Vision HDL Toolbox™

Reference

MATLAB®

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

Revision History

Contents

Blocks

Bilateral Filter

2-D bilateral filtering
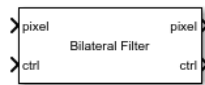
Libraries:
Vision HDL Toolbox / Filtering

Description

The Bilateral Filter block filters images while preserving edges. Some applications of bilateral filtering are denoising while preserving edges, separating texture from illumination, and cartooning to enhance edges. The filter replaces each pixel at the center of a neighborhood by an average that is calculated using spatial and intensity Gaussian filters. The block determines the filter coefficients from:

- Spatial location in the neighborhood (similar to a Gaussian blur filter)
- Intensity difference from the neighborhood center value

The block provides two standard deviation parameters for independent control of the spatial and intensity coefficients.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar intensity value. For multipixel streaming, specify a vector of two, four, or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

This block does not support multicomponent streaming. To process multicomponent streams, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect a single bus to multiple replicated blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: bus

Output

pixel — Output pixel stream
scalar | vector

Output pixel stream, returned as a scalar value representing intensity, or as a vector of two, four, or eight pixel intensity values. The dimensions and data type of the output **pixel** port match the dimensions and data type of the input **pixel** port.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: uint | int | fixed point | double | single

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: bus

Parameters

Main

Neighborhood size — Size of image region to average

3×3 (default) | 5×5 | 7×7 | 9×9 | 11×11 | 13×13 | 15×15

Size of the image region used to compute the average, specified as an N-by-N pixel square.

Spatial standard deviation — Spatial standard deviation target

0.5 (default) | positive real number

Spatial standard deviation target used to compute coefficients for the spatial Gaussian filter, specified as a positive real number. This parameter has no limits, but recommended values are from 0.1 to 10.

At the high end, the distribution becomes flat and the coefficients are small. At the low end, the distribution peaks in the center and has small coefficients in the rest of the neighborhood. These boundary values also depend on the neighborhood size and the data type used for the coefficients.

Intensity standard deviation — Intensity standard deviation target

0.5 (default) | positive real number

Intensity standard deviation target used to compute coefficients for the intensity Gaussian filter, specified as a positive real number. This parameter has no limits, but recommended values are from 0.1 to 10. At the high end, the distribution becomes flat and the coefficients are small. At the low end, the distribution peaks in the center and has small coefficients in the rest of the neighborhood. These boundary values also depend on the neighborhood size and the data type used for the coefficients.

When the intensity standard deviation is large, the bilateral filter acts more like a Gaussian blur filter, because the intensity Gaussian has a lower peak. Conversely, when the intensity standard deviation is smaller, edges in the intensity are preserved or enhanced.

Padding method — Method for padding boundary of input image

Constant (default) | Replicate | Symmetric | Reflection | None

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- Constant — Interpret pixels outside the image frame as having a constant value.
- Replicate — Repeat the value of pixels at the edge of the image.
- Symmetric — Set the value of the padding pixels to mirror the edge of the image.
- Reflection — Set the value of the padding pixels to reflect around the pixel at the edge of the image.
- None — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

Padding value — Value used to pad boundary of input image

0 (default) | integer

Specify an integer to pad the boundary of the input image. The block casts this value to the same data type as the input pixel.

Dependencies

To enable this parameter, set the **Padding method** parameter to Constant.

Line buffer size — Size of line memory buffer

2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

If you specify a value that is not a power of two, the block uses the next largest power of two.

Data Types

Rounding mode — Rounding method for internal fixed-point calculations

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

Specify a rounding method for internal fixed-point calculations.

Saturate on integer overflow — Overflow mode for internal fixed-point calculations

on (default) | off

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. By default, fixed-point values saturate on overflow. This option does not apply when the input data type is single or double.

Coefficients — Method to determine data type of filter coefficients

Inherit: Same as first input (default) | fixdt(1,16,0) | data type expression

Specify an unsigned data type that can represent values less than 1. The coefficients usually require a data type with more precision than the input data type. The block calculates the coefficients based on the neighborhood size and the values of **Intensity standard deviation** and **Spatial standard deviation**. Larger neighborhoods spread the Gaussian function such that each coefficient value is smaller. A larger standard deviation flattens the Gaussian so that the coefficients are more uniform in nature, and a smaller standard deviation produces a peaked response.

Note If you try a data type and after quantization, more than half of the coefficients become zero, the block issues a warning. If all the coefficients are zero after quantization, the block issues an error. These messages mean that the block was unable to express the requested filter by using the data type specified. To avoid this issue, choose a higher-precision coefficient data type or adjust the standard deviation targets.

Output — Method to determine data type of output pixels

Inherit: Same as first input (default) | fixdt(1,16,0) | data type expression

The filtered pixel values are cast to this data type.

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation

results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

The bilateral filter can be described as a Gaussian filter in the spatial dimension that modifies the coefficients of a second Gaussian filter that operates on intensity.

The algorithm stores N-1 lines so that it can form an N-by-N matrix of pixels matching the **Neighborhood size**. Then it applies two Gaussian filters on each neighborhood. The filter coefficients are calculated from the spatial and intensity standard deviations.
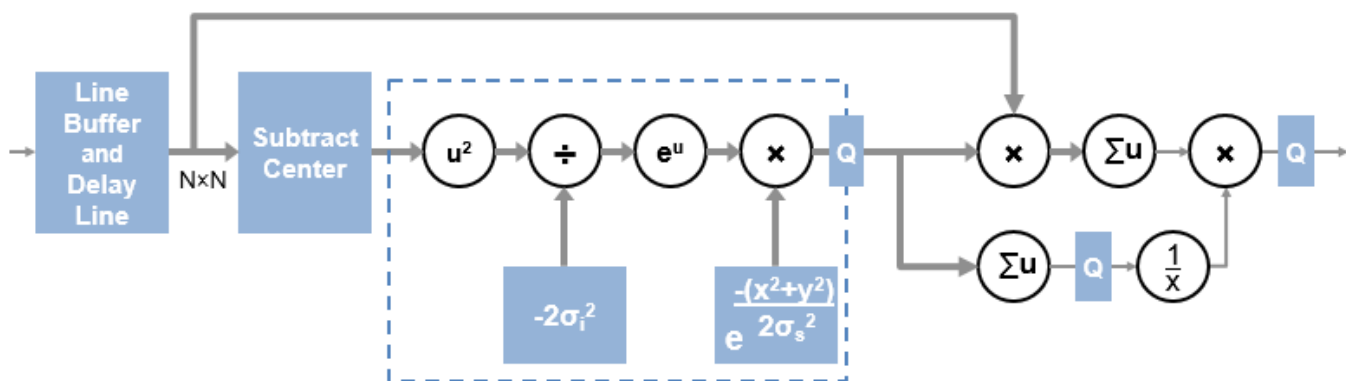
The Subtract Center operation produces a pixel value of zero at the center of the neighborhood. For hardware implementation, and for simulation of fixed-point or integer data types, the calculation in the dashed region is implemented with a lookup table of precomputed values for each pixel. Because the center value is always zero, u^2 and e^u are always one and are not computed. For floating-point input, the simulation computes u^2 and e^u as shown. The output of the dashed region uses the coefficient data type that you specified. The Q blocks in the diagram show quantization points.

The algorithm implements the final normalization step with a reciprocal lookup table in the hardware implementation. The lookup table has 2048 locations, so the coefficient sum is quantized to the most significant 11 bits. The reciprocal values use the output data type that you specified, plus a minimum of two integer bits if the data type does not already include them. The reciprocal lookup value for a zero sum is the maximum representable value in the coefficient data type. For floating-point normalization, the simulation detects a zero sum and instead divides by `eps()` of the dividend.

The output pixel value is then cast to the output data type that you specified. The filter uses the entire range of the data type, so if your color space uses less than the full range, you may need to rescale the pixel values.

Note When filtering multicomponent (color) pixels, false colors can occur, unless the operation is done in a color space based on human perception, such as CIELab. Bilateral filtering of the R'G'B' color space is not recommended.

Latency

The latency of the block is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding by default. The latency of the padding operation depends on the size of the kernel. If edge padding is not necessary for your design, you can reduce the latency by setting the **Padding method** parameter to None. When you use this option, the block latency does not depend on your kernel size. To determine the exact latency for any configuration of the block, measure the number of time steps between the input and output control signals.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch + Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2017b

R2022a: Multipixel streaming

The Bilateral Filter block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

The block supports input column vectors of *NumPixels* values, where *NumPixels* can be 2, 4, or 8. The **ctrl** ports remain scalar, and the control signals in the pixelcontrol bus apply to all pixels in the matrix. The block returns an output vector of the same size as the input vector.

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020a: Option to omit padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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
Frame To Pixels

Objects
visionhdl.BilateralFilter

Birds-Eye View

Transform front-facing camera image into top-down view

Libraries:
Vision HDL Toolbox / Geometric Transforms

Description

The Birds-Eye View block warps a front-facing camera image into a top-down view. It uses a hardware-efficient architecture that supports HDL code generation.

You must provide the homography matrix that describes the transform. This matrix can be calculated from physical camera properties, or empirically derived by analyzing an image of a grid pattern taken by the camera. The block uses the matrix to compute the transformed coordinates of each pixel. The transform does not interpolate between pixel locations. Instead it rounds the result to the nearest coordinate.

The block operates on a trapezoidal region of the input image below the vanishing point. These images show the input region selected for transformation and the resulting top-down view.
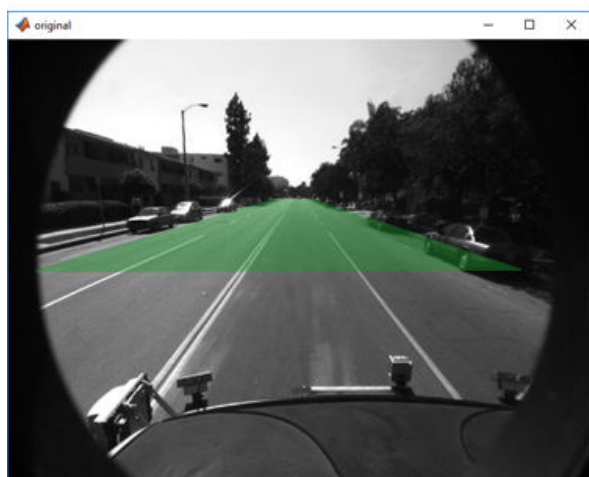
You can specify the number of lines in the transformed region and the size of the output frame. If the specified homography matrix cannot map from the requested number of lines to the requested output size, the block returns a warning.

Because the block replicates lines from the input region to create the larger output frame, it cannot complete the transform of one frame before the next frame arrives. The block ignores any new input frames while it is still transforming the previous frame. Therefore, depending on the stored lines and output size, the block can drop input frames. This timing also enables the block to maintain the blanking intervals of the input pixel stream.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar

Single image pixel in a pixel stream, specified as a scalar that represents grayscale intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Output

pixel — Output pixel stream
scalar

Single image pixel in the pixel stream, returned as a scalar representing grayscale intensity. The output pixel data type is the same as the input pixel data type.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Parameters

Homography matrix — Transfer function derived from camera parameters

`[0.000100990123328 0 0;0.000412396945637 0.001302203393162` `1.293171994e-06;-0.103019798961327 -0.255811259450009 -0.000222053779501]` (default) | 3-by-3 matrix

Transfer function derived from camera parameters, specified as a 3-by-3 matrix.

The homography matrix, *h*, is derived from four intrinsic parameters of the physical camera setup: the focal length, pitch, height, and principal point (from a pinhole camera model). The default value is the matrix for the camera setup used in the "Lane Detection" example.

This matrix can be calculated from physical camera properties, or empirically derived by analyzing an image of a grid test pattern taken by the camera. See `estimateGeometricTransform` or "Using the Single Camera Calibrator App" (Computer Vision Toolbox).

Maximum buffer size, in pixels — Number of input pixels to buffer

`40000` (default) | integer

Number of input pixels to buffer, specified as an integer. Compute this value from **Number of input lines to buffer****ActivePixelsPerLine*. The block uses a memory of this size to store the input pixels. If you specify a value that is not a power of two, the block uses the next largest power of two.

Number of input lines to buffer — Number of lines to transform

`54` (default) | integer

Number of lines to transform, specified as an integer. The block stores and transforms this number of lines into the output bird's-eye view image, starting at the vanishing point as determined by the **Homography matrix**.

Storing the full input frame uses too much memory to implement the algorithm without off-chip storage. Therefore, for a hardware implementation, choose a smaller region to store and transform, one that generates an acceptable output frame size.

For example, using the default **Homography matrix** with an input image of 640-by-480 pixels, the full-sized transform results in a 900-by-640 output image. Analysis of the input-to-output *x*-coordinate mapping shows that around 50 lines of the input image are required to generate the top 700 lines of the bird's-eye view output image. This number of input lines can be stored using on-chip memory. The vanishing point for the default camera setup is around line 200, and lines above that point do not contribute to the resulting bird's-eye view. Therefore, the block can store only input lines 200–250 for transformation.

Output active pixels — Horizontal size of output frame

`640` (default) | integer

Horizontal size of output frame, specified as an integer. This parameter is the number of active pixels in each output line.

Output active lines — Vertical size of output frame

700 (default) | integer

Vertical size of output frame, specified as an integer. This parameter is the number of active lines in each output frame.

Algorithms

The transform from input pixel coordinate *(x,y)* to the bird's-eye pixel coordinate is derived from the homography matrix, *h*. The homography matrix is based on physical parameters and therefore is a constant for a particular camera installation.

$$(\widehat{x}, \widehat{y}) = round\left(\frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}, \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}\right)$$

The implementation of the bird's-eye transform in hardware does not directly perform this calculation. Instead, the block precomputes lookup tables for the horizontal and vertical aspects of the transform.
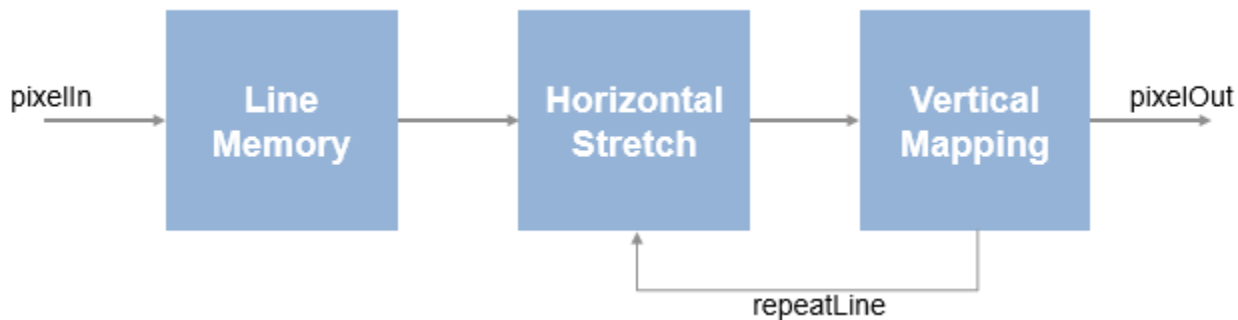
First, the block stores the input lines starting from the precomputed vanishing point. The stored pixels form a trapezoid, with short lines near the vanishing point and wider lines near the camera. This storage uses **Maximum buffer size, in pixels** memory locations.

The horizontal lookup table contains interpolation parameters that describe the stretch of each line of the trapezoidal input region to the requested width of the output frame. Lines that fall closer to the vanishing point are stretched more than lines nearer to the camera.

The vertical lookup table contains the *y*-coordinate mapping, and how many times each line is repeated to fill the requested height of the output frame. Near the vanishing point, one input line maps to many output lines, while each line nearer the camera maps to a diminishing number of output lines.

The lookup tables use 3***Number of input lines to buffer** memory locations.

Version History
Introduced in R2017b

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 one 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
Frame To Pixels | Warp

Objects
visionhdl.BirdsEyeView

Functions
imwarp | estimateGeometricTransform

Topics
"Using the Single Camera Calibrator App" (Computer Vision Toolbox)

Chroma Resampler

Downsample or upsample chrominance component
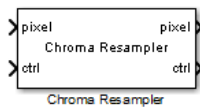
Libraries:
Vision HDL Toolbox / Conversions

Description

The Chroma Resampler block downsamples or upsamples a pixel stream.

- Downsampling reduces bandwidth and storage requirements in a video system by combining pixel chrominance components over multiple pixels. You can specify a filter to prevent aliasing by selecting the default filter or by entering coefficients.
- Upsampling restores a signal to its original rate. To calculate the extra sample, you can use interpolation or replication.

The block accepts luma and chrominance components. The block does not modify the luma component, and applies delay to align it with the resampled chrominance outputs. The rate of the output luma component is the same as the rate of the input.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Pixel in Y'CbCr color space
vector of three values

Pixel in Y'CbCr color space, specified as a vector of three values. Integer and fixed-point data types must be between 8 and 16 bits.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `uint8` | `uint16` | `fixed point`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

Output

pixel — Pixel in Y'CbCr color space
vector of three values

Pixel in Y'CbCr color space, returned as a vector of three values. The data type of the output pixels is the same as the data type of the input pixels.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: single | double | uint8 | uint16 | fixed point

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

Parameters

Main

Resampling — Type of resampling

4:4:4 to 4:2:2 (default) | 4:2:2 to 4:4:4

If you select 4:4:4 to 4:2:2, the block performs a downsampling operation. If you select 4:2:2 to 4:4:4, the block performs an upsampling operation.

