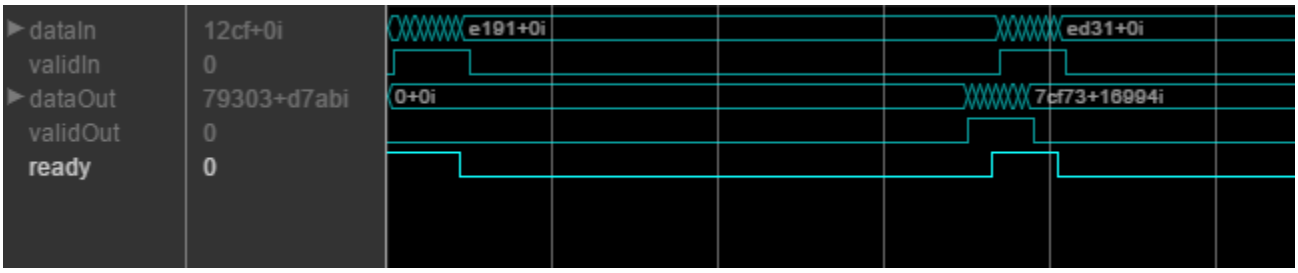
If you apply continuous input frames, the output will also be continuous after the initial latency.



The input **valid** port can be noncontiguous. Data accompanied by an input **valid** port is processed as it arrives, and the resulting data is stored until a frame is filled. Then the algorithm returns contiguous output samples in a frame of *N* (**FFT length**) cycles. This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16.



When you use the burst architecture, 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. You must apply input **data** and **valid** signals only when **ready** is `1` (true). The algorithm ignores any input **data** and **valid** signals when **ready** is `0` (false).

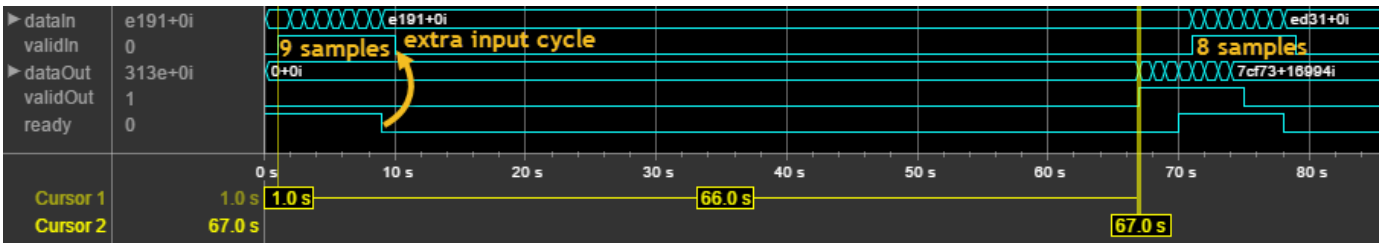


Latency

The latency varies with the FFT length and the vector size. Use the `getLatency` function to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous.

When using the burst architecture with a contiguous input, if your design waits for **ready** to output `0` before de-asserting the input **valid**, then one extra cycle of data arrives at the input. This data sample

is the first sample of the next frame. The algorithm can save one sample while processing the current frame. Due to this one sample advance, the observed latency of the later frames (from input **valid** to output **valid**) is one cycle shorter than the reported latency. The latency is measured from the first cycle, when input **valid** is 1 to the first cycle when output **valid** is 1. The number of cycles between when **ready** port is 0 and the output **valid** port is 1 is always *latency* – *FFTLength*.



Performance

This resource and performance data is the synthesis result from the generated HDL targeted to a Xilinx Virtex-6 (XC6VLX75T-1FF484) FPGA. The examples in the tables have this configuration:

- 1024 FFT length (default)
- Complex multiplication using 4 multipliers, 2 adders
- Output scaling enabled
- Natural order input, Bit-reversed output
- 16-bit complex input data
- Clock enables minimized (HDL Coder parameter)

Performance of the synthesized HDL code varies with your target and synthesis options. For instance, reordering for a natural-order output uses more RAM than the default bit-reversed output, and real input uses less RAM than complex input.

For a scalar input Radix 2^2 configuration, the design achieves 326 MHz clock frequency. The latency is 1116 cycles. The design uses these resources.

Resource	Number Used
LUT	4597
FFS	5353
Xilinx LogiCORE DSP48	12
Block RAM (16K)	6

When you vectorize the same Radix 2^2 implementation to process two 16-bit input samples in parallel, the design achieves 316 MHz clock frequency. The latency is 600 cycles. The design uses these resources.

Resource	Number Used
LUT	7653
FFS	9322
Xilinx LogiCORE DSP48	24

Resource	Number Used
Block RAM (16K)	8

The block supports scalar input data only when implementing burst Radix 2 architecture. The burst design achieves 309 MHz clock frequency. The latency is 5811 cycles. The design uses these resources.

Resource	Number Used
LUT	971
FFS	1254
Xilinx LogiCORE DSP48	3
Block RAM (16K)	6

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this System object was named `dsp.HDLIFFT` and was part of the DSP System Toolbox product.

FFT length of 4
Behavior changed in R2022a

This System object now supports an FFT length of 4. In previous releases the FFT length had to be a power of 2 from 2^3 to 2^{16}.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Objects
`dsphdl.FFT` | `dsp.IFFT`

Blocks
IFFT | FFT

Introduced in R2014b

dsphdl.Channelizer

Package: `dsphdl`

Polyphase filter bank and fast Fourier transform

Description

The `dsphdl.Channelizer` System object separates a broadband input signal into multiple narrowband output signals. It provides hardware speed and area optimization for streaming data applications. The object accepts scalar or vector input of real or complex data, provides hardware-friendly control signals, and has optional output frame control signals. You can achieve gigasamples-per-second (GSPS) throughput by using vector input. The object implements a polyphase filter, with one subfilter per input vector element. The hardware implementation interleaves the subfilters, which results in sharing each filter multiplier (*FFT Length* / *Input Size*) times. The object implements the same pipelined Radix 2^2 FFT algorithm as the `dsphdl.FFT` System object.

To channelize input data:

1. Create the `dsphdl.Channelizer` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
channelizer = dsphdl.Channelizer
channelizer = dsphdl.Channelizer(Name,Value)
```

Description

`channelizer = dsphdl.Channelizer` returns a System object, `channelizer`, that implements a raised-cosine filter and an 8-point FFT.

`channelizer = dsphdl.Channelizer(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`NumFrequencyBands` — FFT length
8 (default) | integer power of two

FFT length, specified as an integer power of two. For HDL code generation, the FFT length must be between 2^2 and 2^{16}, inclusive.

`FilterCoefficients` — Polyphase filter coefficients
`[-0.032 0.121 0.318 0.482 0.546 0.482 0.318 0.121 -0.032]` (default) | vector of real or complex numeric values

Polyphase filter coefficients, specified as a vector of numeric values. If the number of coefficients is not a multiple of `NumFrequencyBands`, the object pads this vector with zeros. The default filter specification is a raised-cosine FIR filter, `rcosdesign(0.25,2,4,'sqrt')`. You can specify a vector of coefficients or a call to a filter design function that returns the coefficient values. By default, the object casts the coefficients to the same data type as the input.

`FilterStructure` — Filter design properties or coefficients
`'Direct form transposed'` (default) | `'Direct form systolic'`

Specify the HDL filter architecture as one of these structures:

- `Direct form transposed` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. For architecture and performance details, see “Fully Parallel Transposed Architecture”.
- `Direct form systolic` — This architecture provides a fully parallel filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For architecture and performance details, see “Fully Parallel Systolic Architecture”.

All implementations share multipliers for symmetric and antisymmetric coefficients and remove multipliers for zero-valued coefficients.

`ComplexMultiplication` — HDL implementation of complex multipliers
`'Use 4 multipliers and 2 adders'` (default) | `'Use 3 multipliers and 5 adders'`

HDL implementation of complex multipliers, specified as either `'Use 4 multipliers and 2 adders'` or `'Use 3 multipliers and 5 adders'`. Depending on your synthesis tool and target device, one option may be faster or smaller.

`OutputSize` — Size of output data
`'Same as number of frequency bands'` (default) | `'Same as input size'`

Size of output data, specified as:

- `'Same as number of frequency bands'` — Output data is a 1-by-*M* vector, where *M* is the FFT length.
- `'Same as input size'` — Output data is an *M*-by-1 vector, where *M* is the input vector size.

The output order is bit natural for both output sizes.

`Normalize` — FFT scaling
`true` (default) | `false`

FFT output scaling, specified as either:

- `true` — The FFT implements an overall 1/*N* scale factor by scaling the result of each pipeline stage by 2. This adjustment keeps the output of the FFT in the same amplitude range as its input.
- `false` — The FFT avoids overflow by increasing the word length by one bit at each stage.

`RoundingMethod` — Rounding mode used for internal fixed-point calculations
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding mode used for fixed-point operations. The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is `single` or `double`. Each FFT stage rounds after the twiddle factor multiplication but before the butterflies. Rounding can also occur when casting the coefficients and the output of the polyphase filter to the data types you specify.

`OverflowAction` — Overflow handling for internal fixed-point calculations
`'Wrap'` (default) | `'Saturate'`

“Overflow Handling” used for fixed-point operations. The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input is `single` or `double`. This option applies to casting the coefficients and the output of the polyphase filter to the data types you specify.

The FFT algorithm avoids overflow by either scaling the output of each stage (`Normalize` enabled), or by increasing the word length by 1 bit at each stage (`Normalize` disabled).

`CoefficientsDataType` — Data type of filter coefficients
`'Same word length as input'` (default) | `numerictype` object

The object casts the polyphase filter coefficients to this data type, using the rounding and overflow settings you specify. When you specify `'Same word length as input'` (default), the object selects the binary point using `fi()` best-precision rules.

`FilterOutputDataType` — Data type of output of polyphase filter
`'Same word length as input'` (default) | `'Full precision'` | `numerictype` object

Data type of the output of the polyphase filter, specified as `'Same word length as input'`, `'Full precision'`, or a `numerictype` object. The object casts the output of the polyphase filter (the input to the FFT) to this data type, using the rounding and overflow settings you specify. When you specify `'Full precision'`, the object selects a best-precision binary point by considering the values of your filter coefficients and the range of your input data type.

By default, the FFT logic does not change the data type. When you disable `Normalize`, the FFT algorithm avoids overflow by increasing the word length by 1 bit at each stage.

`ResetInputPort` — Enable reset argument
`false` (default) | `true`

Enable `reset` input argument to the object. When `reset` is `1` (true), the object stops the current calculation and clears internal states. When the `reset` is `0` (false) and the input `valid` is `1` (true), the object captures data for processing.

`StartOutputPort` — Enable start output argument
`false` (default) | `true`

Enable `startOut` output argument of the object. When enabled, the object returns an additional output signal that is `1` (true) on the first cycle of each valid output frame.

`EndOutputPort` — Enable end output argument

`false` (default) | `true`

Enable `endOut` output argument of the object. When enabled, the object returns an additional output signal that is `1` (true) on the first cycle of each valid output frame.

Usage

Syntax

```
[dataOut,validOut] = channelizer(dataIn,validIn)
[dataOut,validOut] = channelizer(dataIn,validIn,reset)
[dataOut,startOut,endOut,validOut] = channelizer( ___ )
```

Description

`[dataOut,validOut] = channelizer(dataIn,validIn)` filters and computes a fast Fourier transform, and returns the frequency channels, `dataOut`, detected in the input signal, `dataIn`, when `validIn` is `1` (true). The `validIn` and `validOut` arguments are logical scalars that indicate the validity of the input and output signals, respectively.

`[dataOut,validOut] = channelizer(dataIn,validIn,reset)` returns the frequency channels, `dataOut`, detected in the input signal, `dataIn`, when `validIn` is `1` (true) and `reset` is `0` (false). When `reset` is `1` (true), the object stops the current calculation and clears all internal state.

To use this syntax, set the `ResetInputPort` property to `true`. For example:

```
channelizer = dsphdl.Channelizer(...,'ResetInputPort',true);
...
[dataOut,validOut] = channelizer(dataIn,validIn,reset)
```

`[dataOut,startOut,endOut,validOut] = channelizer( ___ )` returns the frequency channels, `dataOut`, computed from the input arguments of any of the previous syntaxes. `startOut` is `1` (true) for the first sample of a frame of output data. `endOut` is `1` (true) for the last sample of a frame of output data.

To use this syntax, set the `StartOutputPort` and `EndOutputPort` properties to `true`. For example:

```
channelizer = dsphdl.Channelizer(...,'StartOutputPort',true,'EndOutputPort',true);
...
[dataOut,startOut,endOut,validOut] = channelizer(dataIn,validIn)
```

Input Arguments

`dataIn` — Input data

scalar or column vector of real or complex values

Input data, specified as a scalar or column vector of real or complex values.

The vector size must be a power of 2 and in the range [2, 64], and is not greater than the number of channels (FFT length).

`double` and `single` data types are supported for simulation, but not for HDL code generation.

The object does not accept `uint64` data.

Data Types: `fi` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `single` | `double`

`validIn` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

`reset` — Clears internal states

logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set `ResetInputPort` to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — Frequency channel output data

row vector

Frequency channel output data, returned as a row vector.

- If you set `OutputSize` to `'Same as number of frequency bands'` (default), the output data is a 1-by-*M* vector, where *M* is the FFT length.
- If you set `OutputSize` to `'Same as input size'`, the output data is an *M*-by-1 vector, where *M* is the input vector size.

The output order is bit natural for either output size. The data type is a result of the `FilterOutputDataType` and the FFT bit growth necessary to avoid overflow.

`validOut` — Indicates valid output data

scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

`startOut` — First valid cycle of output data

`logical` scalar

First sample of output frame, returned as a logical scalar. The object sets `startOut` to `1` (true) during the first valid sample on `dataOut`.

Dependencies

To enable this argument, set `StartOutputPort` to `true`.

Data Types: `logical`

`endOut` — Last valid cycle of output data

`logical` scalar

Last sample of output frame, returned as a logical scalar. The object sets `endOut` to `1` (true) during the last valid sample on `dataOut`.

Dependencies

To enable this argument, set `EndOutputPort` to `true`.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to dsphdl.Channelizer

getLatency	Latency of channelizer calculation

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create Channelizer for HDL Generation

Create a function that contains a channelizer object and supports HDL code generation.

Create the specifications and input signal. The signal has 8 frequency channels.

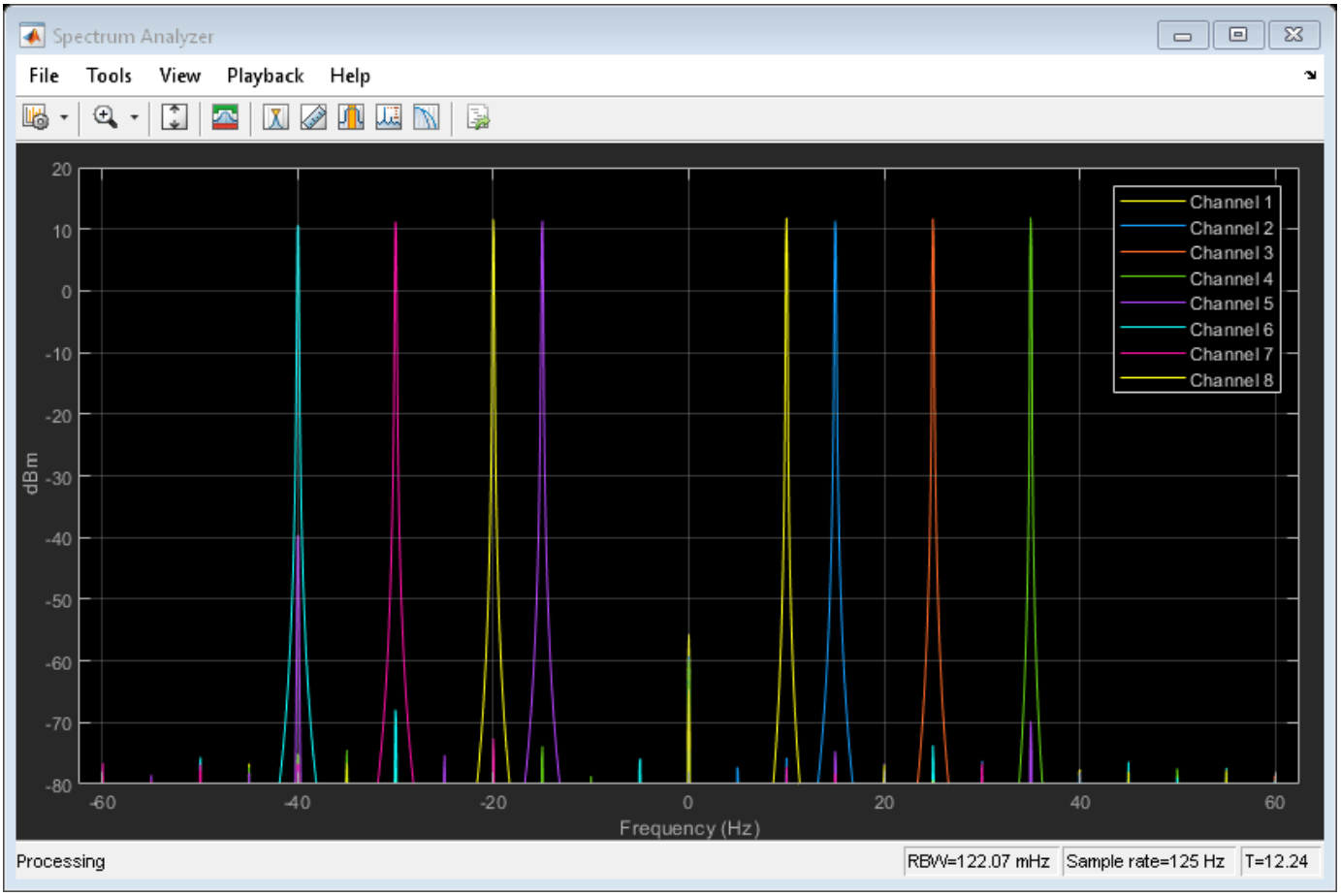
```
N = 8;
loopCount = 1024;
offsets = [-40 -30 -20 10 15 25 35 -15];
sinewave = dsp.SineWave('ComplexOutput',true,'Frequency', ...
     offsets+(-375:125:500),'SamplesPerFrame',loopCount);
spectrumAnalyzer = dsp.SpectrumAnalyzer('ShowLegend',true, ...
     'SampleRate',sinewave.SampleRate/N);
```

Write a function that creates and calls the channelizer System object™. You can generate HDL from this function.