Antialiasing filter — Lowpass filter to follow downsample operation

Auto (default) | Property | None

If you select Auto, the block uses a built-in lowpass filter. If you select Property, the **Horizontal filter coefficients** parameter appears on the dialog box. If you select None, the block does not filter the input signal.

Dependencies

This parameter is visible when you set **Resampling** to 4:4:4 to 4:2:2.

Horizontal filter coefficients — Coefficients for the antialiasing filter

[0.2 0.6 0.2] (default) | vector

Coefficients for the antialiasing filter, specified as a vector.

Dependencies

This parameter is visible if you set **Resampling** to 4:4:4 to 4:2:2 and **Antialiasing filter** to Property.

Interpolation — Interpolation method for an upsample operation

`Linear` (default) | `Pixel replication`

If you select `Linear`, the block uses linear interpolation to calculate the missing values. If you select `Pixel replication`, the block repeats the chrominance values of the preceding pixel to create the missing pixel.

Dependencies

This parameter is visible if you set **Resampling** to `4:2:2 to 4:4:4`.

Data Types

The parameters on this tab appear only when they are relevant. If you configure the block so that no filter coefficients are needed, or no rounding or overflow is possible, the irrelevant parameter is hidden.

Rounding mode — Rounding method for internal fixed-point calculations

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

Specify a rounding method for internal fixed-point calculations.

Saturate on integer overflow — Overflow action for internal fixed-point calculations

`off` (default) | `on`

When you clear this parameter, fixed-point and integer values wrap around to zero when the value overflows what is representable with that data type. When you select this parameter, the value saturates at the maximum representable value.

Filter coefficients — Data type for antialiasing filter coefficients

`fixdt(1,16,15)` (default) | data type expression

This parameter applies when you set **Antialiasing filter** to `Auto` or `Property`.

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

The default antialiasing filter is a 29-tap lowpass filter that matches the default Chroma Resampling block in Computer Vision Toolbox™. In the frequency response of this filter, the passband, [–0.25 0.25], occupies half of the total bandwidth. This filter suppresses aliasing after `4:4:4 to 4:2:2` downsampling.
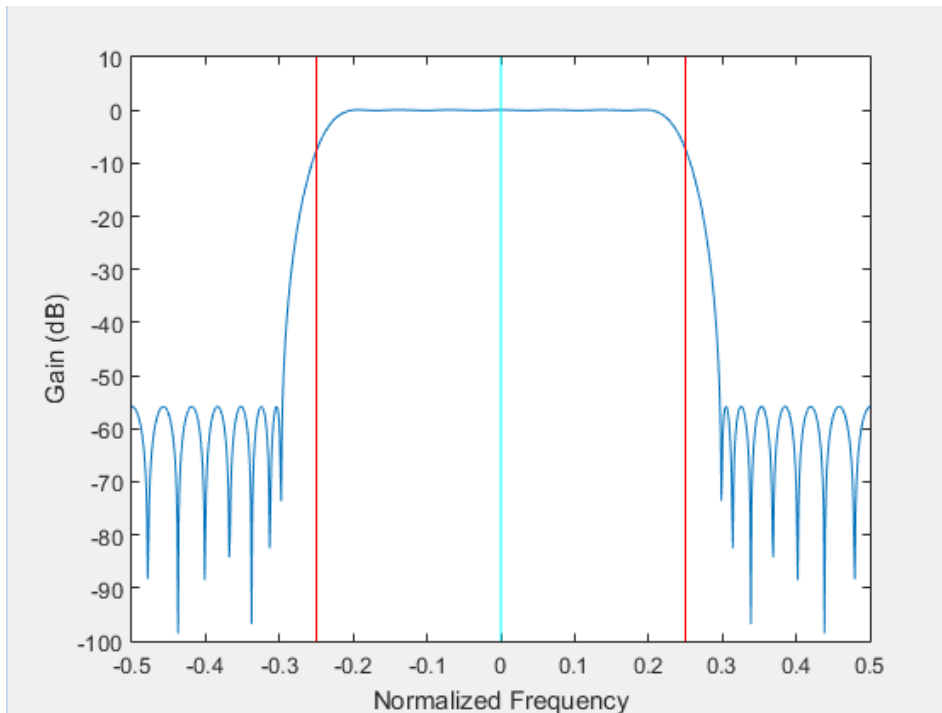
Whether you use the default filter or specify your own coefficients, the filter is implemented in HDL using a fully parallel architecture. To reduce the number of multipliers, HDL code generation takes advantage of symmetric, unity, or zero-value coefficients.

The block pads the edge of the image with symmetric pixel values. See "Edge Padding". Also, if the frame is an odd number of pixels wide, the block symmetrically pads the line. This accommodation makes the block more resilient to video timing variation.

Latency

The latency is the number of cycles between the first valid input pixel and the first valid output pixel. When you use an antialiasing filter, the latency depends on the size and value of the filter coefficients. The FIR delay can be less than the number of coefficients because the block optimizes out duplicate or zero-value coefficients.

Block Configuration	Latency
Downsample (4:4:4 to 4:2:2); no filter	3
Downsample (4:4:4 to 4:2:2); with filter	$4+(N/2)+$FIR delay, where N = number of filter coefficients
Upsample (4:2:2 to 4:4:4); replication	3
Upsample (4:2:2 to 4:4:4); interpolation	5

For example, the latency for a downsample using the default filter is 30 cycles.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming

video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch** + **Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2015a

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 one 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
Chroma Resampling | Frame To Pixels

Objects
visionhdl.ChromaResampler

Closing

Morphological closing of binary pixel data
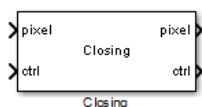
Libraries:
Vision HDL Toolbox / Morphological Operations

Description

The Closing block performs morphological dilation followed by morphological erosion by using the same neighborhood for both calculations. The block operates on a stream of binary intensity values. You can specify a neighborhood, or structuring element, of up to 32-by-32 pixels.

Note This block matches the behavior of the Closing block in Computer Vision Toolbox and of the combination of `imerode(imdilate())` in Image Processing Toolbox™. However, it does not match the edge behavior of the `imclose` function in Image Processing Toolbox. `imclose` assumes that white is the foreground color of a binary image, which can result in differences in edge behavior.

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Ports

Input

pixel — Boolean input pixel or multipixel vector
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar Boolean value. For multipixel streaming, specify a vector of four or eight Boolean values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

Data Types: `Boolean`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Output

pixel — Boolean output pixel or multipixel vector
scalar | vector

This block supports single pixel streaming or multipixel streaming. When using single pixel streaming, the block returns a single pixel as a scalar Boolean value. When using multipixel streaming, the block returns a vector of Boolean values. This vector is the same size as the input **pixel** vector. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

Data Types: `Boolean`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Parameters

Neighborhood — Pixel neighborhood

`[0,1,0; 1,1,1; 0,1,0]` (default) | vector or matrix of 1s and 0s

Pixel neighborhood, specified as a vector or matrix of binary values.

The block supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the **Neighborhood** parameter to `getnhood(strel(`*shape*`))`, where *shape* is specified by the input arguments to the `strel` function..

When you use multipixel vector input, the neighborhood must be at least two pixels in each dimension.

Line buffer size — Size of line memory buffer

`2048` (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (n – 1)-by-**Line buffer size** memory locations to store the pixels, where n is the number of lines in the **Neighborhood** parameter value.

Padding method — Method for padding

`Constant` (default) | `None`

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

*   `Constant` — The block pads the image with zeros for the dilation operation and with ones for the erosion operation. These values prevent closing at the boundaries of the active frame.
*   `None` — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

Tips

*   When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

Latency

The total latency of the block is the line buffer latency plus the latency of the kernel calculation. Morphological closing is a compound operation, so this block contains a second line buffer between the dilation kernel and the erosion kernel. To determine the exact latency for any configuration of the block, monitor the number of time steps between the input control signals and the output control signals.

The latency of the line memory includes edge padding. The latency of the kernel depends on the neighborhood size.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch + Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2015a

R2020a: Option to omit padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

R2020a: Multipixel streaming

The Closing block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

For multipixel streaming, the blocks support input and output column vectors of 4 or 8 pixels. The **ctrl** port remains scalar, and the control signals in the pixelcontrol bus apply to all pixels in the vector.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 for this block inside a Resettable Synchronous Subsystem.

See Also
Dilation | Erosion | Opening | Frame To Pixels | `visionhdl.Closing`

Topics
"Types of Morphological Operations" (Image Processing Toolbox)

"Structuring Elements" (Image Processing Toolbox)

Color Space Converter

Convert color information between color spaces
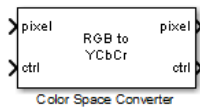
Libraries:
Vision HDL Toolbox / Conversions

Description

The Color Space Converter block converts between R'G'B' and Y'CbCr color spaces, and also converts R'G'B' to intensity.

Note The Color Space Converter block operates on gamma-corrected color spaces, usually indicated with prime notation ('). However, for simplicity, the block and mask labels do not include the prime notation.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

This block also supports multipixel streams, where the pixel input is a matrix of *M*-by-3 values. *M* is the number of pixels and each pixel has 3 R'G'B' or Y'CbCr components. These values correspond to the **Number of pixels** and **Number of components** parameters of the Frame To Pixels block.

Input

pixel — Input pixel stream
vector | matrix

For scalar pixel streams, specify **pixel** as a vector of 1-by-3 values. For multipixel streams, specify **pixel** as a matrix of **Number of pixels**-by-3 pixel intensity values. **Number of pixels** can be two, four, or eight.

The pixel stream must be in Y'CbCr or R'G'B' color space. Integer and fixed-point data types must be between 8 and 16 bits.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `uint8` | `uint16` | `fixed point`

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one valid signal, the pixels in the vector must be either all valid or all invalid. The hStart and vStart signals apply to the pixel with the lowest index in the vector. The hEnd and vEnd signals apply to the pixel with the highest index in the vector.

Data Types: bus

Output

pixel — Output pixel stream in new colorspace
scalar | vector | matrix

Output pixel stream in intensity, Y'CbCr, or R'G'B' color space, returned as a single pixel stream or multipixel stream. The data type and **Number of pixels** of the output stream is the same as the input pixel stream. If the output is intensity values, each pixel has one component. If the output is Y'CbCr or R'G'B', each pixel has three components.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: single | double | uint8 | uint16 | fixed point

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one valid signal, the pixels in the vector must be either all valid or all invalid. The hStart and vStart signals apply to the pixel with the lowest index in the vector. The hEnd and vEnd signals apply to the pixel with the highest index in the vector.

Data Types: bus

Parameters

Conversion — Type of color space conversion

RGB to YCbCr (default) | YCbCr to RGB | RGB to intensity

The block accepts input pixels as vectors of three values that represent a single pixel. If you choose RGB to intensity, each output pixel is a scalar. Otherwise, each output pixel is a vector of three values.

Use conversion specified by — Conversion equation

Rec. 601 (SDTV) (default) | Rec. 709 (HDTV)

Conversion equation used between R'G'B' and Y'CbCr color spaces.

Dependencies

This parameter applies only when you set **Conversion** to `RGB to YCbCr` or `YCbCr to RGB`.

Scanning standard — HDTV scanning standard

`1250/50/2:1` (default) | `1125/60/2:1`

Scanning standard used to convert between R'G'B' and Y'CbCr color spaces in HDTV format.

Dependencies

This parameter applies when you set **Use conversion specified by** to `Rec. 709 (HDTV)`.

Algorithms

When you use multipixel streaming, the block replicates the conversion algorithm for each of the M input pixels, in parallel. This increase in hardware resources is a trade off for increasing throughput compared to single-pixel streaming.

Conversion Between R'G'B' and Y'CbCr Color Spaces

The following equations define R'G'B' to Y'CbCr conversion and Y'CbCr to R'G'B' conversion:

$$\begin{bmatrix} Y' \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + A \times \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = B \times \left(\begin{bmatrix} Y' \\ Cb \\ Cr \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \right)$$

The values in matrices A and B are based on your choices for the **Use conversion specified by** and **Scanning standard** parameters.

Matrix	Use conversion specified by = Rec. 601 (SDTV)	Use conversion specified by = Rec. 709 (HDTV)	
		Scanning standard = 1125/60/2:1	Scanning standard = 1250/50/2:1
A	$\begin{bmatrix} 0.25678824 & 0.50412941 & 0.09790588 \\ -0.1482229 & -0.29099279 & 0.43921569 \\ 0.43921569 & -0.36778831 & -0.07142737 \end{bmatrix}$	$\begin{bmatrix} 0.18258588 & 0.61423059 & 0.06200706 \\ -0.10064373 & -0.33857195 & 0.43921569 \\ 0.43921569 & -0.39894216 & -0.04027352 \end{bmatrix}$	$\begin{bmatrix} 0.25678824 & 0.50412941 & 0.09790588 \\ -0.1482229 & -0.29099279 & 0.43921569 \\ 0.43921569 & -0.36778831 & -0.07142737 \end{bmatrix}$
B	$\begin{bmatrix} 1.1643836 & 0 & 1.5960268 \\ 1.1643836 & -0.39176229 & -0.81296765 \\ 1.1643836 & 2.0172321 & 0 \end{bmatrix}$	$\begin{bmatrix} 1.16438356 & 0 & 1.79274107 \\ 1.16438356 & -0.21324861 & -0.53290933 \\ 1.16438356 & 2.11240179 & 0 \end{bmatrix}$	$\begin{bmatrix} 1.1643836 & 0 & 1.5960268 \\ 1.1643836 & -0.39176229 & -0.81296765 \\ 1.1643836 & 2.0172321 & 0 \end{bmatrix}$

Conversion from R'G'B' to Intensity

The following equation defines conversion from the R'G'B' color space to intensity:

$$\text{intensity} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

Data Types

For fixed-point and integer input, the block converts matrix *A* to `fixdt(1,17,16)`, and matrix *B* to `fixdt(1,17,14)`.

For double or single input, the block applies the conversion matrices in double type, and scales the Y'CbCr offset vector (`[16,128,128]`) by 1/255. The block saturates double or single R'G'B' and intensity outputs to the range `[0,1]`.

The Y'CbCr standard includes headroom and footroom. For 8-bit data, luminance values in the range 16–235 and chrominance values in the range 16–240 are valid. The Color Space Converter block pins out-of-range input to these limits before calculating the conversion. The block scales the offset vector and the allowed headroom and footroom depending on the word length of the input signals. For example, when you convert a Y'CbCr input of type `fixdt(0,10,0)` to R'G'B', the block multiplies the offset vector by $2^{(10-8)} = 4$. As a result, the valid luminance range becomes 64–940 and the valid chrominance range becomes 64–960.

Latency

When you use this block with R'G'B' input, the block has a latency of 9 cycles. When you use this block with Y'CbCr input, the block has a latency of 10 cycles. The extra cycle is required to check for and correct headroom and footroom violations.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch + Back porch**. For more information, see "Configure Blanking Intervals".

Version History

Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2021b: Multipixel streaming

The Color Space Converter block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

The block supports input matrices of *NumPixels*-by-3 values, and output matrices of *NumPixels*-by-*NumComponents* values, where *NumComponents* is 3 or 1. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

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 one 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
Color Space Conversion | Frame To Pixels | `visionhdl.ColorSpaceConverter`

Corner Detector

Find corners using FAST or Harris algorithm

Libraries:
Vision HDL Toolbox / Analysis & Enhancement

Description

The Corner Detector block detects corners by using the features-from-accelerated-segment test (FAST) algorithm or by the intersecting edges (Harris) algorithm . For each pixel, if the pixel is a corner, the block returns the corner metric. If the pixel is not a corner, the block returns a pixel value of zero.

The FAST algorithm tests a circular area around the potential center of the corner. The test detects a corner if a contiguous section of pixels are either brighter than the center plus a threshold or darker than the center minus a threshold. You can specify a minimum contrast threshold as a parameter or port and select from three rules to define a corner. These rules specify how many pixels in a circle of pixels must exceed the minimum contrast for the center pixel to be considered a corner. The block performs parallel tests of all combinations of contiguous pixels around the circle. FAST uses very little hardware resources

The Harris algorithm computes horizontal and vertical gradients, filters the gradient components with a circular 5-by-5 Gaussian filter, and computes a metric that represents the strength of the corner. You can specify a threshold that determines the level at which corners are detected. The block returns a corner for pixels where the metric exceeds this threshold. The Harris algorithm uses more hardware resources than the FAST algorithm but can detect corners that the FAST algorithm might not find.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar

Single image pixel in a pixel stream, specified as a scalar that represents grayscale intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

minC — Threshold that indicates corner for FAST algorithm
nonnegative integer

Threshold that indicates a corner for the FAST algorithm, specified as a nonnegative integer. Each pixel in the circle is subtracted from the center pixel value, and only those differences that exceed this threshold are used to determine if the center pixel is a corner and to calculate the output metric. This value is cast to the same data type as the input **pixel**.

The software supports double and single data types for simulation, but not for HDL code generation.

Dependencies

To enable this port, set the **Method** parameter to one of the FAST options and set the **Source of minimum contrast value** parameter to Input port.

Data Types: single | double | uint | fixed point

thresh — Threshold that indicates corner for Harris algorithm
nonnegative integer

Threshold that indicates a corner for the Harris algorithm, specified as a nonnegative integer. This value represents an approximation of the eigenvalues of the Harris matrix calculated from the gradient results. The block returns a corner for pixels where the eigenvalue metric exceeds this threshold. This value is cast to the same data type as the output **pixel**. The corner metrics are in the range of a fourth power of the input pixel values. For example, for an 8-bit input pixel, the full-precision output size is 44 bits.