```
function [yOut,validOut] = HDLChannelizer8(yIn,validIn)
%HDLChannelizer8
% Process one sample of data using the dsphdl.Channelizer System object
% yIn is a fixed-point scalar or column vector.
% validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent channelize8;
  coder.extrinsic('tf');
  coder.extrinsic('dsp.Channelizer');

  if isempty(channelize8)
    % Use filter coeffs from non-HDL channelizer, or supply your own.
    channelizer = coder.const(dsp.Channelizer('NumFrequencyBands',8));
    coeff = coder.const(tf(channelizer));
    channelize8 = dsphdl.Channelizer('NumFrequencyBands',8,'FilterCoefficients',coeff);
  end
  [yOut,validOut] = channelize8(yIn,validIn);
end
```

Channelize the input data by calling the object for each data sample.

```
y = zeros(loopCount/N,N);
validOut = false(loopCount/N,1);
yValid = zeros(loopCount/(N*N),N);
for reps=1:20
   x = fi(sum(sinewave(),2),1,18);
   for loop=1:length(x)
      [y(loop,:),validOut(loop)]= HDLChannelizer8(x(loop),true);
   end
   yValid = y(validOut == 1,:);
   spectrumAnalyzer(yValid);
end
```



Explore Latency of Channelizer Object

The latency of the `dsphdl.Channelizer` object varies with the FFT length, filter structure, vector size, and input type. Use the `getLatency` function to find the latency of a particular configuration. The latency is measured as the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous. The number of filter coefficients does not affect the latency. Setting the output size equal to the input size reduces the latency because the samples are not saved and reordered.

Create a `dsphdl.Channelizer` object with filter structure set to direct form transposed and request the latency.

```
channelize = dsphdl.Channelizer('NumFrequencyBands',512, 'FilterStructure','Direct form transpose
L512 = getLatency(channelize)
```

```
L512 = 1118
```

Request hypothetical latency information about a similar object with a different number of frequency bands (FFT length). The properties of the original object do not change.

```
L256 = getLatency(channelize,256)
```

```
L256 = 592
```

```
N = channelize.NumFrequencyBands
```

```
N = 512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(channelize,256,8)
```

```
L256v8 = 132
```

Enable scaling at each stage of the FFT. The latency does not change.

```
channelize.Normalize = true;
L512n = getLatency(channelize)
```

```
L512n = 1118
```

Request the same output size and order as the input data. The latency decreases because the object does not need to store and reorder the data before output. The default input size is scalar.

```
channelize.OutputSize = 'Same as input size';
L512r = getLatency(channelize)
```

```
L512r = 1084
```

Check the latency of a vector input implementation where the input and output are the same size. Specify the current value of the FFT length and a vector size of 8 samples. The latency decreases because the object computes results in parallel when the input is a vector.

```
L512rv8 = getLatency(channelize,channelize.NumFrequencyBands,8)
```

```
L512rv8 = 218
```

Check the latency of a vector input implementation where the input type is complex. Specify the current value of the FFT length and a vector size of 16 samples.

```
L512rv16i = getLatency(channelize,channelize.NumFrequencyBands,16,true)
```

```
L512rv16i = 152
```

Algorithms

This object implements the algorithm described on the Channelizer block reference page.

Note The output of the `dsphdl.Channelizer` object does not match the output from the `dsp.Channelizer` object sample-for-sample. This mismatch is because the objects apply the input samples to the subfilters in different orders. The `dsphdl.Channelizer` object applies input X(0) to subfilter $E_{M-1}(z)$, X(1) to subfilter $E_{M-2}(z)$, ..., X(M-1) to subfilter $E_0(z)$. The channels detected by both objects match, when analyzed over multiple frames.

Latency

The latency varies with the FFT length, vector size, and filter structure. Use the `getLatency` function to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous. The filter

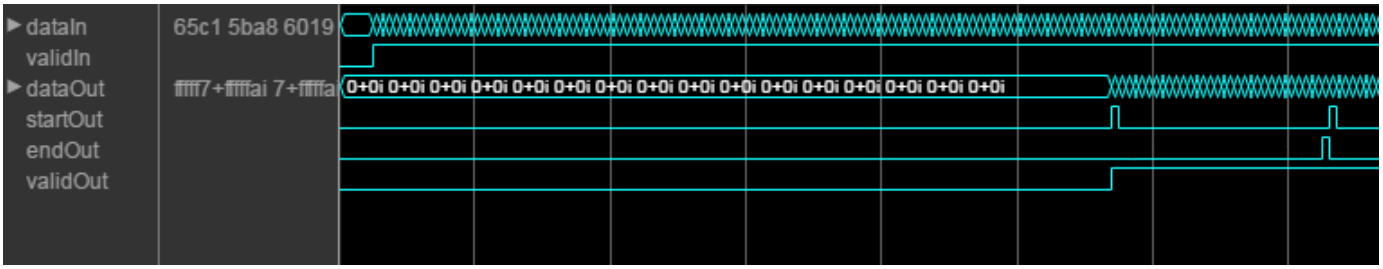
coefficients do not affect the latency. Setting the output size equal to the input size reduces the latency, because the samples are not saved and reordered.

Control Signals

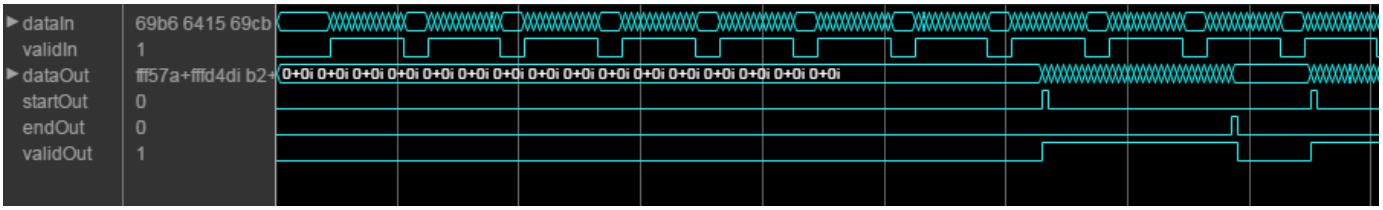
This diagram shows `validIn` and `validOut` signals for contiguous input data with a vector size of 16, an FFT length of 512, and when you select the `Direct form transposed` filter architecture. In this example, the output vector size is specified same as the input vector size.

The diagram also shows the optional `startOut` and `endOut` signals that indicate frame boundaries. When enabled, `startOut` pulses for one cycle with the first `validOut` of the frame, and `endOut` pulses for one cycle with the last `validOut` of the frame.

If you apply continuous input frames (no gap in `validIn` between frames), the output will also be continuous, after the initial latency.



The `validIn` signal can be noncontiguous. Data accompanied by a `validIn` signal is stored until a frame is filled. Then the data in output is a contiguous frame of *N*/*M* cycles. This diagram shows noncontiguous input and contiguous output for an FFT length of 512 and a vector size of 16 samples.



Performance

These resource and performance data are the place-and-route results from the generated HDL targeted to a Xilinx Zynq- 7000 ZC706 evaluation board. The three examples in the tables use this common configuration.

- FFT length (default) — 8
- Filter length — 96 coefficients
- Filter structure — Direct form transposed
- 16-bit complex input data
- Coefficient data type — Same word length as input
- Filter output data type — Same word length as input
- Complex multiplication — Use 4 multipliers and 2 adders
- Output scaling — Enabled

- Output vector size — Same as input size

Performance of the synthesized HDL code varies with your target and synthesis options.

For scalar input, the design achieves a clock frequency of 506.84 MHz. The latency is 51 cycles. The subfilters share each multiplier eight (*N*) times. The design uses these resources.

Resource	Number Used
LUT	2898
FFS	3746
Xilinx LogiCORE DSP48	28

For four-sample vector input, the design achieves a clock frequency of 452 MHz. The latency is 37 cycles. The subfilters share each multiplier twice (*N*/*M*). The design uses these resources.

Resource	Number Used
LUT	1991
FFS	8305
Xilinx LogiCORE DSP48	104

For eight-sample vector input, the design achieves a clock frequency of 360 MHz. The latency is 18 cycles. When the input size is the same as the FFT length, the subfilters do not share any multipliers. The design uses these resources.

Resource	Number Used
LUT	1683
FFS	2992
Xilinx LogiCORE DSP48	208

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this System object was named `dsp.HDLChannelizer` and was part of the DSP System Toolbox product.

The System object now supports fully parallel systolic architecture when you specify the `Filter structure` property to `Direct form systolic`.

FFT length of 4
Behavior changed in R2022a

This System object now supports an FFT length of 4. In previous releases the FFT length had to be a power of 2 from 2^3 to 2^{16}.

Direct form systolic filter structure support
Behavior changed in R2022a

This System object now supports direct form systolic filter structure.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Blocks

Channelizer | Channel Synthesizer | FFT

Objects

`dsphdl.ChannelSynthesizer` | `dsphdl.FFT`

Introduced in R2017a

dsphdl.FIRDecimator

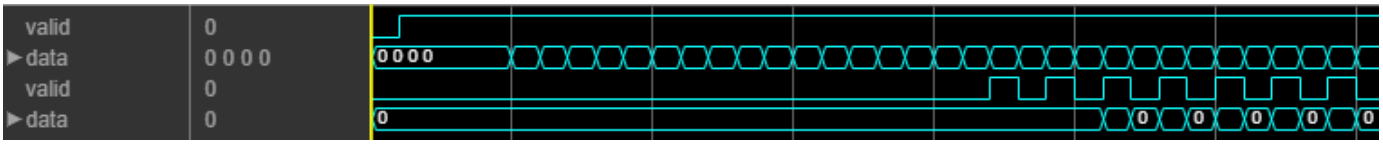
Package: `dsphdl`

Finite impulse response (FIR) decimation filter

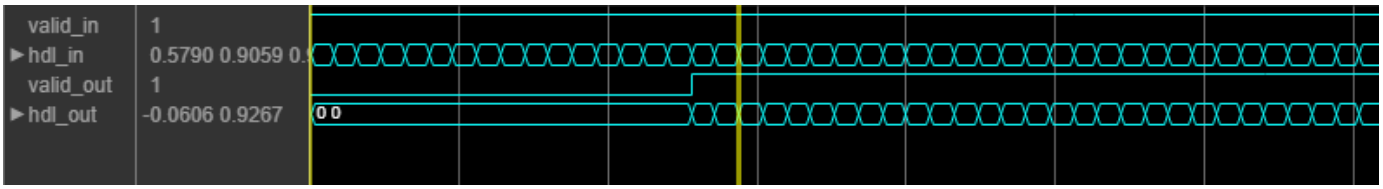
Description

The `dsphdl.FIRDecimator` System object implements a single-rate polyphase FIR decimation filter that is optimized for HDL code generation. The object provides a hardware-friendly interface with input and output control signals. To provide a cycle-accurate simulation of the generated HDL code, the object models architectural latency including pipeline registers and resource sharing.

The object accepts scalar or vector input. When you use vector input and the vector size is less than the decimation factor, the decimation factor must be an integer multiple of the vector size. In this case, the output is scalar and an output valid signal indicates which samples are valid after decimation. The output data is valid every *DecimationFactor*/*VectorSize* samples. The waveform shows an input vector of four samples and a decimation factor of eight. The output data is a scalar that is valid every second cycle.



When you use vector input and the vector size is greater than the decimation factor, the vector size must be an integer multiple of the decimation factor. In this case, the output is a vector of *VectorSize*/*DecimationFactor* samples. The waveform shows an input vector of eight samples and a decimation factor of four. The output data is a vector of two samples on every cycle.



The object provides two filter structures. The direct form systolic architecture provides an implementation that makes efficient use of Intel and Xilinx DSP blocks. This architecture can be fully parallel or serial. To use a serial architecture, the input samples must be spaced out with a regular number of invalid cycles between the valid samples. The direct form transposed architecture is a fully parallel implementation and is suitable for FPGA and ASIC applications. 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.

All filter structures optimize hardware resources by sharing multipliers for symmetric or antisymmetric filters and by removing the multipliers for zero-valued coefficients such as in half-band filters and Hilbert transforms.

The object implements one filter for each sample in the input vector. The object then shares this filter between the polyphase subfilters by interleaving the subfilter coefficients in time.

To filter and decimate input data with an HDL-optimized algorithm:

1. Create the `dsphdl.FIRDecimator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
firDecim = dsphdl.FIRDecimator
firDecim = dsphdl.FIRDecimator(dec,num)
firDecim = dsphdl.FIRDecimator( ___ ,Name,Value)
```

Description

`firDecim = dsphdl.FIRDecimator` creates a default HDL-optimized FIR decimation filter System object.

`firDecim = dsphdl.FIRDecimator(dec,num)` sets the `DecimationFactor` property to `dec` and the `Numerator` property to `num`.

`firDecim = dsphdl.FIRDecimator( ___ ,Name,Value)` sets properties by using one or more name-value pairs in addition to any input argument combination from previous syntaxes. Enclose each property name in quotes. For example, `'FilterStructure','Direct form transposed'` specifies the filter architecture as a fully parallel implementation that is suitable for FPGA and ASIC applications.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Main

`Numerator` — FIR filter coefficients

`fir1(35,0.4)` (default) | real- or complex-valued vector

FIR filter coefficients, specified as a real- or complex-valued vector. You can specify the vector as a workspace variable or as a call to a filter design function. When the input data type is a floating-point type, the object casts the coefficients to the same data type as the input. When the input data type is an integer type or a fixed-point type, you can modify the coefficient data type by using the `CoefficientsDataType` property.

Example: `firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0])` defines coefficients using a linear-phase filter design function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`FilterStructure` — HDL filter architecture

`'Direct form systolic'` (default) | `'Direct form transposed'`

HDL filter architecture, specified as one of these structures:

- `'Direct form systolic'` — This architecture provides a fully parallel or partly serial filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For a partly serial implementation, specify a value greater than 1 for the `NumCycles` property. You cannot use frame-based input with the partly-serial architecture.

  When `NumCycles` is greater than 1, the object chooses a filter architecture that results in the fewest multipliers. If `NumCycles` allows for a single multiplier in each subfilter, then the object implements a single serial filter and decimates the output samples.
- `'Direct form transposed'` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications.

The object implements a polyphase decomposition filter by using `dsphdl.FIRFilter` objects. All implementations share resources by interleaving the subfilter coefficients over one filter implementation for each sample in the input vector.

For architecture details, see “FIR Filter Architectures for FPGAs and ASICs”.

`DecimationFactor` — Decimation factor

2 (default) | integer greater than two

Decimation factor, specified as integer greater than two. When you use vector input and the vector size is less than the decimation factor, the decimation factor must be an integer multiple of the vector size. When you use vector input and the vector size is greater than the decimation factor, the vector size must be an integer multiple of the decimation factor.

`NumCycles` — Serialization requirement for input timing

1 (default) | positive integer

Serialization requirement for input timing, specified as a positive integer. This property represents *N*, the minimum number of cycles between valid input samples. To implement a fully serial architecture, set `NumCycles` to a value greater than the filter length, *L*, or to `Inf`.

The object applies coefficient optimizations before serialization, so the sharing factor of the final filter can be lower than the number of cycles that you specified.

Dependencies

To enable this property, set `FilterStructure` to `'Direct form systolic'`.

You cannot use frame-based input with `NumCycles` greater than 1.

Data Types

`RoundingMethod` — Rounding method for type-casting output

`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method for type-casting the output, specified as `'Floor'`, `'Ceiling'`, `'Convergent'`, `'Nearest'`, `'Round'`, or `'Zero'`. The object uses this property when casting the output to the data

type specified by the `OutputDataType` property. When the input data type is floating point, the object ignores this property. For more details, see “Rounding Modes”.

`OverflowAction` — Overflow handling for type-casting the output
`'Wrap'` (default) | `'Saturate'`

Overflow handling for type-casting the output, specified as `'Wrap'` or `'Saturate'`. The object uses this property when casting the output to the data type specified by the `OutputDataType` property. When the input data type is floating point, the object ignores this property. For more details, see “Overflow Handling”.