The software supports double and single data types for simulation, but not for HDL code generation.

Dependencies

To enable this port, set the **Method** parameter to Harris and set the **Source of threshold value** parameter to Input port.

Data Types: single | double | uint | fixed point

Output

corner — Metric that indicates whether pixel is a corner
scalar

Metric that indicates whether the pixel is a corner, returned as a scalar of the numeric type specified by the **Output** data type parameter. When the pixel is not a corner, the block returns zero for this value.

For corners detected with the FAST algorithm, this value is a contrast metric computed by using the sum-of-absolute-difference (SAD) of the pixels around the circle with the center pixel value. The metric includes only those differences that exceed the threshold.

For corners detected with the Harris algorithm, this value is the approximation of the eigenvalues of the Harris matrix calculated from the gradient results.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Parameters

Main

Method — Detection method

`FAST 5 of 8` (default) | `FAST 7 of 12` | `FAST 9 of 16` | `Harris`

Select the Harris algorithm, or the size of the circle for the FAST algorithm. The three FAST options specify how many contiguous pixels on the circle must exceed the threshold for the center pixel to be marked as a corner. For the circles of 8, 12, and 16 pixels shown in these figures, the block detects a corner when 5, 7, or 9 contiguous pixels, respectively, are above the threshold. The block checks for this number of contiguous pixels anywhere on the circle. The block uses a kernel of size 3-by-3 for `FAST 5 of 8`, 5-by-5 for `FAST 7 of 12`, and 7-by-7 for `FAST 9 of 16`.

corner = any 5 in a row

corner = any 7 in a row

corner = any 9 in a row
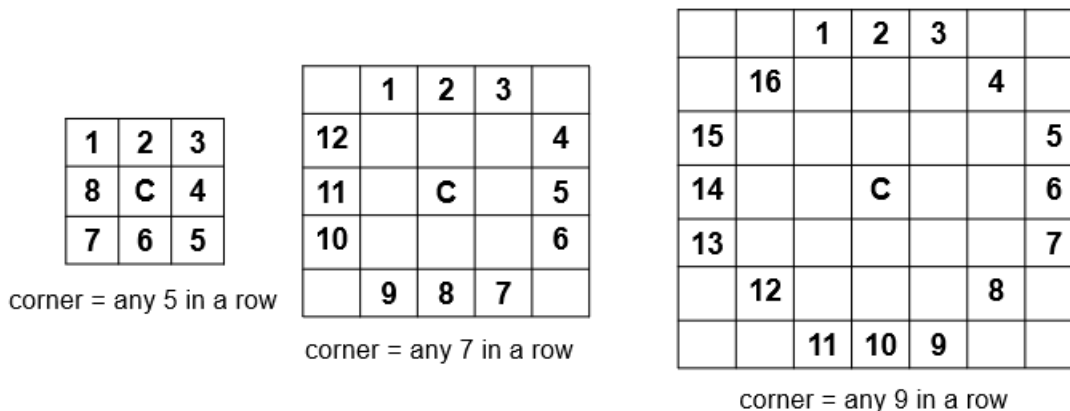
Source of minimum contrast value — Source for threshold that indicates corner for FAST algorithm

`Property` (default) | `Input port`

Specify `Property` to set the threshold by using the **MinContrast value** parameter. Specify `Input port` to enable the **minC** port.

Dependencies

To enable this parameter, set the **Method** parameter to one of the FAST options.

MinContrast value — Threshold that indicates corner for FAST algorithm

20 (default) | nonnegative integer

Each pixel in the circle is subtracted from the center pixel value, and only those differences that exceed this threshold are used to determine if the center pixel is a corner and to calculate the output metric. This value is cast to the same data type as the input **pixel**.

Dependencies

To enable this parameter, set the **Method** parameter to one of the FAST options and set the **Source of minimum contrast value** parameter to `Property`.

Source of threshold value — Source for threshold that indicates corner for Harris algorithm

Property (default) | Input port

Specify `Property` to set the threshold by using the **Threshold value** parameter. Specify `Input port` to enable the **thresh** port.

Dependencies

To enable this parameter, set the **Method** parameter to `Harris`.

Threshold value — Threshold that indicates corner for Harris algorithm

15000 (default) | nonnegative integer

This value represents an approximation of the eigenvalues of the Harris matrix calculated from the gradient results. The block returns a corner for pixels where the metric exceeds this threshold. For details, see the "Harris Corner Detection" example. This value is cast to the same data type as the output **corner**. The corner metric is in the range of a fourth power of the input pixel values. For example, for an 8-bit input pixel, the full-precision output size is 44 bits.

Dependencies

To enable this parameter, set the **Method** parameter to `Harris` and set the **Source of threshold value** parameter to `Property`.

Line buffer size — Size of line buffer

2048 (default) | integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (N – 1)-by-**Line buffer size** memory locations to store the pixels. N is the size of the kernel.

The block uses a kernel of size 3-by-3 when the **Method** parameter is FAST 5 of 8, 5-by-5 when **Method** is FAST 7 of 12, and 7-by-7 when **Method** is FAST 9 of 16.

When you set **Method** to Harris, the block uses one 1-by-3 kernel and one 3-by-1 kernel for the gradient calculation and three 5-by-5 kernels for the circular window filter on the three gradient components.

Padding method — Method for padding

Symmetric (default) | Replicate | Reflection | None

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- Replicate — Repeat the value of pixels at the edge of the image.
- Symmetric — Set the value of the padding pixels to mirror the edge of the image. This option prevents edges from being detected at the boundaries of the active frame.
- Reflection — Set the value of the padding pixels to reflect around the pixel at the edge of the image.
- None — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

The circular window filters used in the Harris algorithm pad the edges of the frame using Replicate padding.

Data Types

Rounding mode — Rounding method for internal fixed-point calculations

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

Specify a rounding method for internal fixed-point calculations.

Saturate on integer overflow — Overflow action for internal fixed-point calculations

off (default) | on

When you clear this parameter, fixed-point and integer values wrap around to zero when the value overflows what is representable with that data type. When you select this parameter, the value saturates at the maximum representable value.

Output — Data type for **corner** output port

Inherit same as first input (default) | data type expression | Inherit via internal rule

Specify a data type for the output **corner** port. By default, the block returns the corner metric using the same data type as the input **pixel** port.

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

When the block implements the FAST algorithm, it uses a single line buffer to extract the kernel windows. The algorithm subtracts the center pixel of the kernel from each of the circle pixels. For kernel diagrams, see the "Method" on page 1-0   parameter. Each result is compared to the *minContrast* value. When the required number of consecutive pixels exceed the threshold, the center is marked as a corner, and the block computes the sum-of-absolute-difference (SAD) metric for the circle. Only those differences that exceed the minimum contrast threshold are included in the metric. Noncontiguous pixels are also included in the metric. This calculation means that the algorithm detects a light center pixel surrounded by dark pixels or a dark center pixel surrounded by light pixels as corners with high metrics. To optimize hardware performance, the algorithm is pipelined between each of the add or subtract operations.

The Harris algorithm uses five image filters to calculate the two gradients and three circular windows. The calculation of the approximation of the eigenvalue of the Harris matrix uses three multipliers and three adders and is pipelined to optimize hardware performance. The diagram shows the architecture of the Harris algorithm, where A is the Harris matrix.
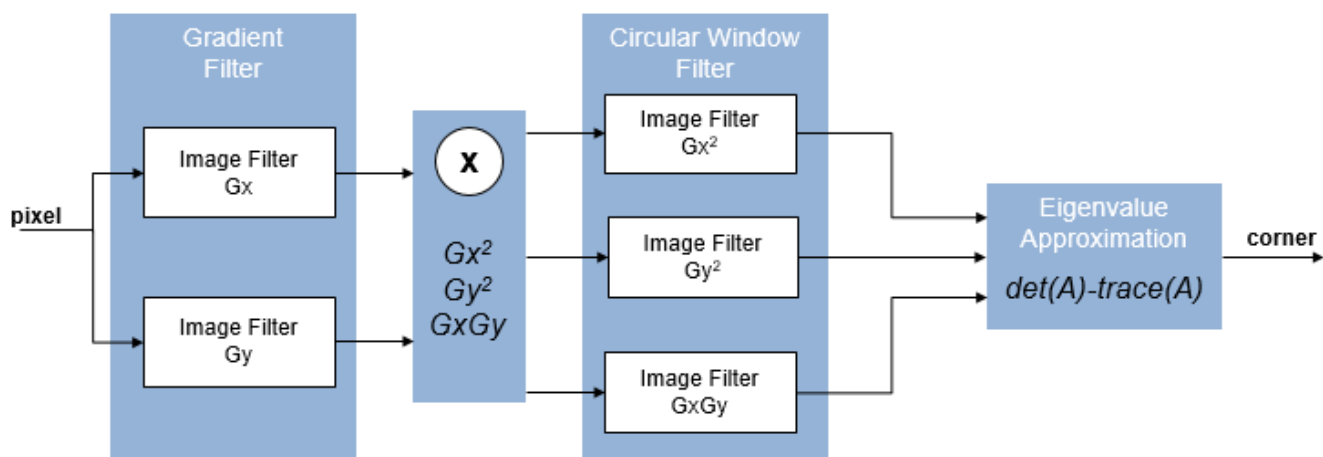
For additional details of the Harris algorithm, see the "Harris Corner Detection" example.

Version History
Introduced in R2020a

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020b: Harris corner detection

The block now provides a choice between the FAST algorithm and the Harris and Stephens interconnecting edges algorithm. See "Harris Corner Detection".

R2020a: Option to omit padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

References

[1] "HIGH-SPEED IMAGE FEATURE DETECTION USING FPGA IMPLEMENTATION OF FAST ALGORITHM:" *Proceedings of the Third International Conference on Computer Vision Theory and Applications*, SciTePress - Science and Technology Publications, 2008, pp. 174–79. doi:10.5220/0001080801740179.

[2] Rosten, E., and T. Drummond. *"Fusing Points and Lines for High Performance Tracking."* Proceedings of the IEEE International Conference on Computer Vision, vol. 2, 2005, pp. 1508–11.

[3] Rosten, E., and T. Drummond. "*Machine Learning for High-Speed Corner Detection.*" Computer Vision - ECCV 2006 Lecture Notes in Computer Science, 2006, 430-43. doi:10.1007/11744023_34.

[4] Harris, C., and M. Stephens. *"A Combined Corner and Edge Detector."* In Proceedings of the Alvey Vision Conference 1988, 23.1-23.6. Manchester: Alvey Vision Club, 1988. *DOI.org (Crossref)*, doi:10.5244/C.2.23.

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 one 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
Frame To Pixels

Objects
visionhdl.CornerDetector

HV Counter

Count active dimensions of pixel stream
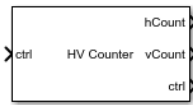
Libraries:
Vision HDL Toolbox / Utilities

Description

The HV Counter block analyzes a video stream and returns the current count of lines per frame and pixels per line. The block also delays control signals of the pixel stream to correspond with the count result. Use this block for algorithms that use the location of a pixel within a frame or region of interest.

This waveform shows the pixel-stream control signals and resulting counter outputs for the first two lines of a video frame that has 10 pixels per line. The HV Counter block has a latency of two cycles to return the current counter values.
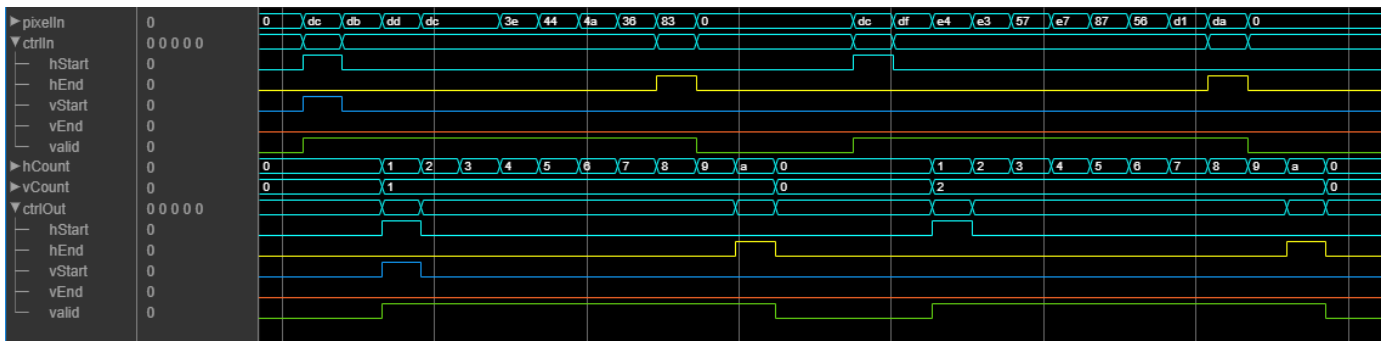
Ports

This block uses a bus for frame control signals associated with each pixel of a pixel stream. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a bus containing five control signals. The control signals indicate the validity of each pixel and its location in the frame. For a full description of the interface, see "Streaming Pixel Interface".

Input

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Output

hCount — Pixel location in line
positive integer

Pixel location within a line, returned as a positive integer. The block clears the pixel count at the start of each line. If the input pixels per line exceeds the size of the counter, the block returns a saturated value until the start of the next line.

Data Types: `fixdt(0,ceil(log2(ActiveVideoLines)),0)`

vCount — Line location in frame
positive integer

Line location within the frame or region of interest, returned as a positive integer. The block clears the line count at the start of each frame. If the input lines per frame exceeds the size of the counter, the block returns a saturated value until the start of the next frame.

Data Types: `fixdt(0,ceil(log2(ActiveVideoLines)),0)`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Parameters

Active pixels per line — Maximum pixels per line

`320` (default) | positive integer

Maximum pixels per line, specified as a positive integer. The block implements a pixel counter that has `ceil(log`$_2$`(ActivePixelsPerLine))` bits. For example, for a frame with 320 pixels per line, the counter has 9 bits. For a frame with 1024 pixels per line, the counter has 11 bits.

Active video lines — Maximum lines per frame

`240` (default) | positive integer

Maximum lines per frame, specified as a positive integer. The block implements a line counter that has `ceil(log`$_2$`(ActiveVideoLines))` bits. For example, for a frame with 240 pixels per line, the counter has 8 bits. For a frame with 1080 pixels per line, the counter has 11 bits.

Version History
Introduced in R2019a

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 one 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
Frame To Pixels

Objects
visionhdl.HVCounter

Demosaic Interpolator

Construct RGB pixel data from Bayer pattern pixel stream
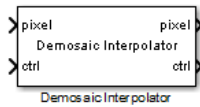
Libraries:
Vision HDL Toolbox / Conversions

Description

The Demosaic Interpolator block provides a Bayer pattern interpolation filter for streaming video data. The block implements the calculations using hardware-efficient, multiplier-free algorithms for HDL code generation. You can select a low-complexity bilinear interpolation, or a moderate-complexity gradient-corrected bilinear interpolation.

- When you select bilinear interpolation, the block operates on a 3-by-3 pixel window using only additions and bit shifts.
- When you select gradient correction, the block operates on a 5-by-5 pixel window. The calculation is performed using bit shift, addition, and low-order canonical signed digit (CSD) multiplication.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

This block also supports multipixel streams. In that case, the pixel input is a vector of *M*-by-1 values, where *M* is number of pixels. The pixel output is a matrix of *M*-by-3 values. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block, and each output pixel has three components in RGB color space.

Input

pixel — Input pixel stream
scalar | vector

For single pixel streams, specify **pixel** as a scalar. For multipixel streams, specify **pixel** as a vector of **Number of pixels**-by-1 pixel values. **Number of pixels** can be two, four, or eight. Images in the Bayer format have one color component for each pixel location. Select the sequence of R, G, and B pixels by using the **Sensor alignment** parameter.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `fixdt(0,W,F)` | `single` | `double`

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one valid signal, the pixels in the vector must be either all valid or all invalid. The hStart and vStart signals apply to the pixel with the lowest index in the vector. The hEnd and vEnd signals apply to the pixel with the highest index in the vector.

Data Types: bus

Output

pixel — Output pixel stream
vector | matrix

For single pixel streams, the output **pixel** is a three-element vector of RGB values. For multipixel streams, the output **pixel** is a matrix of **Number of pixels**-by-3 RGB values. The block calculates the values of the missing color components for each pixel, using the method you specify in the **Interpolation algorithm** parameter.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: uint | fixdt(0,W,F) | single | double

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one valid signal, the pixels in the vector must be either all valid or all invalid. The hStart and vStart signals apply to the pixel with the lowest index in the vector. The hEnd and vEnd signals apply to the pixel with the highest index in the vector.

Data Types: bus

Parameters

Interpolation algorithm — Algorithm used to calculate missing pixel values

Gradient-corrected linear (default) | Bilinear

Algorithm used to calculate missing pixel values, specified as one of the following:

- Gradient-corrected linear — Bilinear average, corrected for intensity gradient
- Bilinear — Average of the pixel values in the surrounding 3-by-3 neighborhood

Sensor alignment — Color sequence of the pixels in the input stream

RGGB (default) | GBRG | GRBG | BGGR

Select the sequence of R, G, and B pixels that correspond to the 2-by-2 block of pixels in the top-left corner of the input image. Specify the sequence in left-to-right, top-to-bottom order. For instance, the default sequence of RGGB represents an image with this pattern.

R	G
G	B

Line buffer size — Size of line memory buffer

2048 (default) | integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The total memory size allocated depends on your selection of **Interpolation algorithm**:

- `Bilinear`: 2-by-**Line buffer size** memory locations
- `Gradient-corrected linear`: 4-by-**Line buffer size** memory locations

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

The block pads the edges of the image with symmetric pixel values. For more details, see "Edge Padding".

When you use multipixel streaming, the block uses a single line memory and implements a demosaic interpolation algorithm for each of the *M* input pixels, in parallel. This increase in hardware resources is a trade off for increasing throughput compared to single-pixel streaming.

Interpolation

Bilinear Interpolation

The block interpolates the missing color values by using a 3-by-3 neighborhood. The average is calculated over the adjacent two pixels or four pixels, depending on the sensor color pattern. The block implements this algorithm using only add and shift operations.

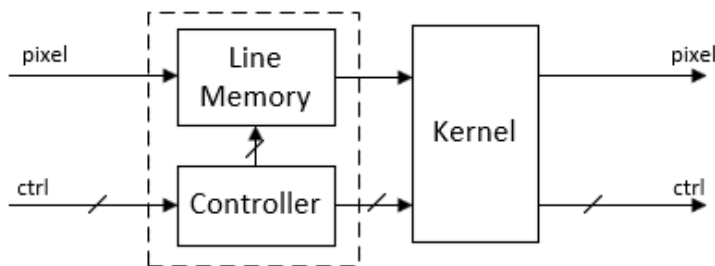#### Gradient-Corrected Linear Interpolation

Gradient correction improves interpolation performance across edges by taking advantage of the correlation between the color channels. The block calculates the missing color values using bilinear interpolation, and then modifies the value corresponding to the intensity gradient calculated over a 5-

by-5 neighborhood. The block applies the gradient correction using a fixed set of filter kernels. The filter coefficients were designed empirically to perform well over a wide range of image data. To enable an efficient hardware implementation, the coefficients are multiples of powers of two. For details of this interpolation algorithm, see [1].

Latency

The block buffers one line of input pixels before starting bilinear interpolation calculations. The gradient correction calculation starts after the block buffers two lines.

The latency of the block is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding by default. The latency of the padding operation depends on the size of the kernel. If edge padding is not necessary for your design, you can reduce the latency by setting the **Padding method** parameter to None. When you use this option, the block latency does not depend on your kernel size. To determine the exact latency for any configuration of the block, measure the number of time steps between the input and output control signals.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch + Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2021b: Multipixel streaming

The Demosaic Interpolator block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

The block accepts an input vector of *NumPixels*-by-1 values and returns an output matrix of *NumPixels*-by-3 values. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

References

[1] Malvar, Henrique S., Li-wei He, and Ross Cutler. "High-Quality Linear Interpolation for Demosaicing of Bayer-Patterned Color Images." *Microsoft Research*, May 2004. http://research.microsoft.com/pubs/102068/Demosaicing_ICASSP04.pdf.

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 one 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 for this block inside a Resettable Synchronous Subsystem.
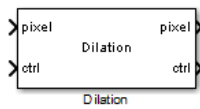
See Also

Blocks
Demosaic | Frame To Pixels

Objects
visionhdl.DemosaicInterpolator

Dilation

Morphological dilation of binary pixel data

Libraries:
Vision HDL Toolbox / Morphological Operations

Description

The Dilation block replaces each pixel with the local maximum of the neighborhood around the pixel. The block operates on a stream of binary intensity values. You can specify a neighborhood or structuring element of up to 32-by-32 pixels.

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Ports

Input

pixel — Boolean input pixel or multipixel vector
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar Boolean value. For multipixel streaming, specify a vector of four or eight Boolean values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

Data Types: `Boolean`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Output

pixel — Boolean output pixel or multipixel vector
scalar | vector

This block supports single pixel streaming or multipixel streaming. When using single pixel streaming, the block returns a single pixel as a scalar Boolean value. When using multipixel streaming, the block returns a vector of Boolean values. This vector is the same size as the input **pixel** vector. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

Data Types: `Boolean`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Parameters

Neighborhood — Pixel neighborhood

`ones(3,3)` (default) | vector or matrix of 1s and 0s

Pixel neighborhood, specified as a vector or matrix of binary values.

The block supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the **Neighborhood** parameter to `getnhood(strel(`*shape*`))`, where *shape* is specified by the input arguments to the `strel` function..

When you use multipixel vector input, the neighborhood must be at least two pixels in each dimension.

Line buffer size — Size of line memory buffer

`2048` (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (n – 1)-by-**Line buffer size** memory locations to store the pixels, where n is the number of lines in the **Neighborhood** parameter value.

Padding method — Method for padding

Constant (default) | None

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- Constant — The block pads the image with zeros. This value prevents dilation at the boundaries of the active frame.

- None — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

Latency

The latency of the block is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding by default. The latency of the padding operation depends on the size of the kernel. If edge padding is not necessary for your design, you can reduce the latency by setting the **Padding method** parameter to None. When you use this option, the block latency does not depend on your kernel size. To determine the exact latency for any configuration of the block, measure the number of time steps between the input and output control signals.

The latency of the kernel depends on the neighborhood size.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch** + **Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2015a

R2020a: Option to not add padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

R2020a: Multipixel streaming

The Dilation block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

For multipixel streaming, the blocks support input and output column vectors of 4 or 8 pixels. The **ctrl** port remains scalar, and the control signals in the pixelcontrol bus apply to all pixels in the vector.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 if it is inside a Resettable Synchronous Subsystem.
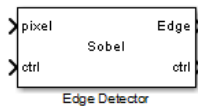
See Also
Erosion | Frame To Pixels | `visionhdl.Dilation`

Topics
"Types of Morphological Operations" (Image Processing Toolbox)
"Structuring Elements" (Image Processing Toolbox)

Edge Detector

Find edges of objects in grayscale pixel stream

Libraries:
Vision HDL Toolbox / Analysis & Enhancement

Description

The Edge Detector block finds the edges in a grayscale pixel stream by using the Sobel, Prewitt, or Roberts method. The block convolves the input pixels with derivative approximation matrices to find the gradient of pixel magnitude along two orthogonal directions. It then compares the sum of the squares of the gradients to the square of a configurable threshold to determine if the gradients represent an edge.

By default, the block returns a binary image as a stream of pixel values. A pixel value of 1 indicates that the pixel is an edge. You can disable the edge output. You can also enable output of the gradient values in the two orthogonal directions at each pixel.

Ports

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel**, **Edge**, and gradient ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus containing five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar intensity value. For multipixel streaming, specify a vector of two, four, or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

This block does not support multicomponent streaming. To process multicomponent streams, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect a single bus to multiple replicated blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Th — Threshold value
scalar

Threshold value that defines an edge, specified as a scalar. The block compares the square of this value to the sum of the squares of the gradients.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

Output

Edge — Boolean pixel value, indicating whether pixel is an edge
scalar | vector

For single pixel streaming, **Edge** is a Boolean scalar. For multipixel streaming, **Edge** is a vector of *M*-by-1 Boolean values. Each pixel value indicates whether the pixel is an edge.

Data Types: `Boolean`

Gv,Gh — Vertical and horizontal gradient
scalar | vector

Vertical and horizontal gradient values calculated over the kernel centered at a pixel location.

For single pixel streaming, the block returns **Gv** and **Gh** as scalar values. For multipixel streaming, the block returns **Gv** and **Gh** as vectors of *M*-by-1 values.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Dependencies

To enable these ports, set **Method** to `Sobel` or `Prewitt`.

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

G45,G135 — Orthogonal gradient
scalar | vector

Orthogonal gradient values calculated over the kernel centered at a pixel location.

For single pixel streaming, the block returns **G45** and **G135** as scalar values. For multipixel streaming, the block returns **G45** and **G135** as vectors of *M*-by-1 values.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Dependencies

To enable these ports set **Method** to `Roberts`.

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

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Parameters

Main

Method — Edge detection algorithm

`Sobel` (default) | `Prewitt` | `Roberts`

When you select `Sobel` or `Prewitt`, the block calculates horizontal and vertical gradients, `Gv` and `Gh`. When you select `Roberts`, the block calculates orthogonal gradients, `G45` and `G135`. For details of each method, see "Algorithms" on page 1-58.

Note If you select `Prewitt`, the full-precision internal data type is large due to the 1/6 coefficient. Consider selecting **Output the gradient components**, so that you can customize the data type to a smaller size.

Output the binary image — Enable edge output port

on (default) | off

When this parameter is selected, the block returns a stream of binary pixels representing the edges detected in the input frame.

You must select at least one of **Output the binary image** and **Output the gradient components**.

Output the gradient components — Enable gradient output ports

off (default) | on

When this parameter is selected, the block returns a stream of values representing the gradients calculated in the two orthogonal directions at each pixel. When you set **Method** to `Sobel` or `Prewitt`, the output ports **Gv** and **Gh** appear on the block. When you set **Method** to `Roberts`, the output ports **G45** and **G135** appear on the block.

You must select at least one of **Output the binary image** and **Output the gradient components**.

Source of threshold value — Source for gradient threshold that indicates an edge

Property (default) | Input port

You can set the threshold from an input port or from the dialog box. The default value is `Property`. Selecting `Input port` enables the **Th** port.

Threshold value — Gradient threshold value that indicates an edge

20 (default) | scalar

The block compares the square of this value to the sum of the squares of the gradients. The block casts this value to the data type of the gradients.

Dependencies

This option is visible when you set **Source of threshold value** to `Property`.

Line buffer size — Size of the line memory buffer

2048 (default) | integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (N – 1)-by-**Line buffer size** memory locations to store the pixels, where N is the number of lines in the differential approximation matrix. If you set **Method** to `Sobel` or `Prewitt`, then N is 3. If you set **Method** to `Roberts`, then N is 2.

Padding method — Method for padding

Symmetric (default) | None

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- `Symmetric` — Set the value of the padding pixels to mirror the edge of the image. This option prevents edges from being detected at the boundaries of the active frame.
- `None` — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

Data Types

Rounding mode — Rounding method for internal fixed-point calculations

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

Specify a rounding method for internal fixed-point calculations.

Saturate on integer overflow — Overflow action for internal fixed-point calculations

off (default) | on

When you clear this parameter, fixed-point and integer values wrap around to zero when the value overflows what is representable with that data type. When you select this parameter, the value saturates at the maximum representable value.

Gradient Data Type — Data type for gradient output ports

Inherit via internal rule (default) | data type expression

Data type for the two gradient output ports. By default, the block automatically chooses full-precision data types.

Dependencies

To enable this parameter, on the **Main** tab, select **Output the gradient components**.

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

The Edge Detector block provides three methods for detecting edges in an input image. The methods use different derivative approximation matrices to find two orthogonal gradients. The Sobel and Prewitt methods calculate the gradient in horizontal and vertical directions. The Roberts method calculates the gradients at 45 degrees and 135 degrees. The block uses the same matrices as the Edge Detection block in Computer Vision Toolbox.

When you use multipixel streaming, the block uses a single line memory and implements one filter for each of the M input pixels, in parallel. This increase in hardware resources is a trade off for increasing throughput compared to single-pixel streaming.

Method	Direction 1	Direction 2
Sobel	$\frac{1}{8}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$	$\frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$

Method	Direction 1	Direction 2
Prewitt	$\frac{1}{6}\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$	$\frac{1}{6}\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$
Roberts	$\frac{1}{2}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\frac{1}{2}\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$

Note The Prewitt coefficients require extra bits of precision because they are not powers of two. The block uses 16 bits to represent the Prewitt coefficients. For 8-bit input, the default size of the full-precision gradients is 27 bits. When using the Prewitt method, a good practice is to reduce the word length used for the gradient calculation. Select the **Output the gradient components** check box, and then on the **Data Types** tab, specify a smaller word length using **Gradient Data Type**.

The block convolves the neighborhood of the input pixel with the derivative matrices, D1 and D2. It then compares the sum of the squares of the gradients to the square of the threshold. Computing the square of the threshold avoids constructing a square root circuit. The block casts the gradients to the type you specified on the **Data Types** tab. The type conversion on the square of the threshold matches the type of the sum of the squares of the gradients.

Latency

The latency of the block is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding by default. The latency of the padding operation depends on the size of the kernel. If edge padding is not necessary for your design, you can reduce the latency by setting the **Padding method** parameter to None. When you use this option, the block latency does not depend on your kernel size. To determine the exact latency for any configuration of the block, measure the number of time steps between the input and output control signals.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch** + **Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2015a

R2023a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2020a: Option to omit padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

R2019b: Multipixel streaming

The Edge Detector block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

For multipixel streaming, the block supports input and output column vectors of 4 or 8 pixels. The **ctrl** port remains scalar, and the control signals in the pixelcontrol bus apply to all pixels in the vector.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your model. A best practice is to synchronize parallel paths in your model by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 for this block inside a Resettable Synchronous Subsystem.
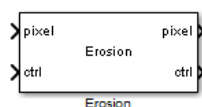
See Also

Blocks
Edge Detection | Frame To Pixels

Objects
visionhdl.EdgeDetector

Erosion

Morphological erosion of binary pixel data

Libraries:
Vision HDL Toolbox / Morphological Operations

Description

The Erosion block replaces each pixel with the local minimum of the neighborhood around the pixel. The block operates on a stream of binary intensity values. You can specify a neighborhood or structuring element of up to 32-by-32 pixels.

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Ports

Input

pixel — Boolean input pixel or multipixel vector
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar Boolean value. For multipixel streaming, specify a vector of four or eight Boolean values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

Data Types: `Boolean`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Output

pixel — Boolean output pixel or multipixel vector
scalar | vector

This block supports single pixel streaming or multipixel streaming. When using single pixel streaming, the block returns a single pixel as a scalar Boolean value. When using multipixel streaming, the block returns a vector of Boolean values. This vector is the same size as the input **pixel** vector. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

Data Types: `Boolean`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Parameters

Neighborhood — Pixel neighborhood

`ones(3,3)` (default) | vector or matrix of 1s and 0s

Pixel neighborhood, specified as a vector or matrix of binary values.

The block supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the **Neighborhood** parameter to `getnhood(strel(`*shape*`))`, where *shape* is specified by the input arguments to the `strel` function..

When you use multipixel vector input, the neighborhood must be at least two pixels in each dimension.

Line buffer size — Size of line memory buffer

`2048` (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (n – 1)-by-**Line buffer size** memory locations to store the pixels, where n is the number of lines in the **Neighborhood** parameter value.

Padding method — Method for padding

`Constant` (default) | `None`

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- `Constant` — The block pads the image with ones. This value prevents erosion at the boundaries of the active frame.
- `None` — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

Latency

The latency of the block is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding by default. The latency of the padding operation depends on the size of the kernel. If edge padding is not necessary for your design, you can reduce the latency by setting the **Padding method** parameter to `None`. When you use this option, the block latency does not depend on your kernel size. To determine the exact latency for any configuration of the block, measure the number of time steps between the input and output control signals.

The latency of the kernel depends on the neighborhood size.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch + Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2015a

R2020a: Option to omit padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

R2020a: Multipixel streaming

The Erosion block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

For multipixel streaming, the blocks support input and output column vectors of 4 or 8 pixels. The **ctrl** port remains scalar, and the control signals in the pixelcontrol bus apply to all pixels in the vector.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 if it is inside a Resettable Synchronous Subsystem.

See Also
Dilation | Frame To Pixels | `visionhdl.Erosion`

Topics
"Types of Morphological Operations" (Image Processing Toolbox)
"Structuring Elements" (Image Processing Toolbox)

FIL Frame To Pixels

Convert frame-based video to pixel stream for FPGA-in-the-loop
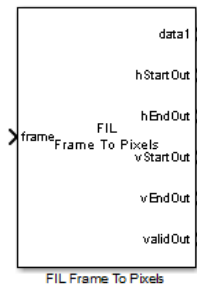
Libraries:
Vision HDL Toolbox / I/O Interfaces

Description

The FIL Frame To Pixels block performs the same frame-to-pixel conversion as the Frame To Pixels block. In addition, you can configure the width of the output vector to be a single pixel, a line, or an entire frame. The block returns control signals in vectors of the same width as the pixel data. This optimization makes more efficient use of the communication link between the FPGA board and your Simulink simulation when using FPGA-in-the-loop (FIL). To run FPGA-in-the-loop, you must have an HDL Verifier™ license.

When you generate a programming file for a FIL target in Simulink, the tool creates a model to compare the FIL simulation with your Simulink design. For Vision HDL Toolbox designs, the FIL block in that model replicates the pixel-streaming interface to send one pixel at a time to the FPGA. You can modify the autogenerated model to use the FIL Frame To Pixels and FIL Pixels To Frame blocks to improve communication bandwidth with the FPGA board by sending one frame at a time. For how to modify the autogenerated model, see "FPGA-in-the-Loop".

Specify the same video format and vector size for the FIL Frames To Pixels block and the FIL Pixels To Frame block.

Ports

Input

frame — Full image frame
matrix

Full image specified as a (**Active pixels per line**)-by-(**Active video lines**)-by-*N* matrix. Height and width are the dimensions of the active image specified in **Video format**. *N* is the **Number of components** used to express a single pixel.