`CoefficientsDataType` — Data type of filter coefficients
`'Same word length as input'` (default) | `numerictype` object

Data type of filter coefficients, specified as `'Same word length as input'` or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- *s* is `1` for signed and `0` for unsigned.
- *w* is the word length in bits.
- *f* is the number of fractional bits.

The object casts the filter coefficients to the specified data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the object ignores this property.

`OutputDataType` — Data type of filter output
`'Full precision'` (default) | `'Same word length as input'` | `numerictype` object

Data type of the filter output, specified as `'Same word length as input'`, `'Full precision'`, or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- *s* is `1` for signed and `0` for unsigned.
- *w* is the word length in bits.
- *f* is the number of fractional bits.

The object casts the output of the filter to the specified data type. The quantization uses the settings of the `RoundingMethod` and `OverflowAction` properties. When the input data type is floating point, the object ignores this property.

The object increases the word length for full precision inside each filter tap and casts the final output to the specified type. The maximum final internal data type (*WF*) depends on the input data type (*WI*), the coefficient data type (*WC*), and the number of coefficients (*L*) and is given by

$WF = WI + WC + \text{ceil}(\log2(L))$.

Because the coefficient values limit the potential growth, usually the actual full-precision internal word length is smaller than *WF*.

Control Arguments

`ResetInputPort` — Option to enable reset input argument
`false` (default) | `true`

Option to enable reset input argument, specified as `true` or `false`. When you set this property to `true`, the object expects a value for the `reset` input argument. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

`HDLGlobalReset` — Option to connect data path registers to generated HDL global reset signal
`false` (default) | `true`

Option to connect data path registers to generated HDL global reset signal, specified as `true` or `false`. Set this property to `true` to connect the generated HDL global reset signal to the data path registers. This property does not change the arguments of the object or modify simulation behavior in MATLAB. When you set this property to `false`, the generated HDL global reset clears only the control path registers. The generated HDL global reset can be synchronous or asynchronous depending on your HDL code generation settings.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Usage

Syntax

```
[dataOut,validOut] = firDecim(dataIn,validIn)
[dataOut,validOut] = firDecim(dataIn,validIn,reset)
```

Description

`[dataOut,validOut] = firDecim(dataIn,validIn)` filters the input data only when `validIn` is `true`.

`[dataOut,validOut] = firDecim(dataIn,validIn,reset)` filters data when `reset` is `false`. When `reset` is `true`, the object resets the filter registers. The object expects the `reset` argument only when you set the `ResetInputPort` property to `true`.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Input Arguments

`dataIn` — Input data
scalar | vector

Input data, specified as a real- or complex-valued scalar or vector. When you use vector input and the vector size is less than the decimation factor, the decimation factor must be an integer multiple of the vector size. When you use vector input and the vector size is greater than the decimation factor, the vector size must be an integer multiple of the decimation factor. The vector size must be less than or equal to 64.

When the input data type is an integer type or fixed-point type, the object uses fixed-point arithmetic for internal calculations.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`
Complex Number Support: Yes

`validIn` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

`reset` — Clears internal states

logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set the `ResetInputPort` property to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — Filtered output data

scalar | vector

Filtered output data, returned as a real- or complex-valued scalar or column vector. When the input data is floating point, the output data inherits the data type of the input data. When the input data is an integer type or fixed-point type, the OutputDataType property specifies the output data type.

The output `valid` signal indicates which samples are valid after decimation. When the input vector size is greater than the decimation factor, the output is a vector of *VectorSize*/*DecimationFactor* samples.

Data Types: `fi` | `single` | `double`

`validOut` — Indicates valid output data

scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to dsphdl.FIRDecimator

getLatency	Latency of FIR decimation filter

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Algorithms

This System object implements the algorithms described on the FIR Decimator block reference page.

Note The output of the `dsphdl.FIRDecimator` object does not match the output from the `dsp.FIRDecimation` object sample-for-sample. This difference is mainly because of the phase that the samples are applied across the subfilters. To match the `dsp.FIRDecimation` object, apply `DecimationFactor` – 1 zeroes to the `dsphdl.FIRDecimator` object at the start of the data stream.

The `dsp.FIRDecimation` object also uses slightly different data types for full-precision calculations. The different data types can also introduce differences in output values if the values overflow the internal datatypes.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox

Behavior changed in R2022a

Before R2022a, this System object was named `dsp.HDLFIRDecimator` and was part of the DSP System Toolbox product.

Serial systolic architecture

This object now supports partly and fully serial systolic architecture. This architecture enables you to share hardware resources if there is a regular pattern of invalid cycles between valid input samples. To use the serial systolic architecture, set `Structure` to `'Direct form systolic'` and `NumCycles` to a value greater than 1.

Input vector size can be greater than decimation factor

In previous releases, the object did not support input vector sizes greater than the decimation factor.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Objects

`dsphdl.FIRFilter`

Blocks

FIR Decimator | Discrete FIR Filter

Introduced in R2020b

dsphdl.FIRInterpolator

Package: `dsphdl`

Finite impulse response (FIR) interpolation filter

Description

The `dsphdl.FIRInterpolator` System object implements a single-rate polyphase FIR interpolation filter that is optimized for HDL code generation. The object provides a hardware-friendly interface with input and output control signals. To provide a cycle-accurate simulation of the generated HDL code, the object models architectural latency including pipeline registers and resource sharing.

The object accepts scalar or vector input and outputs a scalar or vector depending on the interpolation factor and the number of cycles between input samples. The object implements a polyphase decomposition with *InterpolationFactor* subfilters. Each subfilter can implement a serial architecture if there is regular spacing between input samples.

The object provides two filter structures. The direct form systolic architecture provides a fully parallel implementation that makes efficient use of Intel and Xilinx DSP blocks. The direct form transposed architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications. 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.

All filter structures optimize hardware resources by sharing multipliers for symmetric or antisymmetric filters and by removing the multipliers for zero-valued coefficients such as in half-band filters and Hilbert transforms.

To filter and interpolate input data with an HDL-optimized algorithm::

1. Create the `dsphdl.FIRInterpolator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
firInterp = dsphdl.FIRInterpolator
firInterp = dsphdl.FIRInterpolator(INTERP,NUM)
firInterp = dsphdl.FIRInterpolator( ___ ,Name,Value)
```

Description

`firInterp = dsphdl.FIRInterpolator` returns a System object `firInterp`, which upsamples and filters the input signal with the default settings.

`firInterp = dsphdl.FIRInterpolator(INTERP,NUM)` returns a System object `firInterp` with the `InterpolationFactor` property set to `INTERP` and the `Numerator` property set to `NUM`.

`firInterp = dsphdl.FIRInterpolator( ___ ,Name,Value)` returns an HDL FIR Interpolation System object `firInterp`, with specified property `Name` set to the specified `Value`. You can specify additional properties as name-value arguments in any order as (`Name1,Value1,...,NameN,ValueN`).

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Main

`FilterStructure` — HDL filter architecture

`'Direct form systolic'` (default) | `'Direct form transposed'`

HDL filter architecture, specified as one of these structures:

- `'Direct form systolic'` — This architecture provides a fully parallel filter implementation that makes efficient use of Intel and Xilinx DSP blocks. For a partly serial implementation, specify a value greater than 1 for the `NumCycles` property. You cannot use frame-based input with the partly serial architecture.
- `'Direct form transposed'` — This architecture is a fully parallel implementation that is suitable for FPGA and ASIC applications.

The object implements a polyphase decomposition filter by using `dsphdl.FIRFilter` System objects. Each filter phase shares resources internally where coefficients and serial options allow. For architecture details, see “FIR Filter Architectures for FPGAs and ASICs”.

`Numerator` — FIR filter coefficients

`fir1(35,0.4)` (default) | real- or complex-valued vector

FIR filter coefficients, specified as a real- or complex-valued vector. You can specify the vector as a workspace variable or as a call to a filter design function. When the input data type is a floating-point type, the object casts the coefficients to the same data type as the input. When the input data type is an integer type or a fixed-point type, you can modify the coefficient data type by using the `CoefficientsDataType` property.

Example: `firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0])` defines coefficients by using a linear-phase filter design function.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`InterpolationFactor` — Interpolation factor

2 (default) | integer greater than two

Interpolation factor, specified as integer greater than two. The output vector size is *InputSize* * *InterpolationFactor*. The output vector size must be less than 64 samples.

`NumCycles` — Minimum number of cycles between valid input samples

`1` (default) | positive integer

Specify the minimum number of cycles between the valid input samples as `1` or a positive integer. This property represents *N*, the minimum number of cycles between valid input samples. When you set `NumCycles` greater than the filter length, *L*, and the input and coefficients are both real, the filter uses `InterpolationFactor` multipliers.

Because the object applies coefficient optimizations before serialization, the sharing factor of the final filter can be lower than the number of cycles that you specified.

Dependencies

To enable this parameter, set `FilterStructure` to `'Direct form systolic'`.

You cannot use frame-based input with `NumCycles` greater than 1.

Data Types

`RoundingMethod` — Rounding method for type-casting output

`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method for type-casting the output, specified as `'Floor'`, `'Ceiling'`, `'Convergent'`, `'Nearest'`, `'Round'`, or `'Zero'`. The object uses this property when casting the output to the data type specified by the `OutputDataType` property. When the input data type is floating point, the object ignores this property. For more details, see “Rounding Modes”.

`OverflowAction` — Overflow handling for type-casting the output

`'Wrap'` (default) | `'Saturate'`

Overflow handling for type-casting the output, specified as `'Wrap'` or `'Saturate'`. The object uses this property when casting the output to the data type specified by the `OutputDataType` property. When the input data type is floating point, the object ignores this property. For more details, see “Overflow Handling”.

`CoefficientsDataType` — Data type of filter coefficients

`'Same word length as input'` (default) | `numerictype` object

Data type of filter coefficients, specified as `'Same word length as input'` or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- *s* is `1` for signed and `0` for unsigned.
- *w* is the word length in bits.
- *f* is the number of fractional bits.

The object casts the filter coefficients to the specified data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the object ignores this property.

The recommended data type for this parameter is `'Same word length as input'`.

`OutputDataType` — Data type of filter output

`'Full precision'` (default) | `'Same word length as input'` | `numerictype` object

Data type of the filter output, specified as `'Same word length as input'`, `'Full precision'`, or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- *s* is `1` for signed and `0` for unsigned.
- *w* is the word length in bits.
- *f* is the number of fractional bits.

The object casts the output of the filter to the specified data type. The quantization uses the settings of the `RoundingMethod` and `OverflowAction` properties. When the input data type is floating point, the object ignores this property.

The object increases the word length for full precision inside each filter tap and casts the final output to the specified type. The maximum final internal data type (*WF*) depends on the input data type (*WI*), the coefficient data type (*WC*), and the number of coefficients (*L*) and is given by

WF = *WI* + *WC* + ceil(log2(*L*)).

Because the coefficient values limit the potential growth, usually the actual full-precision internal word length is smaller than *WF*.

Control Arguments

`ResetInputPort` — Option to enable reset input argument
`false` (default) | `true`

Option to enable reset input argument, specified as `true` or `false`. When you set this property to `true`, the object expects a value for the `reset` input argument. The reset signal implements a local synchronous reset of the data path registers.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

`HDLGlobalReset` — Option to connect data path registers to generated HDL global reset signal
`false` (default) | `true`

Option to connect data path registers to generated HDL global reset signal, specified as `true` or `false`. Set this property to `true` to connect the generated HDL global reset signal to the data path registers. This property does not change the arguments of the object or modify simulation behavior in MATLAB. When you set this property to `false`, the generated HDL global reset clears only the control path registers. The generated HDL global reset can be synchronous or asynchronous depending on your HDL code generation settings.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Usage

Syntax

```
[dataOut,validOut] = firInterp(dataIn,validIn)
[dataOut,validOut] = firInterp(dataIn,validIn,reset)
```

Description

`[dataOut,validOut] = firInterp(dataIn,validIn)` filters the input data only when `validIn` is `true`.

`[dataOut,validOut] = firInterp(dataIn,validIn,reset)` filters data when `reset` is `false`. When `reset` is `true`, the object resets the filter registers. The object expects the `reset` argument only when you set the `ResetInputPort` property to `true`.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Input Arguments

`dataIn` — Input data

real or complex scalar or vector

Input data, specified as a real- or complex-valued scalar or vector. The vector size must be less than or equal to 64. When the input data type is an integer type or fixed-point type, the object uses fixed-point arithmetic for internal calculations.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`
Complex Number Support: Yes

`validIn` — Indicates valid input data

scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

`reset` — Clears internal states

logical scalar

Control signal that clears internal states. When `reset` is `1` (`true`), the object stops the current calculation and clears internal states. When the `reset` is `0` (`false`) and the input `valid` is `1` (`true`), the block captures data for processing.

For more reset considerations, see the “Reset Signal” section on the “Hardware Control Signals” page.

Dependencies

To enable this argument, set the `ResetInputPort` property to `true`.

Data Types: `logical`

Output Arguments

`dataOut` — Interpolated output data

real or complex scalar or vector

Interpolated output data, returned as a real or complex scalar or vector. The vector size is *InputSize* * *InterpolationFactor*. When `NumCycles` is greater than `InterpolationFactor`, scalar output

samples are spaced out with `floor(NumCycles/InterpolationFactor)` invalid cycles, and the output `valid` signal indicates which samples are valid after interpolation.

When the input data is floating point, the output data inherits the data type of the input data. When the input data is an integer type or fixed-point type, the OutputDataType property specifies the output data type.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`validOut` — Indicates valid output data
scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

`ready` — Indicates object is ready for new input data
logical scalar

Control signal that indicates that the object is ready for new input data sample on the next cycle. When `ready` is `1` (`true`), you can specify the `data` and `valid` inputs for the next time step. When `ready` is `0` (`false`), the object ignores any input data in the next time step.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to dsphdl.FIRInterpolator

getLatency	Latency of FIR filter

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Algorithms

This System object implements the algorithms described on the FIR Interpolator block reference page.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Objects
`dsphdl.FIRDecimator`

Blocks
FIR Interpolator | FIR Decimator | Discrete FIR Filter

Introduced in R2022a

dsphdl.BiquadFilter

Package: `dsphdl`

Biquadratic IIR (SOS) filter

Description

A biquad filter is a form of infinite-impulse response (IIR) filter where the numerator and denominator are split into a series of second-order sections connected by gain blocks. This type of filter can replace a large FIR filter that uses an impractical amount of hardware resources. Designs often use biquad filters as DC blocking filters or to meet a specification originally implemented with an analog filter, such as a pre-emphasis filter.

To filter input data with an HDL-optimized biquad filter:

1. Create the `dsphdl.BiquadFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
myfilt = dsphdl.BiquadFilter
myfilt = dsphdl.BiquadFilter(Name,Value)
```

Description

`myfilt = dsphdl.BiquadFilter` creates an HDL-optimized biquad filter. The default filter is a direct form II architecture with one section.

`myfilt = dsphdl.BiquadFilter(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

For example:

```
myfilt = dsphdl.BiquadFilter('Structure','Direct form II transposed');
[dataOut,validOut] = myfilt(dataIn,validIn);
```

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Structure` — HDL filter architecture
`'Direct form II'` (default) | `'Direct form II transposed'` | `'Pipelined feedback form'`

Both the `'Direct form II'` and `'Direct form II transposed'` architectures are pipelined and quantized to fit well into FPGA DSP blocks. The output of these filters matches the output of the DSP System Toolbox System objects `dsp.SOSFilter` and `dsp.FourthOrderSectionFilter`. These architectures minimize the number of multipliers used by the filter but have a critical path through the feedback loop and sometimes cannot achieve higher clock rates.

`'Pipelined feedback form'` implements a pipelined architecture that uses more multipliers than either direct form II structure, but achieves higher clock rates after synthesis. Frame-based input is supported only when you use `'Pipelined feedback form'`. The output of the pipelined filter is slightly different than the DSP System Toolbox functions `dsp.SOSFilter` and `dsp.FourthOrderSectionFilter` because of the timing of data samples applied in the pipelined filter stages.

`Numerator` — Coefficients of filter numerator
`[1,2,1]` (default) | *NumSections*-by-3 matrix

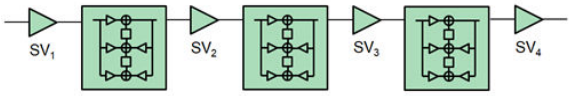
Specify the numerator coefficients as a matrix of *NumSections*-by-3 values. *NumSections* is the number of second-order filter sections. The object infers the number of filter sections from the size of the numerator and denominator coefficients. The numerator coefficient and denominator coefficient matrices must be the same size. The default filter has one section.

`Denominator` — Coefficients of filter denominator
`[1,.1,.2]` (default) | *NumSections*-by-3 matrix

Specify the denominator coefficients as a matrix of *NumSections*-by-3 values. The object assumes the first denominator coefficient of each section is `1.0`.*NumSections* is the number of second-order filter sections. The object infers the number of sections from the size of the numerator and denominator coefficients. The numerator coefficient and denominator coefficient matrices must be the same size. The default filter has one section.

`ScaleValues` — Gain values applied before and after second-order filter sections
`[1]` (default) | vector of 1 to *NumSections*+1 values

Specify the gain values as a vector of up to *NumSections*+1 values. *NumSections* is the number of second-order filter sections. The object infers the number of sections from the size of the numerator and denominator coefficients. If the vector has only one value, the object applies that gain before the first section. If you specify fewer values than there are filter sections, the object sets the remaining section gain values to one. The diagram shows a 3-section filter and the locations of the four scale values before and after the sections.



Implementing these gain factors outside the filter sections reduces the multipliers needed to implement the numerator of the filter.

Data Types

`Rounding` — Rounding method for type-casting the output
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding mode for type-casting the output and accumulator values to the data types specified by the `OutputDataType` and `AccumulatorDataType` properties. When the input data type is floating point, the object ignores this parameter. For more details, see “Rounding Modes”.

`OverflowAction` — Overflow handling for type-casting the output
`'Wrap'` (default) | `'Saturate'`

Overflow handling for type-casting the output and accumulator values to the data types specified by the `OutputDataType` and `AccumulatorDataType` properties. When the input data type is floating point, the object ignores the `OverflowAction` property. For more details, see “Overflow Handling”.

`NumeratorDataType` — Data type of numerator coefficients
`'Same word length as first input'` (default) | `numerictype` object

Data type of numerator coefficients, specified as `'Same word length as first input'` or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- `s` is `1` for signed and `0` for unsigned.
- `w` is the word length in bits.
- `f` is the number of fractional bits.

The object type-casts the numerator coefficients to the specified data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the object ignores this property.

The object returns a warning if the data type of the coefficients does not have enough fractional length to represent the coefficients accurately.

`DenominatorDataType` — Data type of denominator coefficients
`'Same word length as first input'` (default) | `numerictype` object

Data type of denominator coefficients, specified as `'Same word length as first input'` or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- `s` is `1` for signed and `0` for unsigned.
- `w` is the word length in bits.
- `f` is the number of fractional bits.

The object type-casts the denominator coefficients to the specified data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the object ignores this property.

The object returns a warning if the data type of the coefficients does not have enough fractional length to represent the coefficients accurately.

`ScaleValueDataType` — Data type of section gains
`'Same word length as first input'` (default) | `numerictype` object

Data type of the scale values, specified as `'Same word length as first input'` or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- `s` is `1` for signed and `0` for unsigned.
- `w` is the word length in bits.
- `f` is the number of fractional bits.

The object type-casts the scale values to the specified data type. The quantization rounds to the nearest representable value and saturates on overflow. When the input data type is floating point, the object ignores this property.

`AccumulatorDataType` — Data type of accumulator signals within each section
`'Same as first input'` (default) | `numerictype` object

Data type of accumulator signals within each section (as indicated in the diagrams in the "Algorithms" on page 1-138 section), specified as `'Same as first input'`, or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- `s` is `1` for signed and `0` for unsigned.
- `w` is the word length in bits.
- `f` is the number of fractional bits.

The object type-casts the output of the filter to the specified data type. The quantization uses the settings of the `RoundingMethod` and `OverflowAction` properties. When the input data type is floating point, the object ignores this property.

`OutputDataType` — Data type of filter output
`'Same as first input'` (default) | `'Full precision'` | `numerictype` object

Data type of filter output, specified as `'Same as first input'`, `'Full precision'`, or a `numerictype` object. To specify a `numerictype` object, call `numerictype(s,w,f)`, where:

- `s` is `1` for signed and `0` for unsigned.
- `w` is the word length in bits.
- `f` is the number of fractional bits.

The object type-casts the output of the filter to the specified data type. The quantization uses the settings of the `RoundingMethod` and `OverflowAction` properties. When the input data type is floating point, the object ignores this property.

Usage

Syntax

```
[dataOut,validOut] = myfilt(dataIn,validIn)
```

Description

`[dataOut,validOut] = myfilt(dataIn,validIn)` filters the input data only when `validIn` is `true`.

Input Arguments

`dataIn` — Input data
scalar or column vector of real values

Input data, specified as a scalar or column vector of real values. When the input has an integer or fixed-point data type, the object uses fixed-point arithmetic for internal calculations.

Vector input is supported only when you set the `Structure`property to `'Pipelined feedback form'`. The object accepts vectors up to 64 samples, but large vector sizes can make the calculation of internal data types challenging. Vector sizes of up to 16 samples are practical for hardware implementation.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`validIn` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

Output Arguments

`dataOut` — Filtered output data
scalar or column vector of real values

Filtered output data, returned as a scalar or column vector of real values. The output dimensions match the input dimensions. When the input data is floating point, the output data inherits the data type of the input data. When the input data is an integer type or fixed-point type, the `OutputDataType` property determines the output data type.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`validOut` — Indicates valid output data
scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Algorithms

This System object implements the algorithms described on the Biquad Filter block reference page.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Biquad Filter

Introduced in R2022a

dsphdl.ComplexToMagnitudeAngle

Package: `dsphdl`

Magnitude and phase angle of complex signal

Description

The `dsphdl.ComplexToMagnitudeAngle` System object computes the magnitude and phase angle of a complex signal. It provides hardware-friendly control signals. The System object uses a pipelined coordinate rotation digital computer (CORDIC) algorithm to achieve an HDL-optimized implementation.

To compute the magnitude and phase angle of a complex signal:

1. Create the `dsphdl.ComplexToMagnitudeAngle` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
magAngle = dsphdl.ComplexToMagnitudeAngle
magAngle = dsphdl.ComplexToMagnitudeAngle(Name,Value)
```

Description

`magAngle = dsphdl.ComplexToMagnitudeAngle` returns a `dsphdl.ComplexToMagnitudeAngle` System object, `magAngle`, that computes the magnitude and phase angle of a complex input signal.

`magAngle = dsphdl.ComplexToMagnitudeAngle(Name,Value)` sets properties of `magAngle` using one or more name-value pairs. Enclose each property name in single quotes.

Example: `magAngle = dsphdl.ComplexToMagnitudeAngle('AngleFormat','Radians')`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`OutputValue` — Type of values to return
`'Magnitude and angle'` (default) | `'Magnitude'` | `'Angle'`

Type of output values to return, specified as `'Magnitude and angle'`, `'Magnitude'`, or `'Angle'`. You can choose for the object to return the magnitude of the input signal, or the phase angle of the input signal, or both.

`AngleFormat` — Format of phase angle output value
`'Normalized'` (default) | `'Radians'`

Format of the phase angle output value from the object, specified as:

- `'Normalized'` — Fixed-point format that normalizes the angle in the range [–1,1].
- `'Radians'` — Fixed-point values in the range [π,−π].

`ScaleOutput` — Scale output by inverse of CORDIC gain factor
`true` (default) | `false`

Scale output by the inverse of the CORDIC gain factor, specified as `true` or `false`. The object implements this gain factor with either CSD logic or a multiplier, according to the `ScalingMethod` parameter.

Note If your design includes a gain factor later in the datapath, you can set `ScaleOutput` to `false`, and include the CORDIC gain factor in the later gain. For calculation of this gain factor, see "Algorithm" on page 2-159. The object replaces the first CORDIC iteration by mapping the input value onto the angle range [0,π/4]. Therefore, the initial rotation does not contribute a gain term.

`NumIterationsSource` — Source of `NumIterations`
`'Auto'` (default) | `'Property'`

Source of the `NumIterations` property for the CORDIC algorithm, specified as:

- `'Auto'` — Sets the number of iterations to one less than the input word length. If the input is `double` or `single`, the number of iterations is 16.
- `'Property'` — Uses the `NumIterations` property.

For details of the CORDIC algorithm, see "Algorithm" on page 2-159.

`NumIterations` — Number of CORDIC iterations
integer less than or equal to one less than the input word length

Number of CORDIC iterations that the object executes, specified as an integer. The number of iterations must be less than or equal to one less than the input word length.

For details of the CORDIC algorithm, see "Algorithm" on page 2-159.

Dependencies

To enable this property, set `NumIterationsSource` to `'Property'`.

`ScalingMethod` — Implementation of CORDIC gain scaling
`'CSD'` (default) | `'Multipliers'`

When you set this property to `'CSD'`, the object implements the CORDIC gain scaling by using a shift-and-add architecture for the multiply operation. This implementation uses no multiplier resources and may increase the length of the critical path in your design. When you set this property

to `'Multipliers'`, the object implements the CORDIC gain scaling with a multiplier and increases the latency of the object by four cycles.

Dependencies

To enable this property, set the `ScaleOutput` property to `true`.

Usage

Syntax

```
[mag,angle,validOut] = magAngle(X,validIn)
[mag,validOut] = magAngle(X,validIn)
[angle,validOut] = magAngle(X,validIn)
```

Description

`[mag,angle,validOut] = magAngle(X,validIn)` converts a scalar or vector of complex values `X` into their component magnitude and phase angles. `validIn` and `validOut` are logical scalars that indicate the validity of the input and output signals, respectively.

`[mag,validOut] = magAngle(X,validIn)` returns only the component magnitudes of `X`.

To use this syntax, set `OutputValue` to `'Magnitude'`.

Example: `magAngle = dsphdl.ComplextoMagnitudeAngle('OutputValue','Magnitude');`

`[angle,validOut] = magAngle(X,validIn)` returns only the component phase angles of `X`.

To use this syntax, set `OutputValue` to `'Angle'`.

Example: `magAngle = dsphdl.ComplextoMagnitudeAngle('OutputValue','Angle');`

Input Arguments

`X` — Input signal
complex scalar or vector

Input signal, specified as a scalar, a column vector representing samples in time, or a row vector representing channels. Using vector input increases data throughput while using more hardware resources. The object implements the conversion logic in parallel for each element of the vector. The input vector can contain up to 64 elements.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `single` | `double`

`validIn` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

Data Types: `logical`

Output Arguments

`mag` — Magnitude component of input signal
scalar | vector

Magnitude calculated from the complex input signal, returned as a scalar, a column vector representing samples in time, or a row vector representing channels. The dimensions and data type of this argument match the dimensions of the `dataIn` argument.

Dependencies

To enable this argument, set the `OutputValue` property to `'Magnitude and Angle'` or `'Magnitude'`.

`angle` — Phase angle component of input signal
scalar | vector

Angle calculated from the complex input signal, returned as a scalar, a column vector representing samples in time, or a row vector representing channels. The dimensions and data type of this argument match the dimensions of the `dataIn` argument. The format of this value depends on the `AngleFormat` property.

Dependencies

To enable this argument, set the `OutputValue` property to `'Magnitude and Angle'` or `'Angle'`.

`validOut` — Indicates valid output data
scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `mag` and/or `angle` arguments. When `validOut` is `0` (`false`), values from the `mag` and/or `angle` arguments are not valid.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

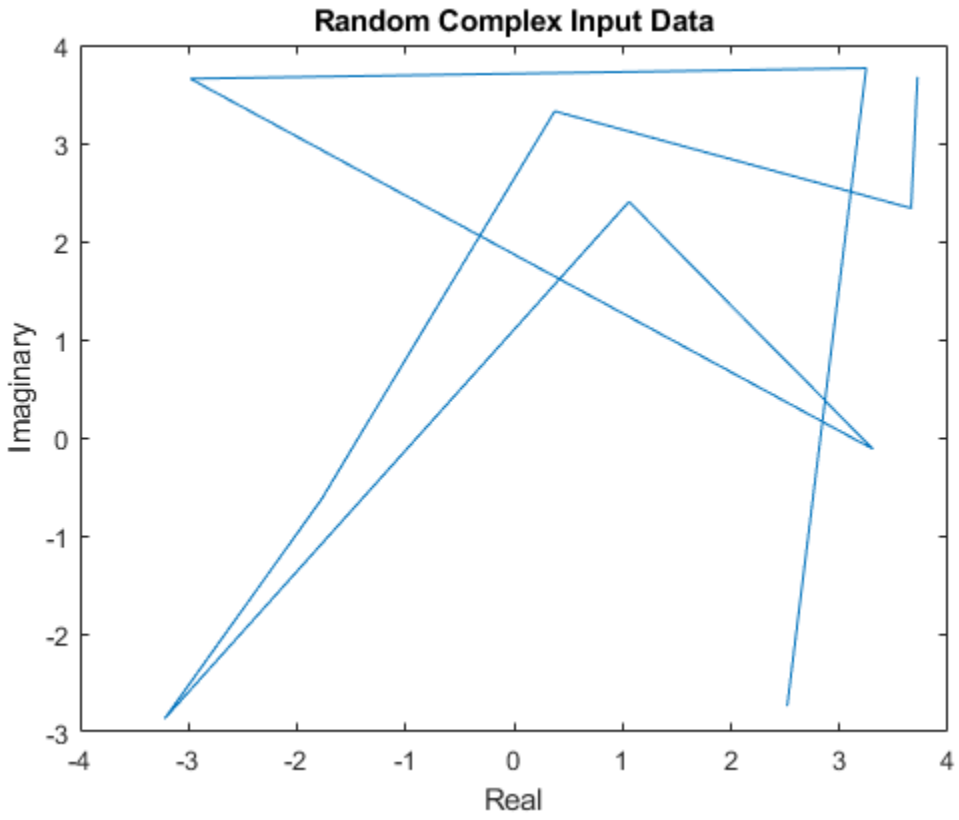
Examples

Compute Magnitude and Phase Angle of Complex Signal

Use the `dsphdl.ComplextoMagnitudeAngle` object to compute the magnitude and phase angle of a complex signal. The object uses a CORDIC algorithm for an efficient hardware implementation.

Choose word lengths and create random complex input signal. Then, convert the input signal to fixed-point.

```
a = -4;
b = 4;
inputWL = 16;
inputFL = 12;
numSamples = 10;
reData = ((b-a).*rand(numSamples,1)+a);
imData = ((b-a).*rand(numSamples,1)+a);
dataIn = (fi(reData+imData*1i,1,inputWL,inputFL));
figure
plot(dataIn)
title('Random Complex Input Data')
xlabel('Real')
ylabel('Imaginary')
```



Write a function that creates and calls the System object™. You can generate HDL from this function.

```
function [mag,angle,validOut] = Complex2MagAngle(yIn,validIn)
%Complex2MagAngle
% Converts one sample of complex data to magnitude and angle data.
% yIn is a fixed-point complex number.
% validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent cma;
```

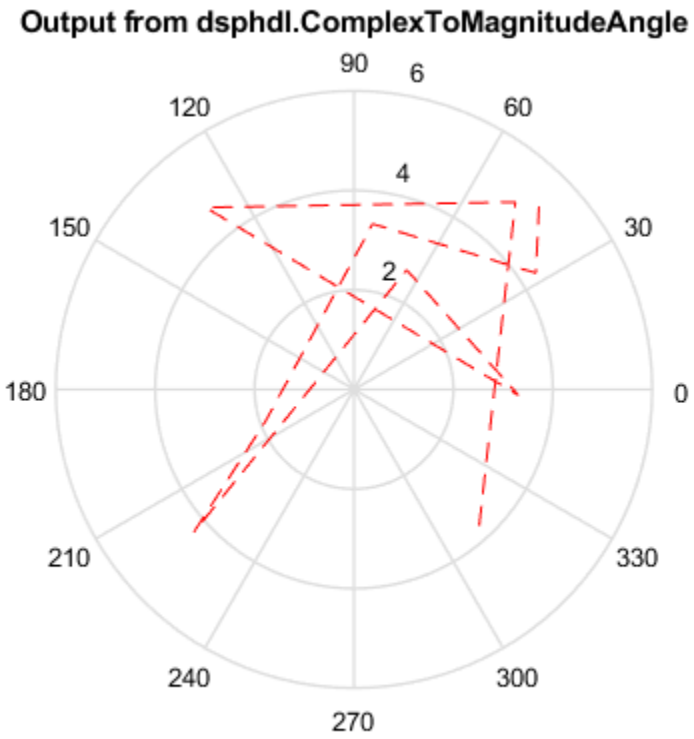
```
  if isempty(cma)
    cma = dsphdl.ComplexToMagnitudeAngle('AngleFormat','Radians');
  end
  [mag,angle,validOut] = cma(yIn,validIn);
end
```

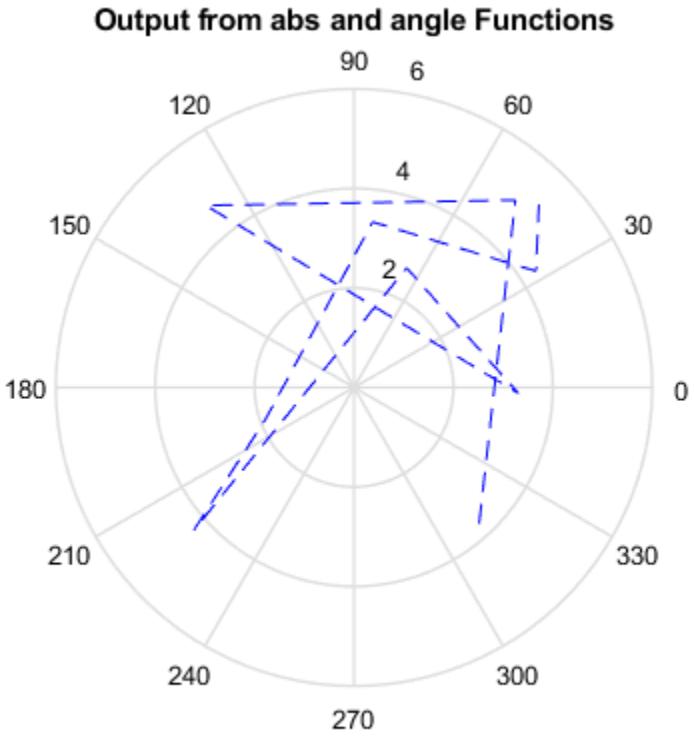
The number of CORDIC iterations determines the latency that the object takes to compute the answer for each input sample. The latency is `NumIterations+4`. In this example, `NumIterationsSource` is set to the default, `'Auto'`, so the object uses `inputWL-1` iterations. The latency is `inputWL+3`.