Data Types: `single` | `double` | `int` | `uint` | `Boolean` | `fixed point`

Output

data1,...,dataN — Image pixels
vector | matrix

Image pixels, returned as a matrix of *M*-by-**Number of pixels** values, where *M* is the width of the **Output vector format**. There are *N* data ports, where *N* is the **Number of components**. The data type is the same as the data type of **frame**.

When you use multipixel streaming, the FIL DUT has **Number of pixels** data ports. You must split the output matrix into **Number of pixels** row vectors of length *M* for the FIL DUT input ports. For details, see "FPGA-in-the-Loop Simulation with Multipixel Streaming".

Similarly, when you use multicomponent-multipixel streaming, the FIL DUT has **Number of pixels×Number of components** data ports. Split the output matrix for each component into **Number of pixels** row vectors of length *M* for the FIL DUT input ports.

Data Types: `single` | `double` | `int` | `uint` | `Boolean` | `fixed point`

hStartOut — First pixel in horizontal line of frame
vector

First pixel in a horizontal line of a frame, returned as a `Boolean` vector of *M* values.

For multipixel streaming, **hStartOut** applies to the pixel with the lowest index in the corresponding column.

Data Types: `Boolean`

hEndOut — Last pixel in horizontal line of frame
vector

Last pixel in a horizontal line of a frame, returned as a `Boolean` vector of *M* values.

For multipixel streaming, **hEndOut** applies to the pixel with the highest index in the corresponding column.

Data Types: `Boolean`

vStartOut — First pixel in first (top) line of frame
vector

First pixel in the first (top) line of a frame, returned as a `Boolean` vector of *M* values.

For multipixel streaming, **vStartOut** applies to the pixel with the lowest index in the corresponding column.

Data Types: `Boolean`

vEndOut — Last pixel in last (bottom) line of frame
vector

Last pixel in the last (bottom) line of a frame, returned as a `Boolean` vector of *M* values.

For multipixel streaming, **vEndOut** applies to the pixel with the highest index in the corresponding column.

Data Types: `Boolean`

validOut — Valid pixel indicator
vector

Valid pixel indicator, returned as a `Boolean` vector of *M* values.

For multipixel streaming, the pixels in the corresponding column are either all valid or all invalid.

Data Types: `Boolean`

Parameters

Number of components — Number of values used to represent each pixel

1 (default) | 2 | 3 | 4

Number of values used to represent each pixel. Each pixel can have from 1 to 4 components.

- For grayscale video, set this parameter to `1`.
- For color video, for example, {R,G,B} or {Y,Cb,Cr}, set this parameter to `3`.
- For color video with an alpha channel for transparency, set this parameter to `4`.

To process multicomponent streams for blocks that do not support multicomponent input, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect the **ctrl** output of the Frame To Pixels block to each replicated block.

Number of pixels — Number of pixels in each column of output matrix

1 (default) | 2 | 4 | 8

Number of pixels in each column of the output matrix, specified as 1, 2, 4, or 8.

Each data port returns a *M*-by-**Number of pixels** matrix. You must split this matrix into **Number of pixels** row vectors of length *M* for input to the FIL DUT. For details, see "FPGA-in-the-Loop Simulation with Multipixel Streaming".

Output vector format — Size of vector used to communicate with FPGA subsystem

Frame (default) | Line | Pixel

Size of the vector used to communicate with the FPGA subsystem. The block returns input pixels and control signals in vectors of the same length. The block calculates the length of the vectors based on the **Video format** parameter.

- `Pixel` — Return scalar values for pixel and control signals.
- `Line` — Return (**Total pixels per line**)-by-1 vectors.
- `Frame` — Return (**Total pixels per line** × **Total video lines**)-by-1 vectors.

A larger value results in faster communication between the FPGA board and Simulink. Choose the largest option that the I/O and memory resources on your board can support.

For multipixel streaming, specify one of these options for output size.

- `Pixel` — Return a 1-by-(**Number of pixels**) vector of pixel values, and scalar control signals that apply to all the pixels in each vector.
- `Line` — Return a (**Total pixels per line/Number of pixels**)-by-**Number of pixels** matrix of pixel values, and a vector of control signals of length (**Total pixels per line**)/(**Number of pixels**).

- Frame — Return a (**Total pixels per line × Total video lines/Number of pixels**)-by **Number of pixels** matrix of pixel values and a vector of control signals of length (**Total pixels per line × Total video lines**)/**Number of pixels**.

Dependencies

When **Number of pixels** and **Number of components** are both greater than 1, you must set **Output vector format** to Frame.

Video format — Dimensions of active and inactive regions in video frame

1080p (default) | 240p | 480p | 480pH | 576p | 720p | 768p | 1024p | 1200p | 2KCinema | 4KUHDTV | 8KUHDTV | Custom

Dimensions of active and inactive regions in a video frame. To select a predefined format, select from the **Video format** list. The actual frame dimensions are displayed in the **Video Format Parameters** section. For a custom format, select Custom, and then specify the dimensions as integers.

The dimensions are defined in the diagram.

Note The sample time of your video source must match the total number of pixels in the frame of your Frame To Pixels block. The total number of pixels is *Total pixels per line × Total video lines*, so set the sample time to this value.

If your frame size is large, you may reach the fixed-step solver step size limit for sample times in Simulink, and receive an error like this.

```
The computed fixed step size (1.0) is 1000000.0 times smaller than all
the discrete sample times in the model.
```

You can avoid this error by choosing the variable-step solver.

Video Format	Active Pixels Per Line	Active Video Lines	Total Pixels Per Line	Total Video Lines	Starting Active Line	Ending Active Line	Front Porch	Back Porch
240p	320	240	402	324	1	240	44	38
480p	640	480	800	525	36	515	16	144
480pH	720	480	858	525	33	512	16	122
576p	720	576	864	625	47	622	12	132
720p	1280	720	1650	750	25	744	110	260
768p	1024	768	1344	806	10	777	24	296
1024p	1280	1024	1688	1066	42	1065	48	360
1080p (default)	1920	1080	2200	1125	42	1121	88	192
1200p	1600	1200	2160	1250	50	1249	64	496
2KCinema	2048	1080	2750	1125	42	1121	639	63
4KUHDTV	3840	2160	4400	2250	42	2201	88	472
8KUHDTV	7680	4320	8800	4500	42	4361	88	1032
Custom	User-defined	User-defined	User-defined	User-defined	User-defined	User-defined	User-defined	User-defined

When using a custom format, the values you enter for the active and inactive dimensions of the image must add up to the total frame dimensions. If you specify a format that does not conform to these rules, the block reports an error.

- For the horizontal direction, **Total pixels per line** must be greater than or equal to **Front porch** + **Active pixels per line**. The block calculates **Back porch = Total pixels per line − Front porch − Active pixels per line**.

- For the vertical direction, **Total video lines** must be greater than or equal to **Starting active line** + **Active video lines** − 1. The block calculates **Ending active line = Starting active line + Active video lines** − 1.

When using the Line Memory block, or blocks that use an internal line memory, with a custom video format, further requirements apply:

- **Active pixels per line** must be greater than 1.

- The horizontal blanking interval, or **Back porch** + **Front porch**, must meet these guidelines:

  - The total of **Back porch** + **Front porch** must be at least 2 times the largest kernel size of the algorithm in the blocks following the Frame To Pixel block. If the kernel size is less than 4, the total porch must be at least 8 pixels.

  - The **Back porch** must be at least 6 pixels. This parameter is the number of inactive pixels before the first valid pixel in a frame.

For more information on blanking intervals, see "Configure Blanking Intervals".

When using multipixel streaming (**Number of pixels** > 1), these requirements apply.

- The video format must have horizontal dimensions divisible by the **Number of pixels** parameter value. The horizontal dimensions are set by these parameters: **Active pixels per line**, **Total pixels per line**, **Front porch**, and **Back porch**. Standard video protocols 480p, 768p, 1024p, 1080p, 1200p, 4k UHD, and 8k UHD support **Number of pixels** equal to 4 or 8.

- The minimum input frame size for multipixel streaming is 18 rows by 32 columns.

- Choose your kernel size and **Active pixels per line** such that (**Active pixels per line**)/(**Number of pixels**) is at least the kernel width.

Version History

Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The block can now generate multipixel streams that have 2 pixels per clock cycle.

R2020b: Multipixel-multicomponent streaming

The block can now generate multipixel-multicomponent streams. When you use multicomponent-multipixel streaming, the FIL DUT has **Number of pixels×Number of components** data ports. Split the output matrix for each component into **Number of pixels** row vectors of length M for the FIL DUT input ports.

R2019b: Multipixel streaming

The block can now generate multipixel streams. You can set the **Number of pixels** parameter to 4 or 8. With this setting, the output of the block is a vector of 4 or 8 pixel values on each time step. For an example of how to use this feature, see "FPGA-in-the-Loop Simulation with Multipixel Streaming".

Video formats for multipixel streams must have horizontal dimensions divisible by the **Number of pixels** parameter value. These horizontal dimensions are set by the following parameters: **Active pixels per line**, **Total pixels per line**, **Front porch**, and **Back porch**. Standard video protocols 480p, 720p, 1080p, and 4k UHD support either 4 or 8 pixels at a time.

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.

See Also
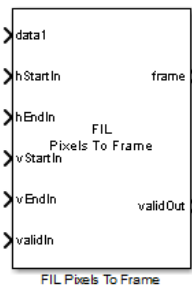
Blocks
FIL Pixels To Frame | Frame To Pixels

Topics
"Streaming Pixel Interface"

"FPGA Verification" (HDL Verifier)

FIL Pixels To Frame

Convert pixel stream from FPGA-in-the-loop to frame-based video

Libraries:
Vision HDL Toolbox / I/O Interfaces

Description

The FIL Pixels To Frame block performs the same pixel-to-frame conversion as the Pixels To Frame block. In addition, you can configure the width of the input to be a single pixel, a line, or an entire frame per step. The block expects control signal input vectors of the same width as the pixel data. This optimization can speed up the communication link between the FPGA board and your Simulink simulation when using FPGA-in-the-loop. To run FPGA-in-the-loop, you must have an HDL Verifier license.

When you generate a programming file for a FIL target in Simulink, the tool creates a model to compare the FIL simulation with your Simulink design. For Vision HDL Toolbox designs, the FIL block in that model replicates the pixel-streaming interface to send one pixel at a time to the FPGA. You can modify the autogenerated model to use the FIL Frame To Pixels and FIL Pixels To Frame blocks to improve communication bandwidth with the FPGA board by sending one frame at a time. For how to modify the autogenerated model, see "FPGA-in-the-Loop".

Specify the same video format for the FIL Frames To Pixels block and the FIL Pixels To Frame block.

Ports

Input

data1,...,dataN — Image pixels
vector | matrix

Image pixels, specified as a matrix of M-by-**Number of pixels** values, where M is the width of the **Output vector format**. There are N data ports, where N is the **Number of components**.

When you use multipixel streaming, the FIL DUT has **Number of pixels** data ports. You must recombine the output vectors from each port into a M-by-**Number of pixels** matrix. For details, see "FPGA-in-the-Loop Simulation with Multipixel Streaming".

Similarly, when you use multicomponent-multipixel streaming, the FIL DUT has **Number of pixels**×**Number of components** data ports. Recombine the output vectors into a M-by-**Number of pixels** matrix for each component.

Data Types: `single` | `double` | `int` | `uint` | `Boolean` | `fixed point`

hStartIn — First pixel in horizontal line of frame
vector

First pixel in a horizontal line of a frame, specified as a `Boolean` vector of *M* values.

For multipixel streaming, **hStartIn** applies to the pixel with the lowest index in the corresponding column.

Data Types: `Boolean`

hEndIn — Last pixel in horizontal line of frame
vector

Last pixel in a horizontal line of a frame, specified as a `Boolean` vector of *M* values.

For multipixel streaming, **hEndIn** applies to the pixel with the highest index in the corresponding column.

Data Types: `Boolean`

vStartIn — First pixel in first (top) line of frame
vector

First pixel in the first (top) line of a frame, specified as a `Boolean` vector of *M* values.

For multipixel streaming, **vStartIn** applies to the pixel with the lowest index in the corresponding column.

Data Types: `Boolean`

vEndIn — Last pixel in last (bottom) line of frame
vector

Last pixel in the last (bottom) line of a frame, specified as a `Boolean` vector of *M* values.

For multipixel streaming, **hEndIn** applies to the pixel with the highest index in the corresponding column.

Data Types: `Boolean`

validIn — Valid pixel indicator
vector

Valid pixel indicator, specified as a `Boolean` vector of *M* values.

For multipixel streaming, the pixels in the corresponding column are either all valid or all invalid.

Data Types: `Boolean`

Output

frame — Full image frame
matrix

Full image specified as a (**Active pixels per line**)-by-(**Active video lines**)-by-*N* matrix. Height and width are the dimensions of the active image specified in **Video format**. *N* is the **Number of components** used to express a single pixel. The data type is the same as **data1,...,dataN**.

validOut — Indicates when output frame is ready
scalar

A `true` (1) value indicates when output frame is successfully recompiled from the input stream.

Data Types: `Boolean`

Parameters

Number of components — Number of values used to represent each pixel

1 (default) | 2 | 3 | 4

Number of values used to represent each pixel. Each pixel can have from 1 to 4 components.

- For grayscale video, set this parameter to 1.
- For color video, for example, {R,G,B} or {Y,Cb,Cr}, set this parameter to 3.
- For color video with an alpha channel for transparency, set this parameter to 4.

To process multicomponent streams for blocks that do not support multicomponent input, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect the **ctrl** output of the Frame To Pixels block to each replicated block.

Number of pixels — Number of pixels in each column of input matrix

1 (default) | 2 | 4 | 8

Number of pixels in each column of the input matrix, specified as 1, 2, 4, or 8.

Each data port expects a *M*-by-**Number of pixels** matrix. You must recombine the FIL DUT output vectors to create this matrix. For details, see "FPGA-in-the-Loop Simulation with Multipixel Streaming".

Input vector format — Size of the vector used to communicate with the FPGA subsystem

Frame (default) | Line | Pixel

Size of the vector used to communicate with the FPGA subsystem. The block accepts input pixels and control signals in vectors of the same length. The block calculates the length of the vectors based on the **Video format** parameter.

- `Pixel` — Accept scalar values for pixel and control signals.
- `Line` — Accept input vectors containing **Total pixels per line** values.
- `Frame` — Accept input vectors containing **Total pixels per line** × **Total video lines** values.

A larger value results in faster communication between the FPGA board and Simulink. Choose the largest option that the I/O and memory resources on your board can support.

For multipixel streaming, specify one of these options for input size.

- `Pixel` — Accept a 1-by-(**Number of pixels**) vector of pixel values, and scalar control signals that apply to all the pixels in each vector.
- `Line` — Accept a (**Total pixels per line/Number of pixels**)-by-**Number of pixels** matrix of pixel values, and a vector of control signals of length (**Total pixels per line**)/(**Number of pixels**).
- `Frame` — Return a (**Total pixels per line × Total video lines/Number of pixels**)-by **Number of pixels** matrix of pixel values and a vector of control signals of length (**Total pixels per line × Total video lines**)/**Number of pixels**.

Dependencies

When **Number of pixels** and **Number of components** are both greater than 1, you must set **Input vector format** to `Frame`.

Video format — Dimensions of active regions of a video frame

`1080p` (default) | `240p` | `480p` | `480pH` | `576p` | `720p` | `768p` | `1024p` | `1200p` | `2KCinema` | `4KUHDTV` | `8KUHDTV` | `Custom`

Dimensions of active regions of a video frame. To select a predefined format, use the **Video format** list. For a custom format, select `Custom`, and then specify the active frame dimensions as integers.

Video Format	Active Pixels Per Line	Active Video Lines
240p	320	240
480p	640	480
480pH	720	480
576p	720	576
720p	1280	720
768p	1024	768
1024p	1280	1024
1080p (default)	1920	1080
1200p	1600	1200
2KCinema	2048	1080
4KUHDTV	3840	2160
8KUHDTV	7680	4320
Custom	User-defined	User-defined

Version History

Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The block can now accept multipixel streams that have 2 pixels per clock cycle.

R2020b: Multipixel-multicomponent streaming

The block can now accept multipixel-multicomponent streams. When you use multi-pixel-multicomponent streaming, the FIL DUT has **Number of pixels**×**Number of components** data ports. Recombine the output vectors into a *M*-by-**Number of pixels** matrix for each component.

R2019b: Multipixel streaming

The block can now accept multipixel streams. You can set the **Number of pixels** parameter to 4 or 8. When you use multipixel streaming, the FIL DUT has **Number of pixels** data ports. You must recombine the output vectors from each port into a *M*-by-**Number of pixels** matrix. For details, see "FPGA-in-the-Loop Simulation with Multipixel Streaming".

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.

See Also

Blocks
FIL Frame To Pixels | Pixels To Frame

Topics
"Streaming Pixel Interface"
"FPGA Verification" (HDL Verifier)

Frame To Pixels

Convert frame-based video to pixel stream

Libraries:
Vision HDL Toolbox / I/O Interfaces

Description

The Frame To Pixels block converts color or grayscale video frames to a pixel stream and control signals. The control signals indicate the validity of each pixel and its location in the frame. The pixel stream format can include padding pixels around the active frame. You can configure the frame and padding dimensions by selecting a common video format or by specifying custom dimensions. The pixel stream can support single pixel streaming or multipixel streaming. Multipixel streaming provides 2, 4, or 8 pixels per clock cycle to support high-rate or high-resolution formats. For details of the pixel stream format, see "Streaming Pixel Interface".