```
latency = inputWL+3;
mag = zeros(1,numSamples+latency);
ang = zeros(1,numSamples+latency);
validOut = false(1,numSamples+latency);
```

Call the function to convert each sample. After you apply all input samples, continue calling the function with invalid input to flush remaining output samples.

```
for ii = 1:1:numSamples
   [mag(ii),ang(ii),validOut] = Complex2MagAngle(dataIn(ii),true);
end
for ii = (numSamples+1):1:(numSamples+latency)
   [mag(ii),ang(ii),validOut(ii)] = Complex2MagAngle(fi(0+0*1i,1,inputWL,inputFL),false);
end
% Remove invalid output values
mag = mag(validOut == 1);
ang = ang(validOut == 1);
figure
polar(ang,mag,'--r')   % Red is output from System object
title('Output from dsphdl.ComplexToMagnitudeAngle')
magD = abs(dataIn);
angD = angle(dataIn);
figure
polar(angD,magD,'--b') % Blue is output from abs and angle functions
title('Output from abs and angle Functions')
```


Output from dsphdl.ComplexToMagnitudeAngle
90
6
120
60
4
150
30
2
180
0
210
330
240
300
270



Algorithms

CORDIC Algorithm

The CORDIC algorithm is a hardware-friendly method for performing trigonometric functions. It is an iterative algorithm that approximates the solution by converging toward the ideal point. The object uses CORDIC vectoring mode to iteratively rotate the input onto the real axis.

The Givens method for rotating a complex number $x+iy$ by an angle θ is as follows. The direction of rotation, d, is $+1$ for counterclockwise and -1 for clockwise.

$$x_r = x\cos\theta - dy\sin\theta$$
$$y_r = y\cos\theta + dx\sin\theta$$

For a hardware implementation, factor out the $\cos\theta$ to leave a $\tan\theta$ term.

$$x_r = \cos\theta(x - dy\tan\theta)$$
$$y_r = \cos\theta(y + dx\tan\theta)$$

To rotate the vector onto the real axis, choose a series of rotations of θ_n so that $\tan\theta_n = 2^{-n}$. Remove the $\cos\theta$ term so each iterative rotation uses only shift and add operations.

$$Rx_n = x_{n-1} - d_n y_{n-1} 2^{-n}$$
$$Ry_n = y_{n-1} + d_n x_{n-1} 2^{-n}$$

Combine the missing cosθ terms from each iteration into a constant, and apply it with a single multiplier to the result of the final rotation. The output magnitude is the scaled final value of x. The output angle, z, is the sum of the rotation angles.

$$x_r = (\cos\theta_0\cos\theta_1...\cos\theta_n)Rx_N$$

$$z = \sum_{0}^{N} d_n\theta_n$$

Modified CORDIC Algorithm

The convergence region for the standard CORDIC rotation is ≈±99.7°. To work around this limitation, before doing any rotation, the object maps the input into the [0,π/4] range using the following algorithm.

```
if abs(x) > abs(y)
  input_mapped = [abs(x), abs(y)];
else
  input_mapped = [abs(y), abs(x)];
end
```

At each iteration, the object rotates the vector towards the real axis. The rotation is counterclockwise when y is negative, and clockwise when y is positive.

Quadrant mapping saves hardware resources and reduces latency by reducing the number of CORDIC pipeline stages by one. The CORDIC gain factor, Kn, therefore does not include the n=0, or cos(π/4) term.

$$K_n = cos\theta_1...\cos\theta_n = \cos(26.565)\cdot\cos(14.036)\cdot\cos(7.125)\cdot\cos(3.576)$$

After the CORDIC iterations are complete, the object corrects the angle back to its original location. First it adjusts the angle to the correct side of π/4.

```
if abs(x) > abs(y)
  angle_unmapped = CORDIC_out;
else
  angle_unmapped = (pi/2) - CORDIC_out;
end
```

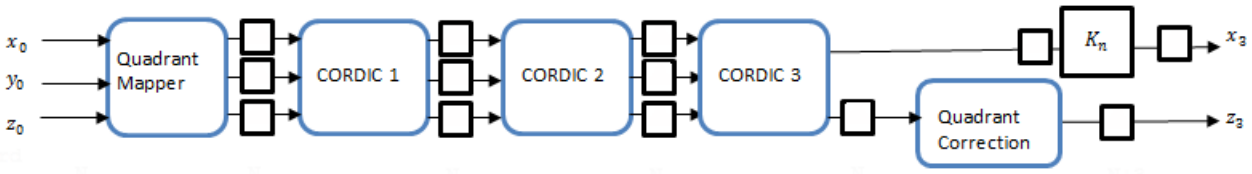
Then it flips the angle to the original quadrant.

```
if (x < 0)
  if (y < 0)
    output_angle = - pi + angle_unmapped;e
  else
    output_angle = pi - angle_unmapped;
else
  if (y<0)
    output_angle = -angle_unmapped;
```

Architecture

The object generates a pipelined HDL architecture to maximize throughput. Each CORDIC iteration is done in one pipeline stage. The gain multiplier, if enabled, is implemented with Canonical Signed Digit (CSD) logic.

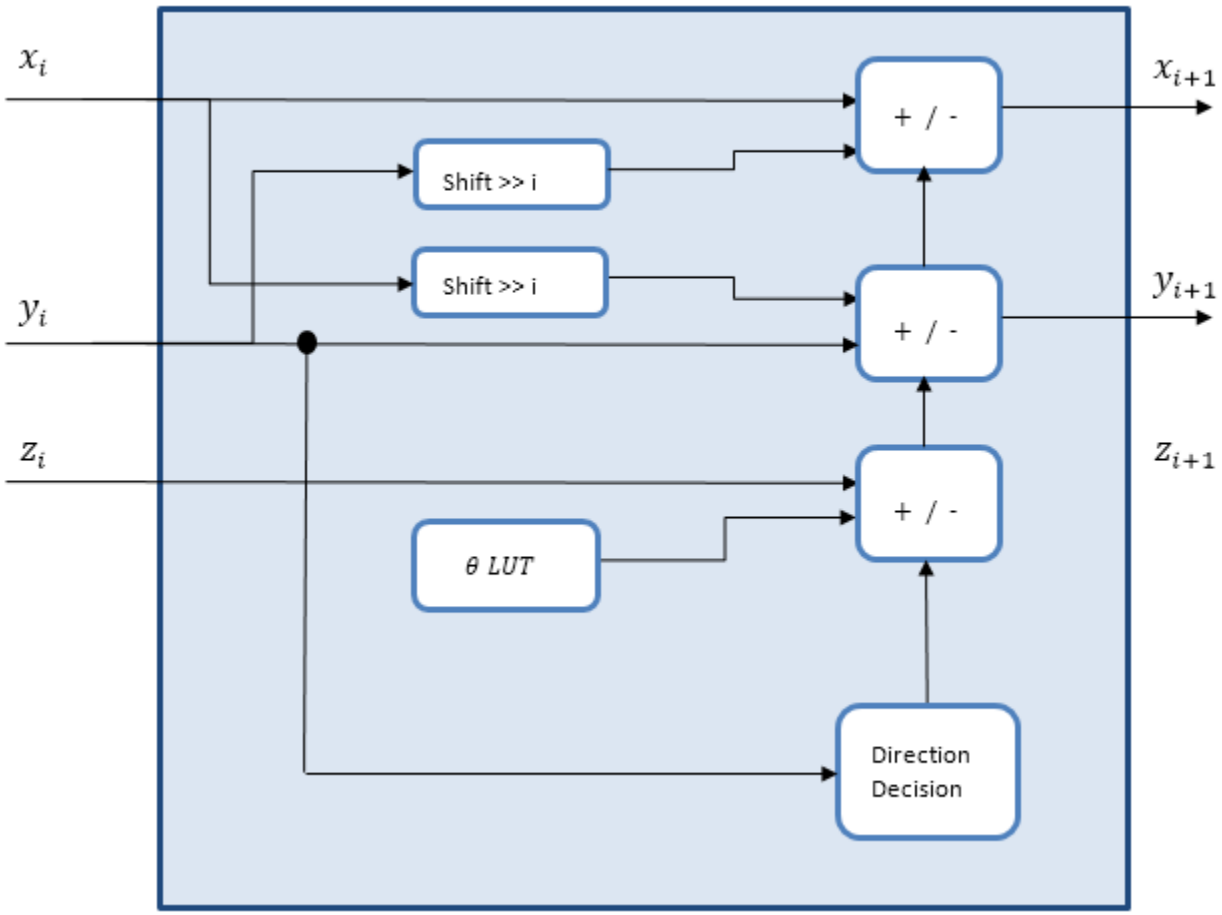
If you use vector input, this object replicates this architecture in parallel for each element of the vector.



Input Word Length	Output Magnitude Word Length
fixdt(0,WL,FL)	fixdt(0,WL+2,FL)
fixdt(1,WL,FL)	fixdt(1,WL+1,FL)

Input Word Length	Output Angle Word Length	
fixdt([ ],WL,FL)	Radians	fixdt(1,WL+3,WL)
	Normalized	fixdt(1,WL+3,WL+2)

The CORDIC logic at each pipeline stage implements one iteration. For each pipeline stage, the shift and angle rotation are constants.

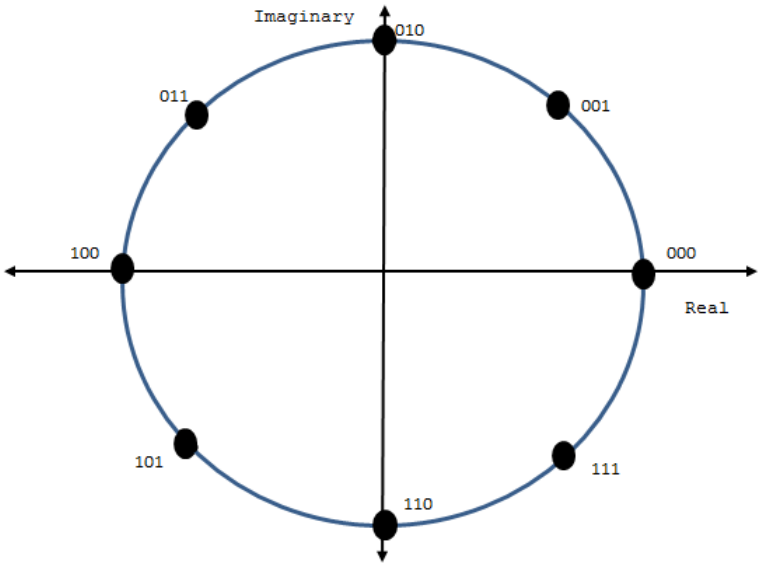


When you set `OutputValue` to `'Magnitude'`, the object does not generate HDL code for the angle accumulation and quadrant correction logic.

Normalized Angle Format

This format normalizes the fixed-point radian angle values around the unit circle. This is a more efficient use of bits than a range of [0,2π] radians. Normalized angle format also enables wraparound at 0/2π without additional detect and correct logic.

For example, representing the angle with 3 bits results in the following normalized values.



Using the mapping described in “Modified CORDIC Algorithm” on page 1-129, the object normalizes the angles across [0,π/4] and maps them to the correct octant at the end of the calculation.

Delay

The latency is `NumIterations` + 4 cycles from input to output. Each call to the object models one clock cycle.

When you set `NumIterationsSource` to `'Auto'`, the number of iterations is one less than the input word length and the latency is three more than the input word length. If the data type of the input is `double` or `single`, the number of iterations is 16 and the latency is 20.

Note When you set the `ScalingMethod` property to `'Multipliers'`, the object latency increases by four cycles.

Performance

Performance was measured for the default configuration, with output scaling disabled and `fixdt(1,16,12)` input. When the generated HDL code is synthesized into a Xilinx ZC706 (XC7Z045FFG900-2) FPGA, the design achieves 350 MHz clock frequency. It uses the following resources.

Resource	Number Used
LUT	891
FFS	899
Xilinx LogiCORE DSP48	0
Block RAM (16K)	0
Critical path	2.792 ns

When you use a multiplier for the CORDIC gain scaling, the design uses one DSP block and has a shorter critical path. The critical path difference is not significant at this number of bits, but as the

size of the data increases, the critical path of the CSD implementation rises faster than the critical path of the multiplier.

Resource	Number Used
LUT	808
FFS	956
Xilinx LogiCORE DSP48	1
Block RAM (16K)	0
Critical path	2.574 ns

Performance of the synthesized HDL code varies depending on your target and synthesis options. When you use vector input, the resource usage is about *VectorSize* times the scalar resource usage.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this System object was named `dsp.HDLComplexToMagnitudeAngle` and was part of the DSP System Toolbox product.

Option to use multiplier for scale factor

In previous releases, the System object implemented the CORDIC gain for hardware by using shift-and-add logic. To use a multiplier, set the `ScalingMethod` property to `'Multipliers'`. To use shift-and-add logic, set this property to `'CSD'`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Blocks
Complex to Magnitude-Angle

Functions
`cordicangle` | `cordiccart2pol` | `cordicabs` | `angle`

Introduced in R2014b

dsphdl.FIRRateConverter

Package: `dsphdl`

Upsample, filter, and downsample input signal

Description

The `dsphdl.FIRRateConverter` System object upsamples, filters, and downsamples input signals. It is optimized for HDL code generation and operates on one sample of each channel at a time. The object implements an efficient polyphase architecture to avoid unnecessary arithmetic operations and high intermediate sample rates.

The object upsamples by an integer factor of *L*, applies an FIR filter, and downsamples by an integer factor of *M*.

To resample and filter input data:

1. Create the `dsphdl.FIRRateConverter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?

Creation

Syntax

```
HDLFIRRC = dsphdl.FIRRateConverter
HDLFIRRC = dsphdl.FIRRateConverter(L,M,num)
HDLFIRRC = dsphdl.FIRRateConverter( ___ ,Name,Value)
```

Description

`HDLFIRRC = dsphdl.FIRRateConverter` returns a System object, `HDLFIRRC`, that resamples each channel of the input. The object upsamples by an integer factor of *L*, applies an FIR filter, and downsamples by an integer factor of *M*. The default *L/M* is 3/2.

`HDLFIRRC = dsphdl.FIRRateConverter(L,M,num)` sets the `InterpolationFactor` property to `L`, the `DecimationFactor` property to `M`, and the `Numerator` property to `num`.

`HDLFIRRC = dsphdl.FIRRateConverter( ___ ,Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example:

```
HDLFIRRC = dsphdl.FIRRateConverter(L,M,Num,'ReadyPort',true);
```

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`InterpolationFactor` — Upsampling factor

3 (default) | positive integer scalar

Upsampling factor, L, specified as a positive integer.

`DecimationFactor` — Downsampling factor

2 (default) | positive integer scalar

Downsampling factor, M, specified as a positive integer scalar.

`Numerator` — FIR filter coefficients

`firpm(70,[0 .28 .32 1],[1 1 0 0])` (default) | vector in descending powers of z^{-1}

FIR filter coefficients, specified as a vector in descending powers of z^{-1}.

You can generate filter coefficients by using the Signal Processing Toolbox filter design functions, such as `fir1`. Design a lowpass filter with normalized cutoff frequency no greater than `min(1/L,1/M)`. The object initializes internal filter states to zero.

`ReadyPort` — Enable ready argument

`false` (default) | `true`

Enable `ready` output argument of the object. When enabled, the object returns a logical scalar value, `ready`, when you call the object. When `ready` is `1` (true), the object is ready for a new input sample the next time you call it.

`RoundingMethod` — Rounding mode used for fixed-point operations

`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding mode used for fixed-point operations. This property does not apply when the input is `single` or `double` type. `'Simplest'` mode is not supported.

`OverflowAction` — Overflow mode used for fixed-point operations

`'Wrap'` (default) | `'Saturate'`

Overflow mode used for fixed-point operations. This property does not apply when the input is `single` or `double` type.

`CoefficientsDataType` — Data type of the FIR filter coefficients

`numerictype(1,16,16)` (default) | `numerictype(s,wl,fl)`

Data type of the FIR filter coefficients, specified as a `numerictype(s,wl,fl)` object with `signedness`, `word length`, and `fractional length` properties.

`OutputDataType` — Data type of the output data samples
`'Same word length as input'` (default) | `numerictype(s,wl,fl)` | `'Full precision'`

Data type of the output data samples, specified as `'Same word length as input'`, `'Full precision'`, or as a `numerictype(s,wl,fl)` object with `signedness`, `word length`, and `fractional length` properties.

Usage

Syntax

```
[dataOut,validOut] = HDLFIRRC(dataIn,validIn)
[dataOut,validOut,ready] = HDLFIRRC(dataIn,validIn)
```

Description

`[dataOut,validOut] = HDLFIRRC(dataIn,validIn)` resamples `dataIn` according to the `InterpolationFactor` (*L*) and `DecimationFactor` (*M*) properties. To avoid dropped samples when using this syntax, apply new valid input samples, with `validIn` set to `true`, only every `ceil(L/M)` calls to the object. The object sets `validOut` to `true` when `dataOut` is a new valid sample.

`[dataOut,validOut,ready] = HDLFIRRC(dataIn,validIn)` resamples the input data and returns `ready` to indicate whether the object can accept a new sample on the next call.

This syntax applies when you set the `ReadyPort` property to `true`. For example:

```
HDLFIRRC = dsphdl.FIRRateConverter(...,'ReadyPort',true);
...
[dataOut,validOut,ready] = rateConverter(dataIn,validIn);
```

Input Arguments

`dataIn` — Data input
scalar or row vector

Data input, specified as a scalar, or as a row vector where each element represents an independent channel.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `single` | `double`

`validIn` — Indicates valid input data
scalar

Control signal that indicates if the input data is valid. When `validIn` is `1` (`true`), the object captures the values from the `dataIn` argument. When `validIn` is `0` (`false`), the object ignores the values from the `dataIn` argument.

You can apply a valid data sample every `ceil(L/M)` calls to the object. You can use the optional `ready` output signal to indicate when the object can accept a new sample.

Data Types: `logical`

Output Arguments

`dataOut` — Resampled and filtered data sample

scalar or row vector

Resampled and filtered data sample, returned as a scalar, or as a vector in which each element represents an independent channel.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

Data Types: `fi` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `single` | `double`

`validOut` — Indicates valid output data

scalar

Control signal that indicates if the output data is valid. When `validOut` is `1` (`true`), the object returns valid data from the `dataOut` argument. When `validOut` is `0` (`false`), values from the `dataOut` argument are not valid.

Data Types: `logical`

`ready` — Indicates object is ready for new input data

scalar

Control signal that indicates that the object is ready for new input data sample on the next cycle. When `ready` is `1` (`true`), you can specify the `data` and `valid` inputs for the next time step. When `ready` is `0` (`false`), the object ignores any input data in the next time step.

Dependencies

To enable this argument, set the `ReadyPort` property to `true`.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Downsample Signal

Convert a signal from 48 kHz to 32 kHz by using the `dsphdl.FIRRateConverter` System object™.

Define the sample rate and length of the input signal, and a 2 kHz cosine waveform. Set `validIn = true` for every sample.