This block itself does not support HDL code generation. However, you can use this block to generate input for a separate subsystem targeted for HDL code generation.

If your model converts frames to a pixel stream and later converts the stream back to frames, specify the same video format for the Frame To Pixels block and the Pixels To Frame block.

Ports

Input

frame — Full image frame
array

Full image frame, specified as an **Active pixels per line**-by-**Active video lines**-by-*N* array. The first two array dimensions are the height and width of the active image specified by the **Video format** parameter. *N* is the **Number of components** used to express a single pixel. **Number of components** can be from one to four.

Data Types: single | double | int | uint | Boolean | fixed point

Output

pixel — Image pixel or pixels
scalar | vector

For single pixel streams, **pixel** is a single image pixel returned as a scalar or a vector of 1-by-**Number of components** values. For multipixel streams, **pixel** is a matrix of **Number of pixels**-by-**Number of components** pixel intensity values.

The output data type is the same as the data type of the **frame** port.

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Parameters

Number of components — Number of values used to represent each pixel

1 (default) | 2 | 3 | 4

Number of values used to represent each pixel. Each pixel can have from 1 to 4 components.

- For grayscale video, set this parameter to 1.
- For color video, for example, {R,G,B} or {Y,Cb,Cr}, set this parameter to 3.
- For color video with an alpha channel for transparency, set this parameter to 4.

To process multicomponent streams for blocks that do not support multicomponent input, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect the **ctrl** output of the Frame To Pixels block to each replicated block.

Number of pixels — Number of pixels streamed per time step

1 (default) | 2 | 4 | 8

Number of pixels transferred on the streaming interface for each time step, specified as 1, 2, 4, or 8. To enable multipixel streaming and increase throughput for high-resolution or high-frame-rate video, set this parameter to 2, 4 or 8. Multipixel streaming processes more pixels with the same clock frequency as a single-pixel streaming interface.

Video Resolution	Clock Frequency Required for Single-Pixel Streaming at 60 fps	Clock Frequency Required for Multipixel Streaming at 60 fps (4 pixels)
1080p	150 MHz	37.5 MHz
4k UHD	600 MHz	150 MHz
8k UHD	1200 MHz	300 MHz

Video format — Dimensions of active and inactive regions in video frame

1080p (default) | 240p | 480p | 480pH | 576p | 720p | 768p | 1024p | 1200p | 2KCinema | 4KUHDTV | 8KUHDTV | Custom

Dimensions of active and inactive regions in a video frame. To select a predefined format, select from the **Video format** list. The actual frame dimensions are displayed in the **Video Format Parameters** section. For a custom format, select `Custom`, and then specify the dimensions as integers.

The dimensions are defined in the diagram.

Note The sample time of your video source must match the total number of pixels in the frame of your Frame To Pixels block. The total number of pixels is *Total pixels per line × Total video lines*, so set the sample time to this value.

If your frame size is large, you may reach the fixed-step solver step size limit for sample times in Simulink, and receive an error like this.

```
The computed fixed step size (1.0) is 1000000.0 times smaller than all
the discrete sample times in the model.
```

You can avoid this error by choosing the variable-step solver.

Video Format	Active Pixels Per Line	Active Video Lines	Total Pixels Per Line	Total Video Lines	Starting Active Line	Ending Active Line	Front Porch	Back Porch
240p	320	240	402	324	1	240	44	38
480p	640	480	800	525	36	515	16	144
480pH	720	480	858	525	33	512	16	122
576p	720	576	864	625	47	622	12	132
720p	1280	720	1650	750	25	744	110	260

Video Format	Active Pixels Per Line	Active Video Lines	Total Pixels Per Line	Total Video Lines	Starting Active Line	Ending Active Line	Front Porch	Back Porch
768p	1024	768	1344	806	10	777	24	296
1024p	1280	1024	1688	1066	42	1065	48	360
1080p (default)	1920	1080	2200	1125	42	1121	88	192
1200p	1600	1200	2160	1250	50	1249	64	496
2KCinema	2048	1080	2750	1125	42	1121	639	63
4KUHDTV	3840	2160	4400	2250	42	2201	88	472
8KUHDTV	7680	4320	8800	4500	42	4361	88	1032
Custom	User-defined	User-defined	User-defined	User-defined	User-defined	User-defined	User-defined	User-defined

When using a custom format, the values you enter for the active and inactive dimensions of the image must add up to the total frame dimensions. If you specify a format that does not conform to these rules, the block reports an error.

- For the horizontal direction, **Total pixels per line** must be greater than or equal to **Front porch** + **Active pixels per line**. The block calculates **Back porch = Total pixels per line − Front porch − Active pixels per line**.

- For the vertical direction, **Total video lines** must be greater than or equal to **Starting active line** + **Active video lines** − 1. The block calculates **Ending active line = Starting active line + Active video lines** − 1.

When using the Line Memory block, or blocks that use an internal line memory, with a custom video format, further requirements apply:

- **Active pixels per line** must be greater than 1.

- The horizontal blanking interval, or **Back porch** + **Front porch**, must meet these guidelines:

  - The total of **Back porch** + **Front porch** must be at least 2 times the largest kernel size of the algorithm in the blocks following the Frame To Pixel block. If the kernel size is less than 4, the total porch must be at least 8 pixels.

  - The **Back porch** must be at least 6 pixels. This parameter is the number of inactive pixels before the first valid pixel in a frame.

  For more information on blanking intervals, see "Configure Blanking Intervals".

When using multipixel streaming (**Number of pixels** > 1), these requirements apply.

- The video format must have horizontal dimensions divisible by the **Number of pixels** parameter value. The horizontal dimensions are set by these parameters: **Active pixels per line**, **Total pixels per line**, **Front porch**, and **Back porch**. Standard video protocols 480p, 768p, 1024p, 1080p, 1200p, 4k UHD, and 8k UHD support **Number of pixels** equal to 4 or 8.

- The minimum input frame size for multipixel streaming is 18 rows by 32 columns.

- Choose your kernel size and **Active pixels per line** such that (**Active pixels per line**)/(**Number of pixels**) is at least the kernel width.

Version History
Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The block can now generate multipixel streams that have 2 pixels per clock cycle.

R2020a: Multipixel-multicomponent streaming

The block can now generate multipixel-multicomponent streams. The "MultiPixel-MultiComponent Video Streaming" example shows how to use a multipixel and multicomponent pixel stream to process a high-resolution color image.

R2019b: Multipixel streaming

The block can now generate multipixel streams. You can set the **Number of pixels** parameter to 4 or 8. With this setting, the output of the block is a vector of 4 or 8 pixel values on each time step. For an example of how to use this feature, see "Filter Multipixel Video Streams".

Video formats for multipixel streams must have horizontal dimensions divisible by the **Number of pixels** parameter value. These horizontal dimensions are set by the following parameters: **Active pixels per line**, **Total pixels per line**, **Front porch**, and **Back porch**. Standard video protocols 480p, 720p, 1080p, and 4k UHD support either 4 or 8 pixels at a time.

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.
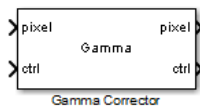
See Also

Blocks
Pixels To Frame

Objects
visionhdl.FrameToPixels

Topics
"Streaming Pixel Interface"

Gamma Corrector

Apply or remove gamma correction of pixel stream

Libraries:
Vision HDL Toolbox / Conversions

Description

Gamma Corrector applies or removes gamma correction on a stream of pixels. Gamma correction adjusts linear pixel values so that the modified values fit a curve. The de-gamma operation performs the opposite operation to obtain linear pixel values.

Ports

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar | vector

For single pixel streams, specify **pixel** as a scalar intensity value. For multipixel streams, specify **pixel** as a vector of **Number of pixels**-by-1 pixel intensity values. **Number of pixels** can be two, four, or eight. Integer and fixed-point data types larger than 16 bits are not supported.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The

`hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Output

pixel — Output pixel stream
scalar | vector

Image pixel, returned as a scalar value or vector of values. The data type and dimension of the output **pixel** stream match those of the input **pixel** stream.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Parameters

Operation — Direction of pixel value adjustment

Gamma (default) | `De-gamma`

Direction of pixel value adjustment, specified as one of these options:

- `Gamma` — Apply gamma correction.
- `De-gamma` — Remove gamma correction.

Gamma — Target gamma value

`2.2` (default) | scalar greater than or equal to 1

Target gamma value, specified as a scalar greater than or equal to 1.

- When you set **Operation** to `Gamma`, **Gamma** is the target gamma value of the output video stream.
- When you set **Operation** to `De-gamma`, **Gamma** is the gamma value of the input video stream.

Linear segment — Include a linear segment in the gamma curve

on (default) | off

Option to include a linear segment in the gamma curve. When you select this check box, the gamma curve has a linear portion near the origin.

Break point — Point where gamma curve and linear segment meet

`0.018` (default) | scalar between 0 and 1

Pixel value that corresponds to the point where the gamma curve and linear segment meet, specified as a scalar value between 0 and 1, exclusive.

Dependencies

This parameter applies only when you select the **Linear segment** check box.

Algorithms

For the equations used for gamma correction, see Gamma Correction in the Computer Vision Toolbox documentation.

To save hardware resources, the block implements the gamma correction equation as a lookup table. The lookup table maps each input pixel value to a corrected output value.

Latency

The latency of the Gamma Corrector block is 2 cycles.

Version History
Introduced in R2015a

R2022b: Multipixel streaming

The block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

For multipixel streaming, the block supports input and output column vectors of 2, 4, or 8 pixels. The **ctrl** port remains scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the vector.

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 one default HDL architecture.

HDL Block Properties

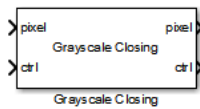|	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline 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).
LUTRegisterResetType	The reset type of the lookup table output register. Select none to synthesize the lookup table to a ROM when your target is an FPGA. See also "LUTRegisterResetType" (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
Gamma Correction | Frame To Pixels

Objects
visionhdl.GammaCorrector

Grayscale Closing

Morphological closing of grayscale pixel data

Libraries:
Vision HDL Toolbox / Morphological Operations

Description

The Grayscale Closing block performs morphological dilation followed by morphological erosion by using the same neighborhood for both calculations. The block operates on a stream of pixel intensity values. You can specify a neighborhood or structuring element of up to 32-by-32 pixels.

For line, square, or rectangle structuring elements more than 8 pixels wide, the block uses the Van Herk algorithm to find the maximum and minimum pixel values. For structuring elements less than 8 pixels wide, or that contain zero elements, the block implements a pipelined comparison tree to find the maximum and minimum pixel values.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Single image pixel
scalar

Single image pixel in a pixel stream, specified as an unsigned integer scalar that represents grayscale intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `uint32` | `uint64` | `double` | `single` | `fi(0,N,0)`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Output

pixel — Single output pixel
scalar

Single output pixel transformed by a morphological operation, returned as a scalar value.

The data type of this output **pixel** is the same as the data type of the input **pixel**.

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

Parameters

Neighborhood — Pixel neighborhood

ones(9,9) (default) | vector or matrix of 1s and 0s

Pixel neighborhood, specified as a vector or matrix of binary values.

The block supports flat neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the **Neighborhood** parameter to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function. The minimum neighborhood size is a 2-by-2 matrix, or a 2-by-1 column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

Line buffer size — Size of line memory buffer

2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (n – 1)-by-**Line buffer size** memory locations to store the pixels, where n is the number of lines in the **Neighborhood** parameter value.

Tips

*   When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".
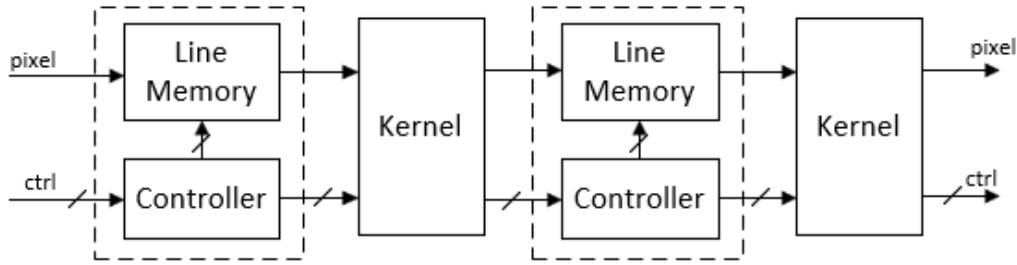
Algorithms

The closing algorithm is morphological dilation followed by morphological erosion. For the respective kernel architectures, see the Grayscale Dilation and Grayscale Erosion reference pages.

The block pads the image with zeros for the dilation operation and with ones for the erosion operation. For more information, see "Edge Padding".

Latency

The total latency of the block is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding. Morphological closing is a compound operation, so this block contains a second line buffer between the dilation kernel and the erosion kernel. To determine the exact latency for any configuration of the block, monitor the number of time steps between the input control signals and the output control signals.

The latency of a Van Herk kernel for a neighborhood of m-by-n pixels is $2m + \log_2(n)$. The block implements this kernel for line, square, or rectangle structuring elements more than 8 pixels wide, with no pixels set to zero.

The latency of a comparison tree kernel for a neighborhood of m-by-n pixels is $\log_2(m)+\log_2(n)$.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch** + **Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2016a

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 if it is inside a Resettable Synchronous Subsystem.

See Also
Grayscale Dilation | Grayscale Erosion | Closing | Frame To Pixels | `visionhdl.GrayscaleClosing`
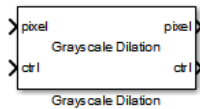
Topics
"Types of Morphological Operations" (Image Processing Toolbox)

"Structuring Elements" (Image Processing Toolbox)

Grayscale Dilation

Morphological dilation of grayscale pixel data

Libraries:
Vision HDL Toolbox / Morphological Operations

Description

The Grayscale Dilation block performs morphological dilation on a stream of pixel intensity values. You can specify a neighborhood or structuring element of up to 32-by-32 pixels. For line, square, or rectangle structuring elements more than 8 pixels wide, the block uses the Van Herk algorithm to find the maximum pixel value. This algorithm uses only three comparators to find the maximums of all the rows, and then uses a comparison tree to find the maximum pixel value of the row results.

For structuring elements less than 8 pixels wide, or that contain zero elements, the block implements a pipelined comparison tree for each row of the neighborhood. An additional comparison tree finds the maximum pixel value of the row results. If the structuring element contains zeros that exclude pixels, the algorithm saves hardware resources by not implementing comparators for those pixel locations.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Single image pixel
scalar

Single image pixel in a pixel stream, specified as an unsigned integer scalar that represents grayscale intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `uint32` | `uint64` | `double` | `single` | `fi(0,N,0)`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

Output

pixel — Single output pixel
scalar

Single output pixel transformed by a morphological operation, returned as a scalar value.

The data type of this output **pixel** is the same as the data type of the input **pixel**.

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

Parameters

Neighborhood — Pixel neighborhood

ones(7,7) (default) | vector or matrix of 1s and 0s

Pixel neighborhood, specified as a vector or matrix of binary values.

The block supports flat neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the **Neighborhood** parameter to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function. The minimum neighborhood size is a 2-by-2 matrix, or a 2-by-1 column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

Line buffer size — Size of line memory buffer

2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (*n* – 1)-by-**Line buffer size** memory locations to store the pixels, where *n* is the number of lines in the **Neighborhood** parameter value.

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

The block pads the image with zeros for the dilation operation. For more information, see "Edge Padding".

Van Herk Implementation

For line, square, or rectangle structuring elements more than 8 pixels wide, the block implements a Van Herk algorithm. All pixels in the structuring element must be set to one. The block decomposes the structuring element into rows and serially finds the maximum pixel value in each row by using the Van Herk algorithm. If the size of the input frame is not a multiple of $m$ pixels, the line memory also adds horizontal padding to a multiple of m. This implementation uses only three comparators total for all rows. Then, if there is more than one row, it calculates the maximum pixel value of the row results by using a comparison tree. The diagram indicates the latency of each computation block.
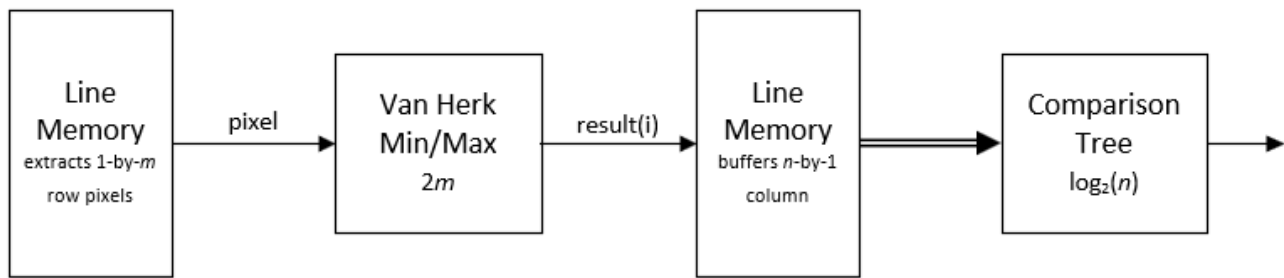
The Van Herk kernel computes a running forward maximum and a running backward maximum of the pixel values in each row of the neighborhood. For this computation, the pixels in the row must be buffered and the order reversed. The buffer adds latency relative to the comparison tree implementation. The Mirror Buffer is a ping-pong RAM of m pixels, where one memory reads values in reverse order while the other memory is writing. The kernel uses $3+n-1$ comparators.
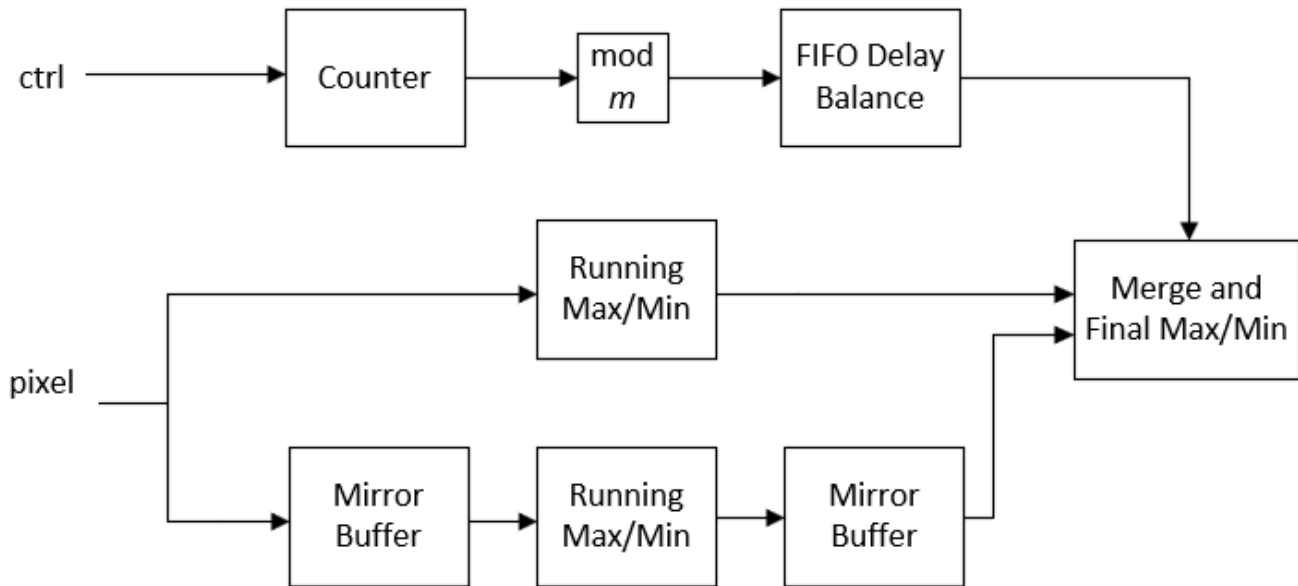
Comparison Tree Implementation

For structuring elements smaller than 8 pixels wide or with one or more pixels set to zero, the block implements a comparison tree.

The diagram shows the architecture of the dilation operation. The algorithm finds the maximum pixel value of each row of the neighborhood in parallel. Then it calculates the maximum pixel value of the rows using another comparison tree. The diagram indicates the latency of each computation block.
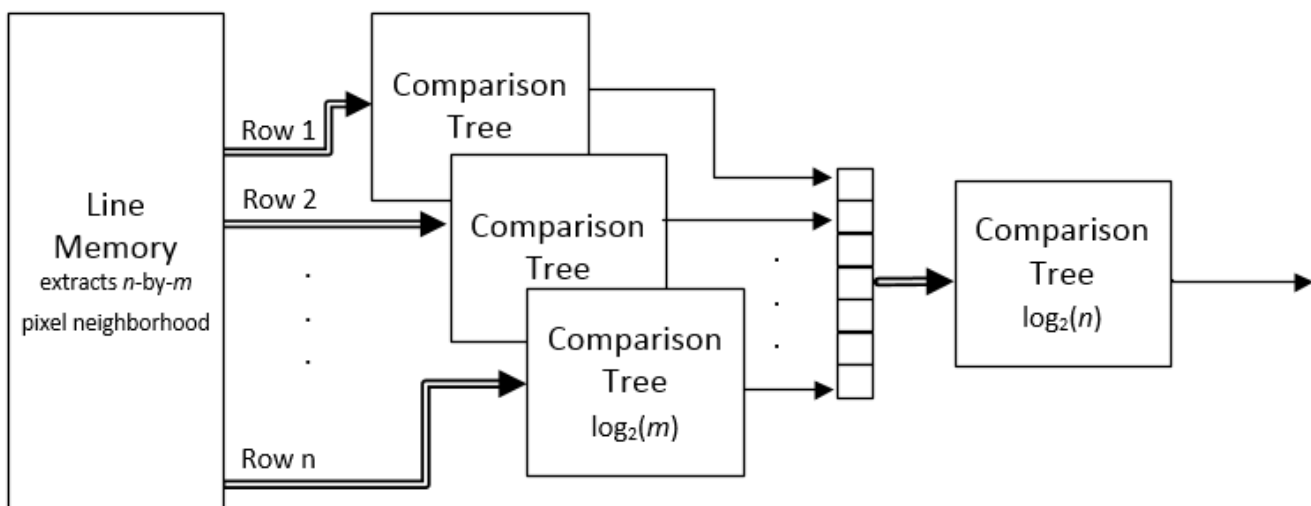
For a rectangular neighborhood that is m pixels wide, the first-stage comparison trees contain $m - 1$ comparators over $\log_2(m)$ clock cycles. For instance, for a rectangular neighborhood that is 7 pixels wide, the comparison tree has six comparators over 3 clock cycles.
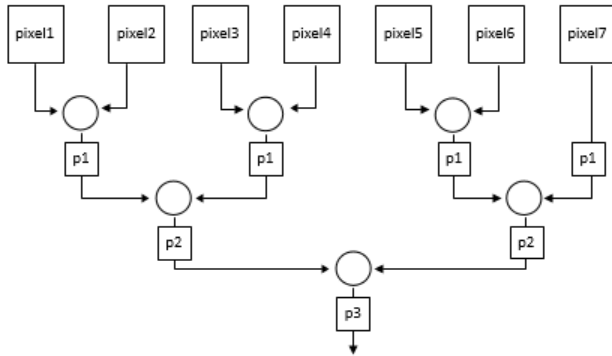
If the neighborhood that you specify contains zeroes, the generated HDL excludes the comparator for the zero locations. The pipeline delay through the comparison tree does not change. For instance, for a nonrectangular neighborhood with a row of [0 0 1 1 0 0 1], the comparison tree for that row contains two comparators and still uses 3 clock cycles.

Latency

The latency of the operation is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding.

The latency of a Van Herk kernel for a neighborhood of m-by-n pixels is $2m + \log_2(n)$. The block implements this kernel for line, square, or rectangle structuring elements more than 8 pixels wide, with no pixels set to zero.

The latency of a comparison tree kernel for a neighborhood of m-by-n pixels is $\log_2(m)+\log_2(n)$. The block implements this kernel for structuring elements smaller than 8 pixels wide or with one or more pixels set to zero.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch** + **Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2016a

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 if it is inside a Resettable Synchronous Subsystem.

See Also

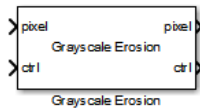Dilation | Grayscale Erosion | Frame To Pixels | `visionhdl.GrayscaleDilation`

Topics
"Types of Morphological Operations" (Image Processing Toolbox)
"Structuring Elements" (Image Processing Toolbox)

Grayscale Erosion

Morphological erosion of grayscale pixel data

Libraries:
Vision HDL Toolbox / Morphological Operations

Description

The Grayscale Erosion block performs morphological erosion on a stream of pixel intensity values. You can specify a neighborhood or structuring element of up to 32-by-32 pixels. For line, square, or rectangle structuring elements more than 8 pixels wide, the block uses the Van Herk algorithm to find the minimum pixel value. This algorithm uses only three comparators to find the minimum pixel values of all the rows, then uses a comparison tree to find the minimum pixel value of the row results.

For structuring elements less than 8 pixels wide, or that contain zero elements, the block implements a pipelined comparison tree for each row of the neighborhood. An additional comparison tree finds the minimum pixel value of the row results. If the structuring element contains zeros that mask off pixels, the algorithm saves hardware resources by not implementing comparators for those pixel locations.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Single image pixel
scalar

Single image pixel in a pixel stream, specified as an unsigned integer scalar that represents grayscale intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `uint32` | `uint64` | `double` | `single` | `fi(0,N,0)`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

Output

pixel — Single output pixel
scalar

Single output pixel transformed by a morphological operation, returned as a scalar value.

The data type of this output **pixel** is the same as the data type of the input **pixel**.

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

Parameters

Neighborhood — Pixel neighborhood

ones(3,3) (default) | vector or matrix of 1s and 0s

Pixel neighborhood, specified as a vector or matrix of binary values.

The block supports flat neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the **Neighborhood** parameter to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function. The minimum neighborhood size is a 2-by-2 matrix, or a 2-by-1 column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

Line buffer size — Size of line memory buffer

2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (n – 1)-by-**Line buffer size** memory locations to store the pixels, where n is the number of lines in the **Neighborhood** parameter value.

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

The block pads the image with ones for the erosion operation. For more information, see "Edge Padding".

Van Herk Implementation

For line, square, or rectangle structuring elements more than 8 pixels wide, the block implements a Van Herk algorithm. All pixels in the structuring element must be set to one. The block decomposes the structuring element into rows and serially finds the minimum pixel value of each row by using the Van Herk algorithm. If the size of the input frame is not a multiple of m pixels, the line memory also adds horizontal padding to a multiple of m. This implementation uses only three comparators total for all rows. Then, if there is more than one row, it calculates the minimum pixel value of the row results by using a comparison tree. The diagram indicates the latency of each computation block.
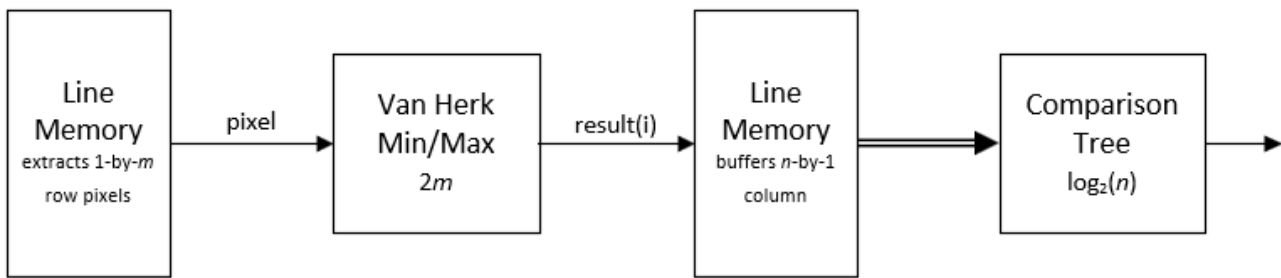
The Van Herk kernel computes a running forward minimum and a running backward minimum of the pixel values in each row of the neighborhood. For this computation, the pixels in the row must be buffered and the order reversed. The buffer adds latency relative to the comparison tree implementation. The Mirror Buffer is a ping-pong RAM of m pixels, where one memory reads values in reverse order while the other memory is writing. The kernel uses $3+n$-1 comparators.
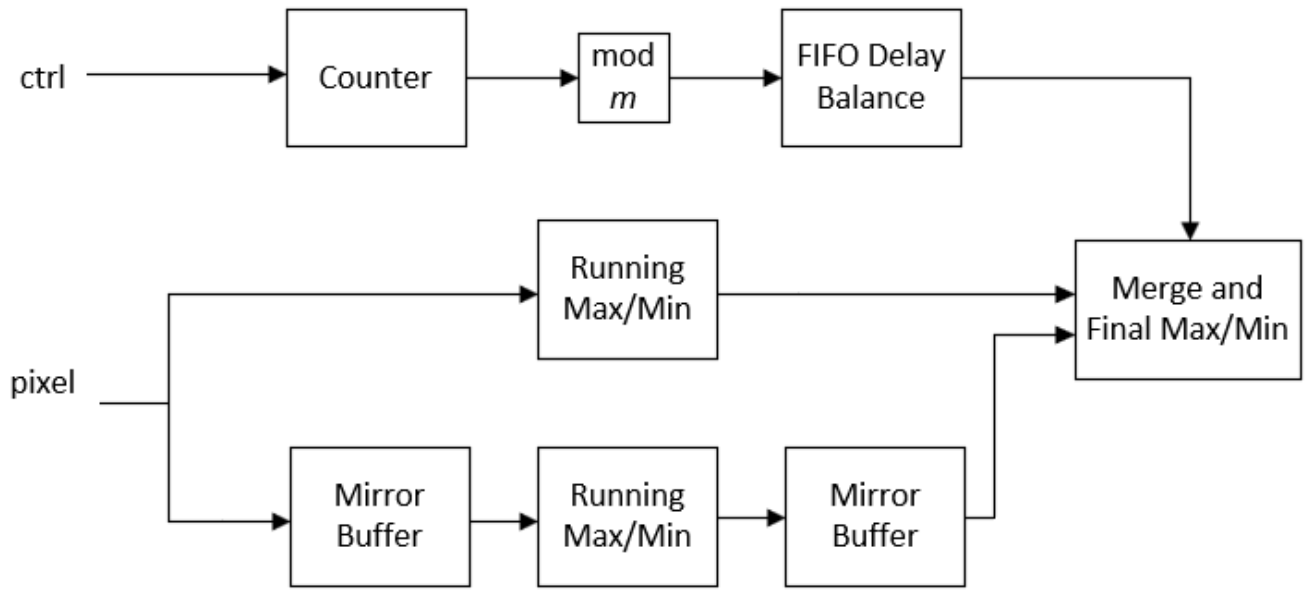
Comparison Tree Implementation

For structuring elements smaller than 8 pixels wide or with one or more pixels set to zero, the block implements a comparison tree.

The diagram shows the architecture of the erosion operation. The algorithm finds the minimum pixel value of each row of the neighborhood in parallel. Then it calculates the minimum pixel value of the rows using another comparison tree. The diagram indicates the latency of each computation block.
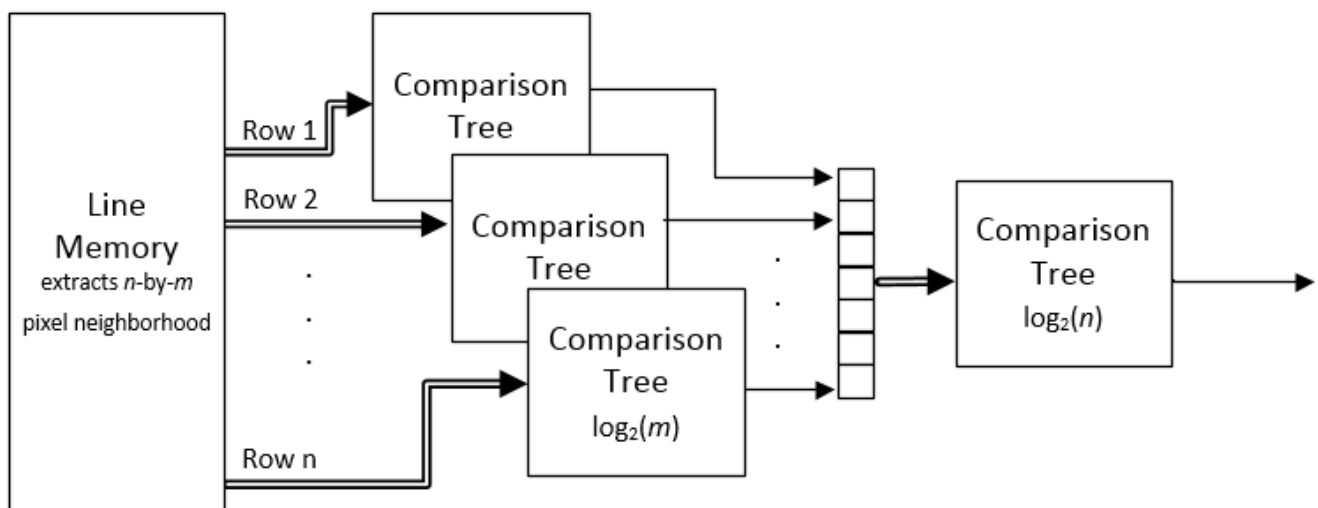
For a rectangular neighborhood that is m pixels wide, the first-stage comparison trees contain $m-1$ comparators over $\log_2(m)$ clock cycles. For instance, for a rectangular neighborhood that is 7 pixels wide, the comparison tree has six comparators over 3 clock cycles.
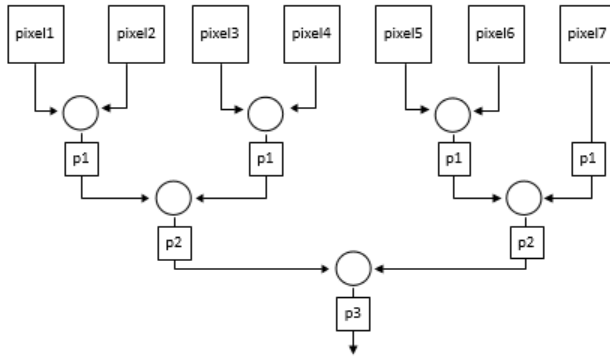
If the neighborhood you specify contains zeroes, the generated HDL excludes the comparator for the zero locations. The pipeline delay through the comparison tree does not change. For instance, for a nonrectangular neighborhood with a row of [0 0 1 1 0 0 1], the comparison tree for that row contains two comparators and still uses 3 clock cycles.

Latency

The latency of the operation is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding.