```
Fs = 48e3;
Ns = 100;
t = (0:Ns-1).'/Fs;
dataIn = cos(2*pi*2e3*t);
validIn = true(Ns,1);
```

Preallocate `dataOut` and `validOut` signals for faster simulation.

```
dataOut = zeros(Ns,1);
validOut = false(Ns,1);
```

Create the System object. Configure it to perform rate conversion by a factor of 2/3, using an equiripple filter.

```
Numerator = firpm(70,[0 0.25 0.32 1],[1 1 0 0]);
firrc = dsphdl.FIRRateConverter(2,3,Numerator);
```

Call the System object to perform the rate conversion and obtain each output sample.

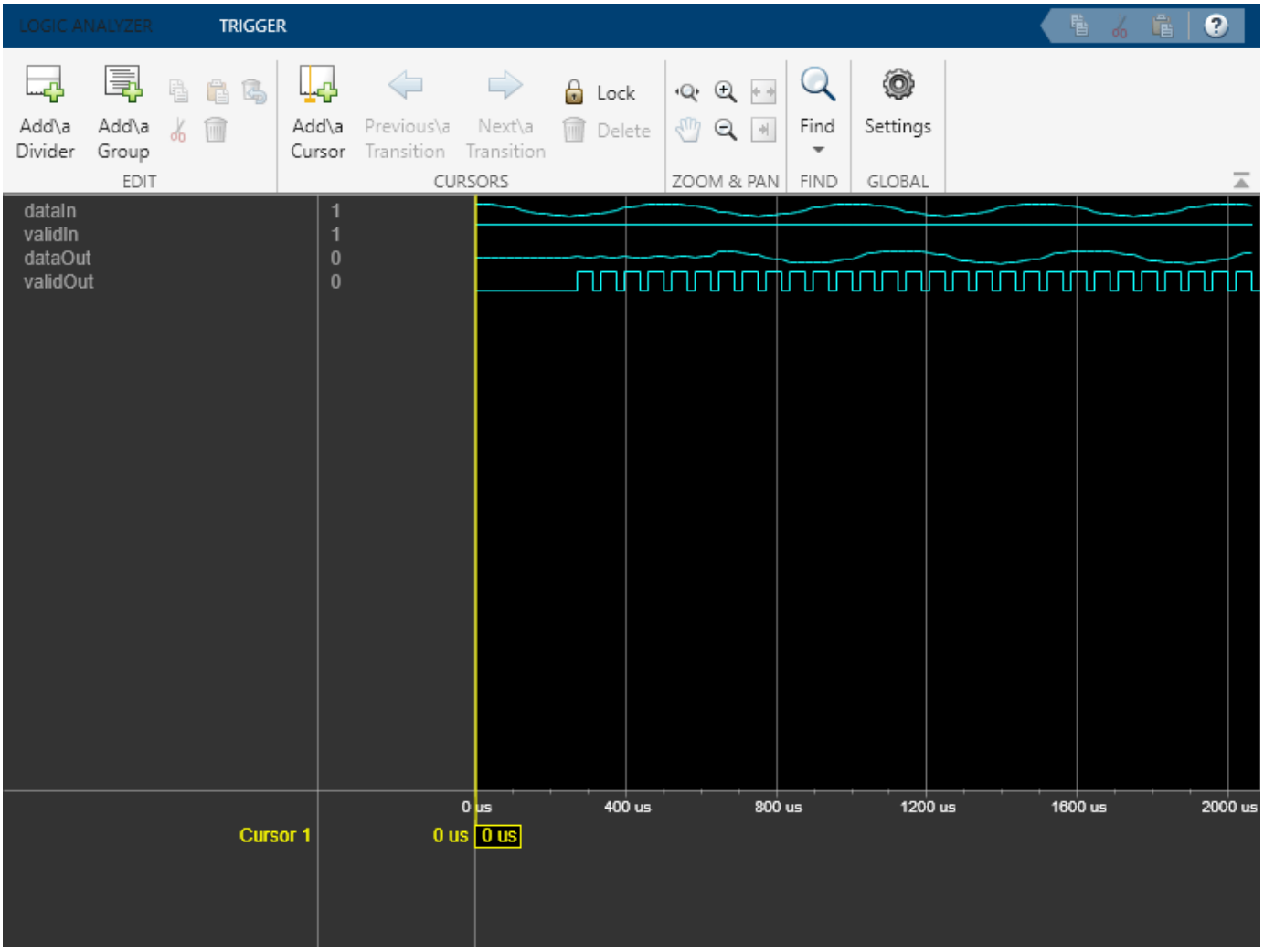
```
for k = 1:Ns
    [dataOut(k),validOut(k)] = firrc(dataIn(k),validIn(k));
end
```

Because the input sample rate is higher than the output sample rate, not every member of `dataOut` is valid. Use `validOut` to extract the valid samples from `dataOut`.

```
y = dataOut(validOut);
```

View the input and output signals with the Logic Analyzer.

```
la = dsp.LogicAnalyzer('NumInputPorts',4,'SampleTime',1/Fs,'TimeSpan',Ns/Fs);
modifyDisplayChannel(la,1,'Name','dataIn','Format','Analog','Height',8)
modifyDisplayChannel(la,2,'Name','validIn')
modifyDisplayChannel(la,3,'Name','dataOut','Format','Analog','Height',8)
modifyDisplayChannel(la,4,'Name','validOut')
la(dataIn,validIn,dataOut,validOut)
```



Upsample Signal

Convert a signal from 40 MHz to 100 MHz by using the `dsphdl.FIRRateConverter` System object™. To avoid overrunning the object as the signal is upsampled, control the input rate manually.

Define the sample rate and length of the input signal, and a fixed-point cosine waveform.

```
Fs = 40e6;
Ns = 50;
t = (0:Ns-1).'/Fs;
x = fi(cos(2*pi*1.2e6*t),1,16,14);
```

Define the rate conversion parameters. Use an interpolation factor of 5 and a decimation factor of 2. Calculate how often the object can accept a new input sample.

```
L = 5;
M = 2;
```

```
stepsPerInput = ceil(L/M);
numSteps = stepsPerInput*Ns;
```

Generate `dataIn` and `validIn` based on how often the object can accept a new sample.

```
dataIn = zeros(numSteps,1,'like',x);
dataIn(1:stepsPerInput:end) = x;
validIn = false(numSteps,1);
validIn(1:stepsPerInput:end) = true;
```

Create the System object. Configure it to perform rate conversion using the specified factors and an equiripple FIR filter.

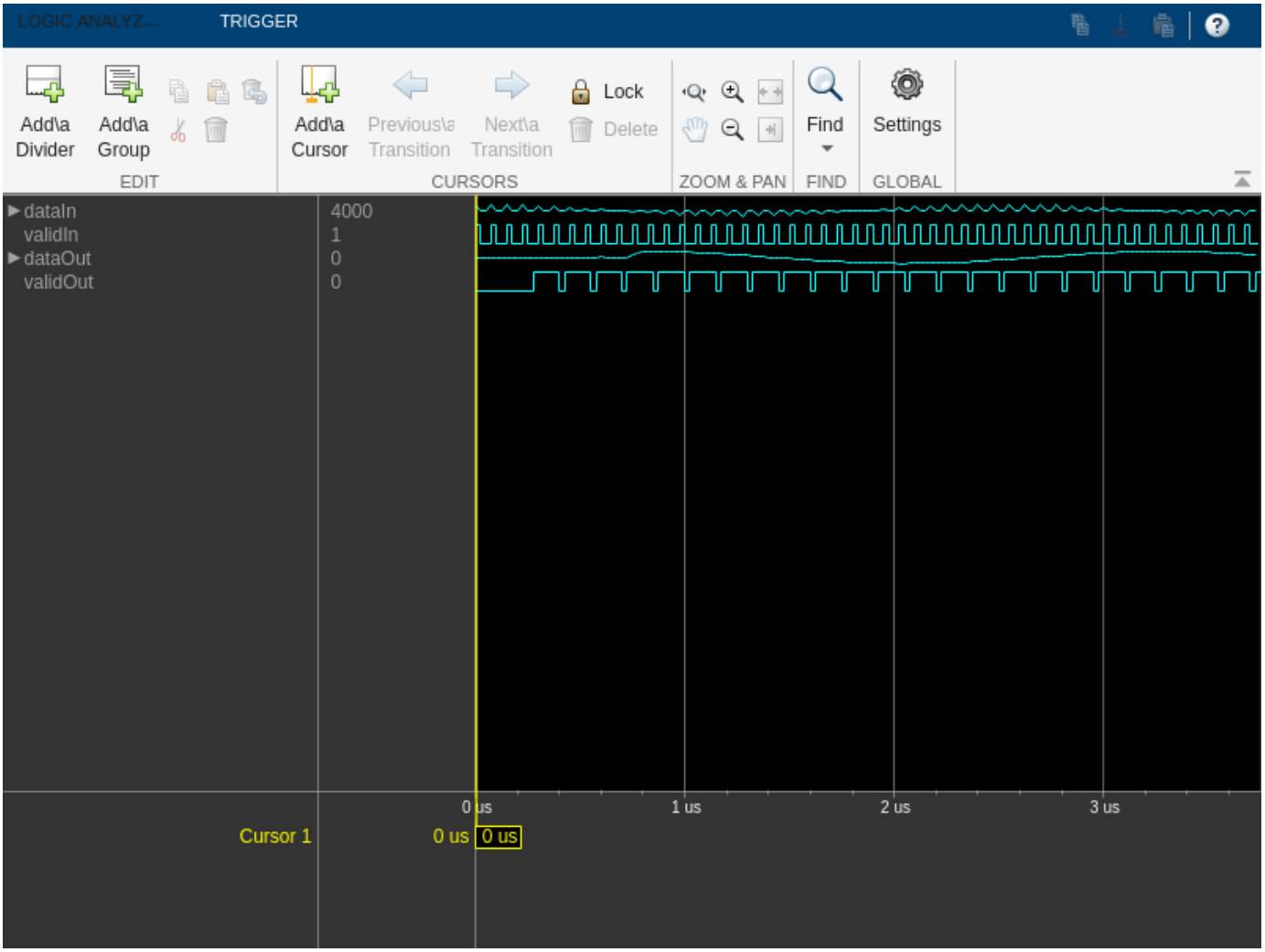
```
Numerator = firpm(70,[0 0.15 0.25 1],[1 1 0 0]);
rateConverter = dsphdl.FIRRateConverter(L,M,Numerator);
```

Create a Logic Analyzer to capture and view the input and output signals.

```
la = dsp.LogicAnalyzer('NumInputPorts',4,'SampleTime',1/Fs,'TimeSpan',numSteps/Fs);
modifyDisplayChannel(la,1,'Name','dataIn','Format','Analog','Height',8)
modifyDisplayChannel(la,2,'Name','validIn')
modifyDisplayChannel(la,3,'Name','dataOut','Format','Analog','Height',8)
modifyDisplayChannel(la,4,'Name','validOut')
```

Call the System object to perform the rate conversion and obtain each output sample. Call the Logic Analyzer to add each sample to the waveform display.

```
for k = 1:numSteps
    [dataOut,validOut] = rateConverter(dataIn(k),validIn(k));
    la(dataIn(k),validIn(k),dataOut,validOut)
end
```



Control Input Rate When Upsampling

Convert a signal from 40 MHz to 100 MHz by using the `dsphdl.FIRRateConverter` System object™. Use the optional `ready` output signal to avoid overrunning the object as the data is upsampled. The `ready` signal indicates the object can accept a new data sample on the next call to the object.

Define the sample rate and length of the input signal, and a fixed-point cosine waveform. Create a `SignalSource` object to provide data samples on demand.

```
Fs = 40e6;
Ns = 50;
t = (0:Ns-1).'/Fs;
x = fi(cos(2*pi*1.2e6*t),1,16,14);
inputSource = dsp.SignalSource(x);
```

Define the rate conversion parameters. Use an interpolation factor of 5 and a decimation factor of 2. Determine the number of calls to the object needed to convert `Ns` samples.

```
L = 5;
M = 2;
numSteps = floor(Ns*L/M);
```

Create the FIR rate converter System object. Configure it to perform rate conversion using the specified factors and an equiripple FIR filter. Enable the optional `ready` output port.

```
Numerator = firpm(70,[0 0.15 0.25 1],[1 1 0 0]);
rateConverter = dsphdl.FIRRateConverter(L,M,Numerator,'ReadyPort',true);
```

Create a Logic Analyzer to capture and view the input and output signals.

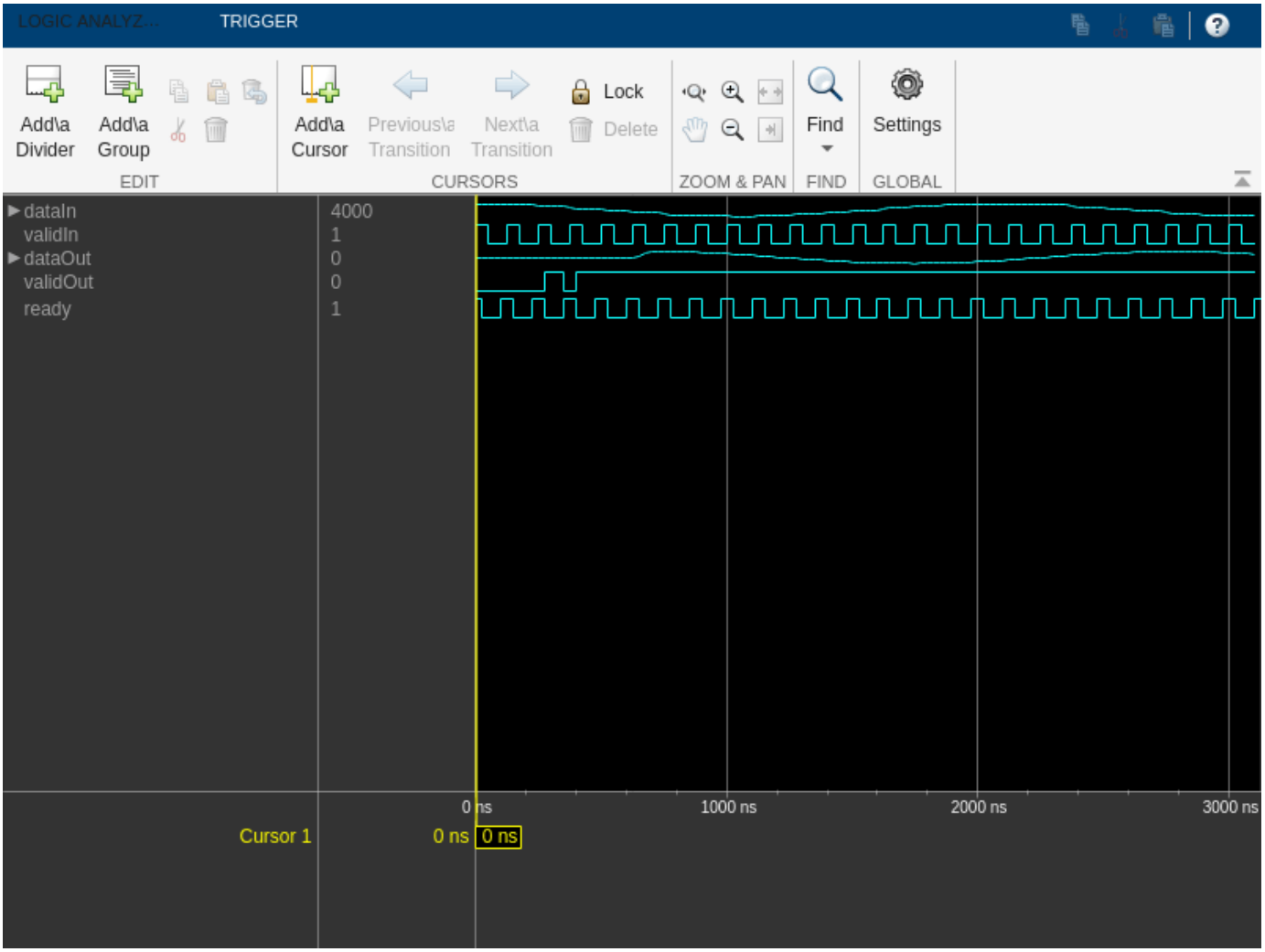
```
la = dsp.LogicAnalyzer('NumInputPorts',5,'SampleTime',1/Fs,'TimeSpan',numSteps/Fs);
modifyDisplayChannel(la,1,'Name','dataIn','Format','Analog','Height',8)
modifyDisplayChannel(la,2,'Name','validIn')
modifyDisplayChannel(la,3,'Name','dataOut','Format','Analog','Height',8)
modifyDisplayChannel(la,4,'Name','validOut')
modifyDisplayChannel(la,5,'Name','ready')
```

Initialize the `ready` signal. The object is always ready for input data on the first call.

```
ready = true;
```

Call the System object to perform the rate conversion and obtain each output sample. Apply a new input sample when the object indicates it is ready. Otherwise, set `validIn` to `false`.

```
for k = 1:numSteps
    if ready
        dataIn = inputSource();
    end
    validIn = ready;
    [dataOut,validOut,ready] = rateConverter(dataIn,validIn);
    la(dataIn,validIn,dataOut,validOut,ready)
end
```



Design for HDL Code Generation from FIR Rate Converter

Create a rate conversion function targeted for HDL code generation, and a test bench to exercise it. The function converts a signal from 40 MHz to 100 MHz. To avoid overrunning the object, the test bench manually controls the input rate.

Define the sample rate and length of the input signal, and a fixed-point cosine waveform.

```
Fs = 40e6;
Ns = 50;
t = (0:Ns-1).'/Fs;
x = fi(cos(2*pi*1.2e6*t), 1, 16, 14);
```

Define the rate conversion parameters. Use an interpolation factor of 5 and a decimation factor of 2. Calculate how often the object can accept a new data sample.

```
L = 5;
M = 2;
```

```
stepsPerInput = ceil(L/M);
numSteps = stepsPerInput*Ns;
```

Generate `dataIn` and `validIn` based on how often the object can accept a new sample.

```
dataIn = zeros(numSteps,1,'like',x);
dataIn(1:stepsPerInput:end) = x;
validIn = false(numSteps,1);
validIn(1:stepsPerInput:end) = true;
```

Create a Logic Analyzer to capture and view the input and output signals.

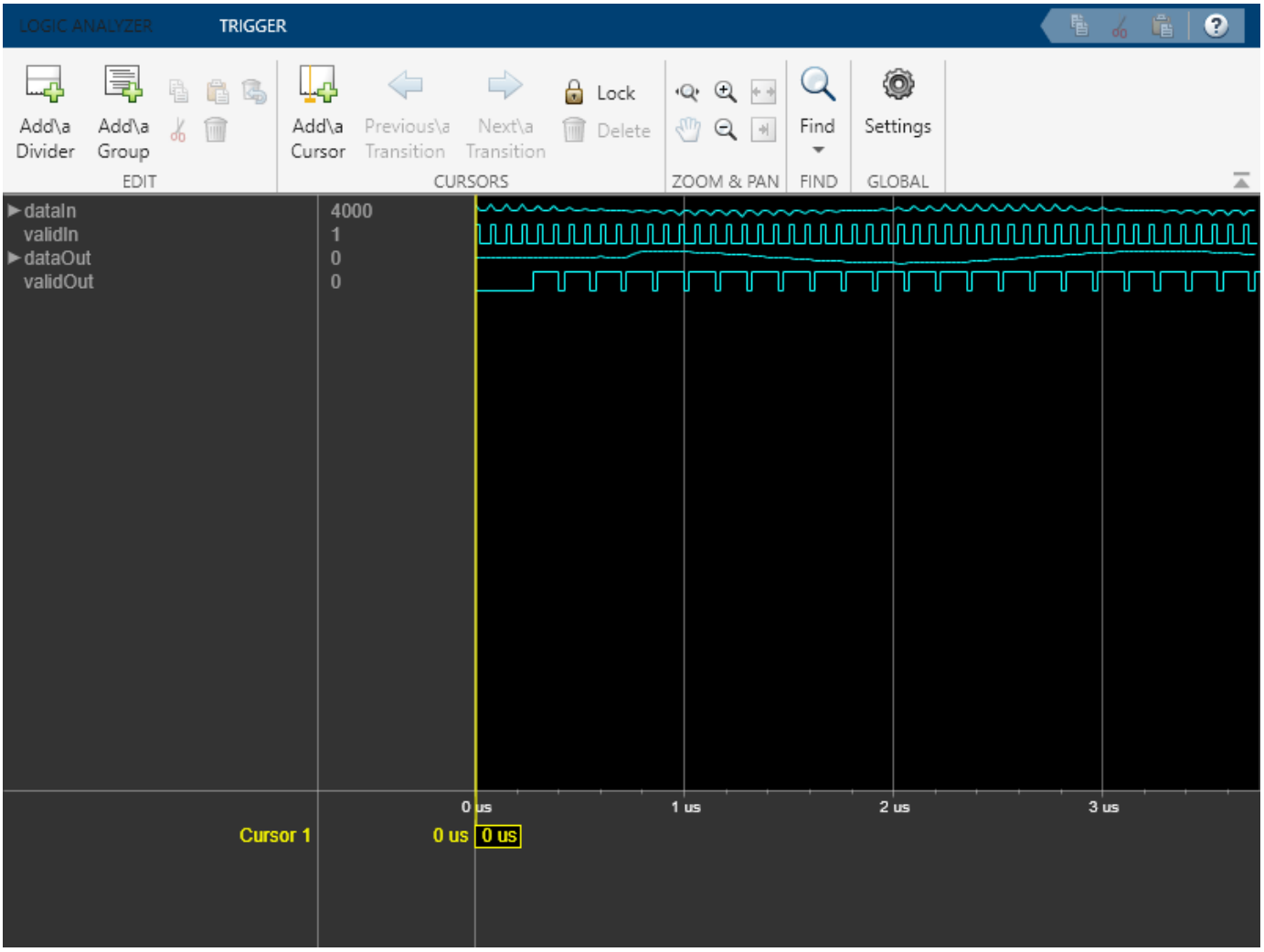
```
la = dsp.LogicAnalyzer('NumInputPorts',4,'SampleTime',1/Fs,'TimeSpan',numSteps/Fs);
modifyDisplayChannel(la,1,'Name','dataIn','Format','Analog','Height',8)
modifyDisplayChannel(la,2,'Name','validIn')
modifyDisplayChannel(la,3,'Name','dataOut','Format','Analog','Height',8)
modifyDisplayChannel(la,4,'Name','validOut')
```

Write a function that creates and calls the System object.

```
function [dataOut,validOut] = HDLFIRRC5_2(dataIn,validIn)
%HDLFIRRC5_2
% Processes one sample of data using the dsphdl.FIRRateConverter System
% object. dataIn is a fixed-point scalar value. validIn is a logical scalar value.
% You can generate HDL code from this function.

  persistent firrc5_2;
  if isempty(firrc5_2)
    Numerator = firpm(70,[0,.15,.25,1],[1,1,0,0]);
    firrc5_2 = dsphdl.FIRRateConverter(5,2,Numerator);
  end
  [dataOut,validOut] = firrc5_2(dataIn,validIn);
end
```

Resample the signal by calling the function for each data sample.

```
for k = 1:numSteps
    [dataOut,validOut] = HDLFIRRC5_2(dataIn(k),validIn(k));
    la(dataIn(k),validIn(k),dataOut,validOut)
end
```



Algorithms

This object implements the algorithms described on the FIR Rate Converter block reference page.

Flow Control

The object accepts and returns control signal arguments for pacing the flow of samples. By default, the object uses `validIn` and `validOut` control signals. You can also enable a `ready` output signal.

The `ready` output indicates that the object can accept a new input data sample on the next call to the object. When $L \geq M$, you can use the `ready` argument to achieve continuous output data samples. If you apply a new input sample after each time object returns `ready = true`, each call to the object returns a data output sample with `validOut = true`.

When you do not enable the `ready` argument, you can apply a valid data sample only every `ceil(L/M)` calls to the object. For example:

- $L/M = 4/5$ — You can apply a new input sample on every call.

- *L/M* = 3/2 — You can apply a new input sample on every other call.

Version History

Moved to DSP HDL Toolbox from DSP System Toolbox
Behavior changed in R2022a

Before R2022a, this object was named `dsp.HDLFIRRateConverter` and was part of the DSP System Toolbox product.

Remove request argument
Behavior changed in R2022a

In previous releases, the object provided an optional `request` argument. This argument is no longer available.

Synchronous ready signal
Behavior changed in R2022a

In previous releases, the `ready` output signal was direct feedthrough without an output pipeline register. This signal is now pipelined at the output of the object.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This System object supports C/C++ code generation for accelerating MATLAB simulations, and for DPI component generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

`double` and `single` data types are supported for simulation, but not for HDL code generation.

To generate HDL code from predefined System objects, see “HDL Code Generation from Viterbi Decoder System Object” (HDL Coder).

See Also

Blocks
FIR Rate Converter

Objects
`dsp.FIRRateConverter`

Introduced in R2015b

getLatency

Package: `dsphdl`

Latency of FIR filter

Syntax

```
Y = getLatency(hdlobj)
Y = getLatency(hdlobj,inputType,[],isInputComplex,V)
Y = getLatency(hdlobj,coeffType,coeffPrototype,isInputComplex,V)
```

Description

`Y = getLatency(hdlobj)` returns the latency, `Y`, between the first valid input sample and the first valid output sample, assuming contiguous input samples.. Use this syntax when the `CoefficientsDataType` is set to a numeric type, you are not using programmable coefficients, and the input data is not complex or a vector.

`Y = getLatency(hdlobj,inputType,[],isInputComplex,V)` returns the latency, `Y`. The latency depends on filter structure, filter coefficients, and input vector size. Use this syntax when you are not using programmable coefficients. The these arguments may be optional, depending on the object configuration.

- Use `inputType` when you set `CoefficientsDataType` property to `'Same word length as input'`. The latency can change with input data type because the object casts the coefficients to the input data type, which can affect multiplier sharing for equal-absolute-value coefficients.
- Use `isInputComplex` when your input data is complex and you are using a partly-serial systolic architecture. The latency changes when you have complex data and complex coefficients because of the extra adder pipeline. When you specify `isInputComplex`, you must also give a placeholder argument, `[]` for the unused third argument.
- Use `V` to specify the input vector size when your input is not scalar.

`Y = getLatency(hdlobj,coeffType,coeffPrototype,isInputComplex,V)` returns the latency, `Y`. Use this syntax when you are using programmable coefficients. `coeffType` is the data type of the input coefficients. The final two arguments may be optional, depending on the object configuration.

- Use `coeffPrototype` to optimize the programmable filter for symmetric or antisymmetric coefficients. The prototype specifies a pattern that all input coefficients must follow. Based on the prototype, the object implements an optimized filter that shares the multipliers for symmetric coefficients. If your input coefficients do not all conform to the same pattern, or to opt out of multiplier optimization, you can omit this argument or specify the prototype as an empty vector, `[]`.
- Use `isInputComplex` when your input data is complex. When you specify `isInputComplex`, you must also specify the `coeffPrototype` or a placeholder argument, `[]`.
- Use `V` to specify the input vector size when your input is not scalar.

Examples

Explore Latency of FIR Object

The latency of the `dsphdl.FIRFilter` System object™ varies with filter structure, serialization options, input vector size, and whether the coefficient values provide optimization opportunities. Use the `getLatency` function to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output.

Create a `dsphdl.FIRFilter` System object™ and request the latency. The default architecture is fully parallel systolic. The default data type for the coefficients is `'Same word length as input'`. Therefore, when you call the `getLatency` object function, you must specify an input data type. The object casts the coefficient values to the input data type, and then checks for symmetric coefficients. This `Numerator` has 31 symmetric coefficients, so the object optimizes for the shared coefficients, and implements 16 multipliers.

```
Numerator = firpm(30,[0 0.1 0.2 0.5]*2,[1 1 0 0]);
Input_type = numerictype(1,16,15); % object uses only the word length for coefficient type cast
hdlfir = dsphdl.FIRFilter('Numerator',Numerator);
L_sysp = getLatency(hdlfir,Input_type)
```

```
L_sysp = 23
```

For the same fully parallel filter with vector input, the latency is lower. Call `getLatency` with an input vector size of four to check the latency for that case. The empty arguments are placeholders for when you use programmable coefficients or complex input data.

```
L_syspv = getLatency(hdlfir,Input_type,[],[],4)
```

```
L_syspv = 17
```

Check the latency for a partly serial systolic implementation of the same filter. By default, the `SerializationOption` property is `'Minimum number of cycles between valid input samples'`, and so you must specify the serialization rule using the `NumCycles` property. To share each multiplier between 8 coefficients, set the `NumCycles` to 8. The object then optimizes based on the coefficient symmetry, so there are 16 unique coefficients shared 8 times each over 2 multipliers. This serial filter implementation requires input samples that are valid every 8 cycles.

```
hdlfir = dsphdl.FIRFilter('Numerator',Numerator,'FilterStructure','Partly serial systolic','NumCy
L_syss = getLatency(hdlfir,Input_type)
```

```
L_syss = 19
```

Check the latency of a nonsymmetric fully parallel systolic filter. The `Numerator` has 31 coefficients.

```
Numerator = sinc(0.4*[-30:0]);
hdlfir = dsphdl.FIRFilter('Numerator',Numerator);
L_sysp = getLatency(hdlfir,Input_type)
```

```
L_sysp = 37
```

Check the latency of the same nonsymmetric filter implemented as a partly serial systolic filter. In this case, specify the `SerializationOption` by the number of multipliers. The object implements a filter that has 2 multipliers and requires 8 cycles between input samples.

```
hdlfir = dsphdl.FIRFilter('Numerator',Numerator,'FilterStructure','Partly serial systolic',...
                          'SerializationOption','Maximum number of multipliers','NumberOfMultipl
L_syss = getLatency(hdlfir,Input_type)
```

```
L_syss = 37
```

Check the latency of a fully parallel transposed architecture. The latency for this filter structure with scalar input is always 6 cycles.

```
hdlfir = dsphdl.FIRFilter('Numerator',Numerator,'FilterStructure','Direct form transposed');
L_trans = getLatency(hdlfir,Input_type)
```

```
L_trans = 6
```

The latency of the transposed filter increases with input vector size.

```
L_transv4 = getLatency(hdlfir,Input_type,[],[],4)
```

```
L_transv4 = 9
```

```
L_transv8 = getLatency(hdlfir,Input_type,[],[],16)
```

```
L_transv8 = 11
```

Input Arguments

`hdlobj` — HDL-optimized filter System object

`dsphdl.FIRFilter`

HDL-optimized filter System object that you created and configured.

`inputType` — Input data type

`numerictype` object

Input data type, specified as a `numerictype` object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

If you specify `[]` for this argument, the object uses `double` data type to calculate the latency. The result is equivalent to the fixed-point latency as long as the coefficient data type is large enough to represent the coefficient values exactly.

Dependencies

This argument applies when the `CoefficientsDataType` is `'Same word length as input'`.

`coeffType` — Input coefficients data type

`numerictype` object

Input coefficients data type, specified as a `numerictype` object. This argument applies when tyou use programmable coefficients. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

Dependencies

This argument applies when you set `NumeratorSource` to `'Input port (Parallel interface)'`.

`coeffPrototype` — Prototype filter coefficients

`[]` (default) | vector of numeric real values

Prototype filter coefficients, specified as a vector of numeric real values. The prototype specifies a pattern that all input coefficients must follow. Based on the prototype, the object implements an optimized filter that shares the multipliers for symmetric coefficients. If your input coefficients do not all conform to the same pattern, or to opt out of multiplier optimization, specify the prototype as an empty vector, `[]`.

Coefficient optimizations affect the latency of the filter object.

Dependencies

This argument applies when you set `NumeratorSource` to `'Input port (Parallel interface)'`. When you have complex input data, but are not using programmable coefficients, set this argument to `[]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`isInputComplex` — Complexity of input data

`false` (default) | `true`

Set this argument to `true` if your input data is complex. You can omit this argument if your input data is real. When your filter has complex input data and complex coefficients there is an additional adder at the output of the filter that adds pipeline latency.

Data Types: `logical`

`V` — Vector size

power of 2 from 1 to 64

Vector size, specified as a power of 2 from 1 to 64. Use this argument to request the latency of an object similar to `hdlobj`, but with `V`-sample vector input. When you do not specify this argument, the function assumes scalar input.

Output Arguments

`Y` — Cycles of latency

integer

Cycles of latency that the filter object takes between the first valid input and the first valid output. Each call to the object simulates one cycle. This latency assumes valid input data on every cycle.

See Also

Objects

`dsphdl.FIRFilter`

Introduced in R2017a

getLatency

Package: `dsphdl`

Latency of FFT calculation

Syntax

```
Y = getLatency(hdlfft)
Y = getLatency(hdlfft,N)
Y = getLatency(hdlfft,N,V)
```

Description

`Y = getLatency(hdlfft)` returns the number of cycles, `Y`, that the object takes to calculate the FFT of an input frame. The latency depends on the input vector size and the FFT length.

`Y = getLatency(hdlfft,N)` returns the number of cycles that an object would take to calculate the FFT of an input frame, if it had FFT length `N`, and scalar input. This function does not change the properties of the `hdlfft`.

`Y = getLatency(hdlfft,N,V)` returns the number of cycles that an object would take to calculate the FFT of an input frame, if it had FFT length `N`, and vector input of size `V`. This function does not change the properties of `hdlfft`.

Examples

Explore Latency of HDL FFT Object

The latency of the object varies with the FFT length and the vector size. Use the `getLatency` function to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous.

Create a new `dsphdl.FFT` object and request the latency.

```
hdlfft = dsphdl.FFT('FFTLength',512);
L512 = getLatency(hdlfft)
```

```
L512 = 599
```

Request hypothetical latency information about a similar object with a different FFT length. The properties of the original object do not change.

```
L256 = getLatency(hdlfft,256)
```

```
L256 = 329
```

```
N = hdlfft.FFTLength
```

```
N = 512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(hdlfft,256,8)
```

```
L256v8 = 93
```

Enable scaling at each stage of the FFT. The latency does not change.

```
hdlfft.Normalize = true;
L512n = getLatency(hdlfft)
```

```
L512n = 599
```

Request the same output order as the input order. The latency increases because the object must collect the output before reordering.

```
hdlfft.BitReversedOutput = false;
L512r = getLatency(hdlfft)
```

```
L512r = 1078
```

Input Arguments

`hdlfft` — FFT System object

`dsphdl.FFT` | `dsphdl.IFFT`

FFT System object. See `dsphdl.IFFT` or `dsphdl.FFT`.

`N` — FFT length

integer power of 2 from 2^2 to 2^{16}

FFT length, specified as an integer power of 2 from 2^2 to 2^{16}. Use this argument to request the latency of an object similar to `hdlfft`, but with FFT length `N`.

`V` — Vector size

power of 2 from 1 to 64

Vector size, specified as a power of 2 from 1 to 64. The vector size cannot be greater than the FFT length. Use this argument to request the latency of an object similar to `hdlfft`, but with `V`-sample vector input. When you do not specify this argument, the function assumes scalar input.

Output Arguments

`Y` — Cycles of latency

integer

Cycles of latency that the object takes to calculate the FFT of an input frame, returned as an integer. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. Each call to the object simulates one cycle.

See Also

Objects

`dsphdl.Channelizer` | `dsphdl.ChannelSynthesizer` | `dsphdl.IFFT` | `dsphdl.FFT`

Introduced in R2014a

getLatency

Package: dsphdl

Latency of CIC decimation filter

Syntax

```
Y = getLatency(hdlcic)
Y = getLatency(hdlcic,V)
```

Description

`Y = getLatency(hdlcic)` returns the latency, `Y`, between the first valid input sample and the first valid output sample, assuming contiguous input samples. The latency depends on the `NumSections` and `GainCorrection` properties.

`Y = getLatency(hdlcic,V)` returns the latency, `Y`, between the first valid input sample and the first valid output sample, assuming contiguous input samples and vector input of size `V`. The latency depends on the vector input size, `NumSections` property, and the `GainCorrection` property.

Examples

Explore Latency of CIC Decimator Object

The latency of the `dsphdl.CICDecimator` System object™ varies depending on how many integrator and comb sections your filter has, the input vector size, and whether you enable gain correction. Use the `getLatency` function to find the latency of a particular filter configuration. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is continuously valid.

Create a `dsphdl.CICDecimator` System object and request the latency. The default System object filter has two integrator and comb sections, and the gain correction is disabled.

```
hdlcic = dsphdl.CICDecimator
```

```
hdlcic =
  dsphdl.CICDecimator with properties:

     DecimationSource: 'Property'
     DecimationFactor: 2
    DifferentialDelay: 1
          NumSections: 2
       GainCorrection: false

  Show all properties
```

```
L_def = getLatency(hdlcic)
```

```
L_def = 5
```

Modify the filter object so it has three integrator and comb sections. Check the resulting change in latency.

```
hdlcic.NumSections = 3;
L_3sec = getLatency(hdlcic)
```

```
L_3sec = 6
```

Enable the gain correction on the filter object with vector input size 2. Check the resulting change in latency.

```
hdlcic.GainCorrection = true;
vecSize = 2;
L_wgain = getLatency(hdlcic,vecSize)
```

```
L_wgain = 25
```

Input Arguments

`hdlcic` — CIC decimation filter System object

`dsphdl.CICDecimator`

CIC decimation filter System object that you created and configured. See `dsphdl.CICDecimator`.

`V` — Vector size

in range from 1 to 64

Vector size, specified in the range from 1 to 64. `DecimationFactor` property must be an integer multiple of the input frame size. Use this argument to request the latency of an object similar to `hdlcic`, but with `V` sample vector input. When you do not specify this argument, the function assumes scalar input.

Output Arguments

`Y` — Cycles of latency

integer

Cycles of latency that the CIC decimator object takes between the first valid input and the first valid output. Each call to the object simulates one cycle. This latency assumes valid input data on every cycle.

See Also

Objects

`dsphdl.CICDecimator`

Introduced in R2019b

getLatency

Package: `dsphdl`

Latency of CIC interpolation filter

Syntax

```
Y = getLatency(hdlcic)
Y = getLatency(hdlcic,V)
```

Description

`Y = getLatency(hdlcic)` returns the latency, `Y`, between the first valid input sample and the first valid output sample, assuming contiguous input samples. The latency depends on the `NumSections` property and whether `GainCorrection` is enabled.

`Y = getLatency(hdlcic,V)` returns the latency, `Y`, between the first valid input sample and the first valid output sample, assuming contiguous input samples and vector input of size `V`. The latency depends on the `NumSections` property, vector input size, and whether `GainCorrection` is enabled.

Examples

Input Arguments

`hdlcic` — CIC interpolation filter System object

`dsphdl.CICInterpolator`

CIC interpolation filter System object that you created and configured. See `dsphdl.CICInterpolator`.

`V` — Vector size

in range from 1 to 64

Vector size, specified in the range from 1 to 64. Use this argument to request the latency of an object similar to `hdlcic`, but with `V` sample vector input. When you do not specify this argument, the function assumes scalar input.

Output Arguments

`Y` — Cycles of latency

integer

Cycles of latency that the CIC interpolator object takes between the first valid input and the first valid output. Each call to the object simulates one cycle. This latency assumes valid input data on every cycle.

See Also

Objects
`dsphdl.CICInterpolator`

Introduced in R2022a

getLatency

Package: `dsphdl`

Latency of FIR decimation filter

Syntax

```
Y = getLatency(hdlfird,inputType,isInputComplex,inputVecSize)
Y = getLatency(hdlfird)
```

Description

`Y = getLatency(hdlfird,inputType,isInputComplex,inputVecSize)` returns the latency, `Y`, between the first valid input sample and the first valid output sample, assuming contiguous input samples. The latency depends on filter structure and filter coefficients. The final two arguments may be optional, depending on the object configuration.

- Use `inputType` when you set `CoefficientsDataType` property to `'Same word length as input'`. Otherwise, set it to `[]`.
- Set `isInputComplex` to `true` when your input data is complex. The latency changes when you have complex data and complex coefficients, because of the extra adder pipeline.

`Y = getLatency(hdlfird)` returns the latency, `Y`. Use this syntax when the `CoefficientsDataType` is set to a numeric type, you are using scalar input, and the input data is not complex.

Examples

Explore Latency of FIR Decimator Object

The latency of the `dsphdl.FIRDecimator` System object™ varies with filter architecture and input vector size. Use the `getLatency` function to find the latency of a particular configuration. The latency is the number of cycles between the first valid input and the first valid output.

Create a `dsphdl.FIRDecimator` System object™ and request the latency. The default filter is a direct-form systolic architecture. The default data type for the coefficients is `'Same word length as input'`. Therefore, when you call the `getLatency` object function, you must specify an input data type. The default filter has 36 coefficients. This example assumes the data input to your filter is complex-valued. The default coefficients are real-valued. Complexity affects filter latency only when you have complex-valued data and complex-valued coefficients.

```
inputType = numerictype(1,16,15); % object uses only the word length for coefficient type cast
complexInput = true;
downBy4 = dsphdl.FIRDecimator('DecimationFactor',4);
L_by4scalar = getLatency(downBy4,inputType,complexInput)
```

```
L_by4scalar = 44
```

Check the latency for the same filter with vector input.

```
vectorSize = 2;
L_by4Vec2 = getLatency(downBy4,inputType,complexInput,vectorSize)
```

```
L_by4Vec2 = 28
```

Check the latency of a transposed architecture.

```
downBy4.FilterStructure = 'Direct form transposed';
L_by4trans = getLatency(downBy4,inputType,complexInput)
```

```
L_by4trans = 11
```

Check the latency for the transposed filter with vector input.

```
vectorSize = 4;
L_by4transVec4 = getLatency(downBy4,inputType,complexInput,vectorSize)
```

```
L_by4transVec4 = 9
```

Input Arguments

`hdlfird` — HDL-optimized FIR filter System object

`dsphdl.FIRDecimator`

HDL-optimized FIR decimation filter System object that you created and configured. See `dsphdl.FIRDecimator`.

`inputType` — Input data type

`numerictype` object

Input data type, specified as a `numerictype` object. Call `numerictype(s,w,f)`, where `s` is 1 for signed and 0 for unsigned, `w` is the word length in bits, and `f` is the number of fractional bits.

If you specify `[]` for this argument, the object uses `double` data type to calculate the latency. The result is equivalent to the fixed-point latency as long as the coefficient data type is large enough to represent the coefficient values exactly.

Dependencies

This argument applies when the `CoefficientsDataType` is `'Same word length as input'`.

`isInputComplex` — Complexity of input data

`false` (default) | `true`

Set this argument to `true` if your input data is complex. You can omit this argument if your input data is real. When your filter has complex input data and complex coefficients there is an additional adder at the output of the filter that adds pipeline latency.

Data Types: `logical`

`inputVecSize` — Vector size

integer from 1 to 64

Vector size, specified as an integer from 1 to 64. When you do not specify this argument, the function assumes scalar input.

Output Arguments

Y — Cycles of latency

integer

Cycles of latency that the filter object takes between the first valid input and the first valid output. Each call to the object simulates one cycle. This latency assumes valid input data on every cycle.

See Also

Objects

`dsphdl.FIRDecimator`

Introduced in R2020b

getLatency

Package: `dsphdl`

Latency of channelizer calculation

Syntax

```
Y = getLatency(channelizer)
Y = getLatency(channelizer,N)
Y = getLatency(channelizer,N,V)
Y = getLatency(channelizer,N,V,isInputType)
```

Description

`Y = getLatency(channelizer)` returns the number of cycles, `Y`, that the object takes to channelize an input frame. The latency depends on the input vector size and the FFT length. The channelizer filter coefficients does not affect the latency.

`Y = getLatency(channelizer,N)` returns the number of cycles that an object would take to channelize an input frame, if it had FFT length `N`, and scalar input. This function does not change the properties of the `channelizer`.

`Y = getLatency(channelizer,N,V)` returns the number of cycles that an object would take to channelize an input frame, if it had FFT length `N`, and vector input of size `V`. This function does not change the properties of `channelizer`.

`Y = getLatency(channelizer,N,V,isInputType)` returns the number of cycles that an object would take to channelize an input frame, if it had FFT length `N`, vector input of size `V`, and `isInputType` that indicates the complexity of the input when the filter coefficients of channelizer are complex. This function does not change the properties of `channelizer`.

Examples

Explore Latency of Channelizer Object

The latency of the `dsphdl.Channelizer` object varies with the FFT length, filter structure, vector size, and input type. Use the `getLatency` function to find the latency of a particular configuration. The latency is measured as the number of cycles between the first valid input and the first valid output, assuming that the input is contiguous. The number of filter coefficients does not affect the latency. Setting the output size equal to the input size reduces the latency because the samples are not saved and reordered.

Create a `dsphdl.Channelizer` object with filter structure set to direct form transposed and request the latency.

```
channelize = dsphdl.Channelizer('NumFrequencyBands',512, 'FilterStructure','Direct form transpose
L512 = getLatency(channelize)
```

```
L512 = 1118
```

Request hypothetical latency information about a similar object with a different number of frequency bands (FFT length). The properties of the original object do not change.

```
L256 = getLatency(channelize,256)
```

```
L256 = 592
```

```
N = channelize.NumFrequencyBands
```

```
N = 512
```

Request hypothetical latency information of a similar object that accepts eight-sample vector input.

```
L256v8 = getLatency(channelize,256,8)
```

```
L256v8 = 132
```

Enable scaling at each stage of the FFT. The latency does not change.

```
channelize.Normalize = true;
L512n = getLatency(channelize)
```

```
L512n = 1118
```

Request the same output size and order as the input data. The latency decreases because the object does not need to store and reorder the data before output. The default input size is scalar.

```
channelize.OutputSize = 'Same as input size';
L512r = getLatency(channelize)
```

```
L512r = 1084
```

Check the latency of a vector input implementation where the input and output are the same size. Specify the current value of the FFT length and a vector size of 8 samples. The latency decreases because the object computes results in parallel when the input is a vector.

```
L512rv8 = getLatency(channelize,channelize.NumFrequencyBands,8)
```

```
L512rv8 = 218
```

Check the latency of a vector input implementation where the input type is complex. Specify the current value of the FFT length and a vector size of 16 samples.

```
L512rv16i = getLatency(channelize,channelize.NumFrequencyBands,16,true)
```

```
L512rv16i = 152
```

Input Arguments

`channelizer` — Channelizer System object

`dsphdl.Channelizer`

Channelizer System object that you created and configured. See `dsphdl.Channelizer`.

`N` — Number of frequency bands (FFT length)

integer power of 2 from 2^2 to 2^{16}

Number of frequency bands (FFT length), specified as an integer power of 2 from 2^2 to 2^{16}.

`V` — Vector size
power of 2 from 1 to 64

Vector size, specified as a power of 2 from 1 to 64. The vector size cannot be greater than the FFT length. When you do not specify this argument, the function assumes scalar input.

`isInputType` — Indicator of input data complexity
`logical` scalar

Indicator of input data whether it is complex or not.

Output Arguments

`Y` — Cycles of latency
integer

Cycles of latency that the object takes to channelize an input frame, returned as an integer. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. Each call to the object simulates one cycle.

See Also

Objects
`dsphdl.Channelizer` | `dsphdl.ChannelSynthesizer` | `dsphdl.IFFT` | `dsphdl.FFT`

Introduced in R2017a

getLatency

Package: `dsphdl`

Latency of channel synthesizer calculation

Syntax

```
Y = getLatency(channelsynthesizer)
Y = getLatency(channelsynthesizer,N)
```

Description

`Y = getLatency(channelsynthesizer)` returns the number of cycles, `Y` that the object would take to synthesize an input frame. The filter coefficients do not affect the latency. This function does not change the properties of the `channelsynthesizer`.

`Y = getLatency(channelsynthesizer,N)` returns the number of cycles that an object would take to synthesize an input frame. The latency depends on the filter structure and the IFFT length (`N`). The filter coefficients do not affect the latency. This function does not change the properties of the `channelsynthesizer`.

Examples

Explore Latency of Channel Synthesizer Object

The latency of the `dsphdl.ChannelSynthesizer` object varies with the IFFT length and filter structure.

Create a `dsphdl.ChannelSynthesizer` object with a direct form transposed filter structure and 16 frequency bands, and then calculate the latency.

```
synthesizerDT = dsphdl.ChannelSynthesizer('FilterStructure','Direct form transposed');
latencyDT = getLatency(synthesizerDT,16)
```

```
latencyDT = 20
```

Calculate the latency information for `dsphdl.ChannelSynthesizer` object with a direct form systolic filter structure and and 8 frequency bands.

```
synthesizerDS = dsphdl.ChannelSynthesizer('FilterStructure','Direct form systolic');
latencyDS = getLatency(synthesizerDS,8)
```

```
latencyDS = 21
```

Enable scaling at each stage of the IFFT. The latency does not change.

```
synthesizerDT.Normalize = true;
latencyDTn = getLatency(synthesizerDT,16)
```

```
latencyDTn = 20
```

Input Arguments

`channelsynthesizer` — Channel synthesizer System object

`dsphdl.ChannelSynthesizer`

Channel synthesizer System object that you created and configured. See `dsphdl.ChannelSynthesizer`.

`N` — Number of frequency bands (IFFT length)

integer power of 2 from 2^2 to 2^6

Number of frequency bands (IFFT length), specified as an integer power of 2 from 2^2 to 2^6.

Output Arguments

`Y` — Cycles of latency

integer

Cycles of latency that the object takes to synthesize the input frame, returned as an integer. The latency is the number of cycles between the first valid input and the first valid output, assuming the input is contiguous. Each call to the object simulates one cycle.

See Also

Objects

`dsphdl.Channelizer` | `dsphdl.ChannelSynthesizer` | `dsphdl.IFFT` | `dsphdl.FFT`

Introduced in R2022a