The latency of a Van Herk kernel for a neighborhood of m-by-n pixels is $2m + \log_2(n)$. The block implements this kernel for line, square, or rectangle structuring elements more than 8 pixels wide, with no pixels set to zero.

The latency of a comparison tree kernel for a neighborhood of m-by-n pixels is $\log_2(m)+\log_2(n)$. The block implements this kernel for structuring elements smaller than 8 pixels wide or with one or more pixels set to zero.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch** + **Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2016a

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 if it is inside a Resettable Synchronous Subsystem.

See Also

Erosion | Grayscale Dilation | Frame To Pixels | `visionhdl.GrayscaleErosion`

Topics
"Types of Morphological Operations" (Image Processing Toolbox)
"Structuring Elements" (Image Processing Toolbox)

Grayscale Opening

Morphological opening of grayscale pixel data

Libraries:
Vision HDL Toolbox / Morphological Operations

Description

The Grayscale Opening block performs morphological erosion followed by morphological dilation by using the same neighborhood for both calculations. The block operates on a stream of pixel intensity values. You can specify a neighborhood, or structuring element, of up to 32-by-32 pixels. For line, square, or rectangle structuring elements more than 8 pixels wide, the block uses the Van Herk algorithm to find the maximum and minimum pixel values. For structuring elements less than 8 pixels wide, or that contain zero elements, the block implements a pipelined comparison tree to find the maximum and minimum pixel values.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Single image pixel
scalar

Single image pixel in a pixel stream, specified as an unsigned integer scalar that represents grayscale intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `uint32` | `uint64` | `double` | `single` | `fi(0,N,0)`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Output

pixel — Single output pixel
scalar

Single output pixel transformed by a morphological operation, returned as a scalar value.

The data type of this output **pixel** is the same as the data type of the input **pixel**.

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

Parameters

Neighborhood — Pixel neighborhood

ones(5,5) (default) | vector or matrix of 1s and 0s

Pixel neighborhood, specified as a vector or matrix of binary values.

The block supports flat neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the **Neighborhood** parameter to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function. The minimum neighborhood size is a 2-by-2 matrix, or a 2-by-1 column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

Line buffer size — Size of line memory buffer

2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (n – 1)-by-**Line buffer size** memory locations to store the pixels, where n is the number of lines in the **Neighborhood** parameter value.

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

The opening algorithm is morphological erosion followed by morphological dilation. For the respective kernel architectures, see the Grayscale Erosion and Grayscale Dilation reference pages.

The line memory pads the image with zeros for the dilation operation and with ones for the erosion operation. For more information, see "Edge Padding".

Latency

The total latency of the block is the line buffer latency plus the latency of the kernel calculation. The latency of the line memory includes edge padding. Morphological opening is a compound operation, so this block contains a second line buffer between the erosion kernel and the dilation kernel. To determine the exact latency for any configuration of the block, monitor the number of time steps between the input control signals and the output control signals.

The latency of a Van Herk kernel for a neighborhood of m-by-n pixels is $2m + \log_2(n)$. The block implements this kernel for line, square, or rectangle structuring elements more than 8 pixels wide, with no pixels set to zero.

The latency of a comparison tree kernel for a neighborhood of m-by-n pixels is $\log_2(m)+\log_2(n)$.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch** + **Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2016a

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 if it is inside a Resettable Synchronous Subsystem.

See Also
Opening | Grayscale Dilation | Grayscale Erosion | Frame To Pixels |
visionhdl.GrayscaleOpening

Topics
"Types of Morphological Operations" (Image Processing Toolbox)
"Structuring Elements" (Image Processing Toolbox)

Histogram

Frequency distribution of pixel values in video stream
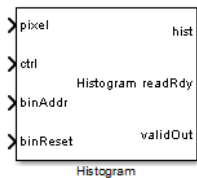
Libraries:
Vision HDL Toolbox / Statistics

Description

The Histogram block computes the frequency distribution of pixel values in a video stream. You can configure the number and size of the bins. The block provides a read interface for accessing each bin. The block keeps a running histogram until you reset the bin values.

Ports

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar intensity value. For multipixel streaming, specify a vector of two, four, or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

This block does not support multicomponent streaming. To process multicomponent streams, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect a single bus to multiple replicated blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `fixdt(0,W,0)` | `Boolean` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

binAddr — Bin number for reading histogram values
integer

The block captures this value each cycle that the **readRdy** output port is 1 (`true`). The data type is `fixdt(0,W,0)`, W = 5,6,...,12. The word length must be $\log_2$(**Number of bins**).

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

binReset — Reset histogram bin values
scalar

A **binReset** value of 1 (`true`) triggers a RAM initialization sequence that resets the histogram bin values. Reset takes `NumBins` cycles to clear all locations. Input signals are ignored during this interval.

Data Types: `Boolean`

Output

readRdy — Indicates histogram bins available for read
scalar

When **readRdy** is 1 (`true`), the histogram bins are ready to read. The block returns **readRdy** set to 1 (`true`) two cycles after the final pixel of a frame.

Data Types: `Boolean`

hist — Bin value corresponding to requested address
integer

Histogram bin value corresponding to the requested address, **binAddr**. The **Data type** parameter specifies the data type for this value.

Data Types: `fixed point`

validOut — Indicates that histogram value is valid
scalar

When **validOut** is 1 (`true`), the histogram bin value, **hist**, is valid.

Data Types: `Boolean`

Parameters

Number of bins — Number of histogram bins

256 (default) | power of 2 between 32 and 4096

Choose the number of bins depending on the input word length (WL). If the number of bins is less than 2^{WL}, the block truncates the least-significant bits of each pixel. If the number of bins is greater

than 2^{WL}, some bins are not used, and after you synthesize your design it will use more hardware resources than necessary.

Data type — Data type of histogram bin values

Unsigned fixed point (default) | double | single

Data type of histogram bin values.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Word length — Word length of histogram bins

16 (default) | integer

Word length of the histogram bins. If a bin overflows, the count saturates and the block shows a warning.

Dependencies

This parameter applies when you set **Data type** to Unsigned fixed point.

Algorithms

RAM Reset and Ready Sequence

Before sending the first input data, you must wait **Number of bins** cycles for the block to reset the RAM. This initial reset happens without asserting **binReset**.

You cannot read histogram bins and apply pixel data at the same time. To read the bin values, wait for the **readRdy** port to be 1 (`true`), and then apply each bin address of interest to the **binAddr** port. The block provides the corresponding histogram values on the **hist** port, with an accompanying **validOut** signal.

The histogram values persist and accumulate across frames until you assert **binReset**. At that time, the block takes **Number of bins** cycles to clear the RAM and be ready for new input. Other input signals are ignored during reset.

The diagram shows an overview of the reset sequence. `vStart` and `vEnd` are control signals in the `pixelcontrol` input bus.

The diagram shows the automatic startup reset, followed by a frame of video input. The read window starts when **readRdy** is asserted. The **binReset** signal initiates a bin reset. The next input frame is not applied until after the reset is complete.

The following diagram illustrates a bin read sequence. `vEnd` is a control signal in the `pixelcontrol` input bus. **validOut** indicates when the bin values on **hist** are available.

After the last pixel of a video frame, indicated by `vEnd = true`, the block asserts **readRdy** to show that the histogram is ready for reading. Two cycles after applying a bin address, the block provides the value of that bin on **hist**, with a corresponding valid signal. You can request the last bin address and assert **binReset** at the same time.

Latency

The block sets **readRdy** to `true` two cycles after receiving the last pixel of a frame. The input `pixelcontrol` bus indicates the last pixel of a frame by `vEnd = true`. While **readRdy** is `true`, the block captures **binAddr** requests on each cycle. The block provides the corresponding histogram bin values on **hist** two cycles later.

Version History
Introduced in R2015a

R2022b: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2022a: Multipixel streaming

The Histogram block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

The block supports input column vectors of *NumPixels* values, where *NumPixels* is 4 or 8. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix. The output interface is the same whether the input is scalar or vector.

R2019b: Increased histogram sizes

You can now configure the block to have 2048 or 4096 histogram bins.

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 one 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 for this block inside a Resettable Synchronous Subsystem.

See Also

Blocks
2-D Histogram | Frame To Pixels

Objects
visionhdl.Histogram

Functions
imhist

Image Filter

2-D FIR filtering

Libraries:
Vision HDL Toolbox / Filtering

Description

The Image Filter block performs two-dimensional finite impulse response (FIR) filtering on a pixel stream and supports the use of programmable filter coefficients.

Ports

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar intensity value. For multipixel streaming, specify a vector of two, four, or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

This block does not support multicomponent streaming. To process multicomponent streams, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect a single bus to multiple replicated blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

coeff — Filter coefficients
matrix

Specify the filter coefficients as a 2-D matrix of numeric values. Each dimension of the matrix must have at least 2 elements, but no more than 64 elements.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

The block samples the values from the **coeff** port only at the start of a frame and ignores any changes within a frame.

Dependencies

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

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

Output

pixel — Output pixel stream
scalar | vector

Output pixel stream, returned as a scalar value representing intensity, or as a vector of two, four, or eight pixel intensity values. The dimensions and data type of the output **pixel** port match the dimensions and data type of the input **pixel** port.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Parameters

Main

Filter coefficients source — Source to provide filter coefficients

Property (default) | Input port

Select the source for specifying the filter coefficients.

- Property (default) — Use this value to specify filter coefficients using the **Filter coefficients** parameter.
- Input port — Use this value to specify filter coefficients through the **coeff** input port.

Filter coefficients — Coefficients of filter

[1, 0; 0, -1] (default) | matrix

Specify the filter coefficients as a matrix. Each dimension of the matrix must have at least 2 elements, but no more than 64 elements.

Dependencies

To enable this parameter, set the **Filter coefficients source** parameter to Property.

Padding method — Method for padding

Constant (default) | Replicate | Symmetric | Reflection | None

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- Constant — Interpret pixels outside the image frame as having a constant value.
- Replicate — Repeat the value of pixels at the edge of the image.
- Symmetric — Set the value of the padding pixels to mirror the edge of the image.
- Reflection — Set the value of the padding pixels to reflect around the pixel at the edge of the image.
- None — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

Padding value — Value used to pad boundary of input image

0 (default) | integer

Specify an integer to pad the boundary of the input image. The block casts this value to the same data type as the input pixel.

Dependencies

To enable this parameter, set the **Padding method** parameter to `Constant`.

Line buffer size — Size of line buffer

2048 (default) | integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (N – 1)-by-**Line buffer size** memory locations to store the pixels. N represents the rows of the coefficient matrix.

Data Types

Rounding mode — Rounding method for internal fixed-point calculations

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

Specify a rounding method for internal fixed-point calculations.

Saturate on integer overflow — Overflow action for internal fixed-point calculations

off (default) | on

When you clear this parameter, fixed-point and integer values wrap around to zero when the value overflows what is representable with that data type. When you select this parameter, the value saturates at the maximum representable value.

Coefficients — Filter coefficients data type

Inherit: Same as first input (default) | fixdt(1,16,0) | <data type expression>

Select the method for determining the data type of the filter coefficients.

Click the Show data type assistant button to display the **Data Type Assistant**, which helps you set the data type of the **Coefficients** parameter. For details, see "Specify Data Types Using Data Type Assistant" (Simulink).

When converting the coefficients to the specified data type, the block rounds to the nearest representable value and saturates to the maximum value if the value exceeds the maximum value representable with the data type.

Dependencies

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

Output — Output data type

Inherit: Same as first input (default) | Inherit: Inherit via internal rule | fixdt(1,16,0) | <data type expression>

Select the method for determining the data type of the output pixel.

Click the Show data type assistant button to display the **Data Type Assistant**, which helps you to set the data type of the **Output** parameter. For details, see "Specify Data Types Using Data Type Assistant" (Simulink).

Lock data type settings against changes by the fixed-point tools — Lock data type settings
off (default) | on

Select this parameter to lock all data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see "Lock the Output Data Type Setting" (Fixed-Point Designer).

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

The block implements the 2-D FIR filter with a fully pipelined architecture. Each multiplier has two pipeline stages on each input and two pipeline stages on each output. The adder is a pipelined tree structure. HDL code generation uses symmetric, unity, or zero-value coefficients to reduce the number of multipliers.

When you use multipixel streaming, the block uses a single line memory and implements one filter kernel for each of the *M* input pixels, in parallel. This increase in hardware resources is a trade off for increasing throughput compared to single-pixel streaming.

When you provide coefficients using the **Filter coefficients** parameter, you can optimize the multipliers for HDL code generation by using a canonical signed digit (CSD) representation or factored CSD representation. To use a CSD of factored CSD representation, right-click the block, select **HDL Code > HDL Block Properties**, and set the **ConstMultiplierOptimization** parameter to csd or fcsd.

When you provide coefficients using the **coeff** port, the latency depends on the size of the filter coefficients. For an *N*-by-*M* coefficient matrix provided using the **coeff** port, the block generates *N*x*M* multipliers.

Latency

The latency of the block is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding by default. The latency of the padding operation depends on the size of the kernel. If edge padding is not necessary for your design, you can reduce the latency by setting the **Padding method** parameter to None. When you use this option, the block latency does not depend on your kernel size. To determine the exact latency for any configuration of the block, measure the number of time steps between the input and output control signals.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch + Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020a: Option to omit padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

R2019b: Multipixel streaming

The block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

For multipixel streaming, the block supports input and output column vectors of 4 or 8 pixels. The **ctrl** port remains scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the vector.

R2019a: Increased kernel size limits

The block now allows for a coefficient kernel with up to 64-by-64 elements.

R2018b: Programmable coefficients
Behavior changed in R2018b

The block now accepts coefficients from an input port. Each dimension of the matrix must have at least 2 and no more than 16 elements. The block samples the values from the **coeff** port at the start of a frame only and ignores any changes within a frame.

In previous releases, you could specify a row vector of coefficients, that is, a matrix of 1-by-N elements. Now, the coefficient matrix must have at least 2 elements in each dimension.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one default HDL architecture.

HDL Block Properties

ConstMultiplierOptimization	Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also "ConstMultiplierOptimization" (HDL Coder).
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 if it is inside a Resettable Synchronous Subsystem.

See Also
2-D FIR Filter | Frame To Pixels | visionhdl.ImageFilter

Image Resizer

Change dimensions of image in pixel stream

Libraries:
Vision HDL Toolbox / Geometric Transforms

Description

The Image Resizer block downscales greyscale input images. The horizontal and vertical scale factors are independent. The block provides an optional antialiasing filter, and you can choose nearest-neighbor or bilinear interpolation.

You can use this block for image resize operations such as:

- Converting images to a uniform size for input to deep learning algorithms
- Modifying disparate source videos for processing, resource, or display requirements

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar

Single pixel, specified as a scalar value.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `fixdt(0,WL,0), WL between 8 and 16` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Output

pixel — Output pixel stream
scalar

Configure the data type of the output pixel by using the **Output** parameter on the **Data Type** tab.

Data Types: `uint8` | `uint16` | `fixdt(0,WL,0), WL between 8 and 16` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Parameters

Main

Input height and width — Dimensions of active frame in input stream

`[1080,1920]` (default) | `[ActiveVideoLines,ActivePixelsPerLine]`

Dimensions of the input frame, specified as a row vector of the form [*ActiveVideoLines,ActivePixelsPerLine*]. The dimensions must be integers greater than or equal to `[16,16]` and less than or equal to `[4320,8192]`.

Output height and width — Dimensions of active frame in output stream

`[720,1280]` (default) | `[ActiveVideoLines,ActivePixelsPerLine]`

Dimensions of the output frame, specified as a matrix of the form [*ActiveVideoLines,ActivePixelsPerLine*]. The dimensions must be integers greater than or equal to `[16,16]` and less than or equal to `[4320,8192]`.

The output dimensions must be less than or equal to the input dimensions.

Insert horizontal blanking — Add invalid cycles between active lines

off (default) | on

The horizontal blanking interval is the period of inactive cycles between the end of one line and the beginning of the next line. When you select this option, the block includes a buffer to store the output pixels while inserting extra horizontal blanking cycles between the active lines of the output video stream. Set the number of blanking cycles in the **Minimum output horizontal blanking** parameter. For information on when to use horizontal blanking, see "Configure Blanking Intervals".

Minimum output horizontal blanking — Number of inactive cycles to insert between output lines

370 (default)

The horizontal blanking interval is the period of inactive cycles between the end of one line and the beginning of the next line. If your downstream processing requires a certain blanking interval, specify

that interval in this parameter. This parameter value must be an integer between 1 and 1195. The block includes a buffer that stores the output pixels until the minimum number of blanking cycles is reached.

The recommended minimum horizontal blanking interval for Vision HDL Toolbox library blocks that use a line buffer is 2×*Kw* when using padding or 12 cycles when you set the **Padding method** parameter to None. For more information on blanking requirements for image processing systems, see "Configure Blanking Intervals".

Dependencies

To enable this parameter, select **Insert horizontal blanking**.

Interpolation method — Method of interpolating pixel values

Bilinear (default) | Nearest Neighbor

Method of interpolating pixel values in horizontal and vertical directions, specified as one of these types:

- Bilinear — The block implements a two-tap filter in each direction, where the taps are the current and previous pixel values.
- Nearest Neighbor — The object implements muxes to select the closest of the current pixel value or the previous pixel value. This interpolation method uses the fewest hardware resources, at the tradeoff of coarser interpolation accuracy.

For more information about interpolation methods, see the "Change Image Size" example.
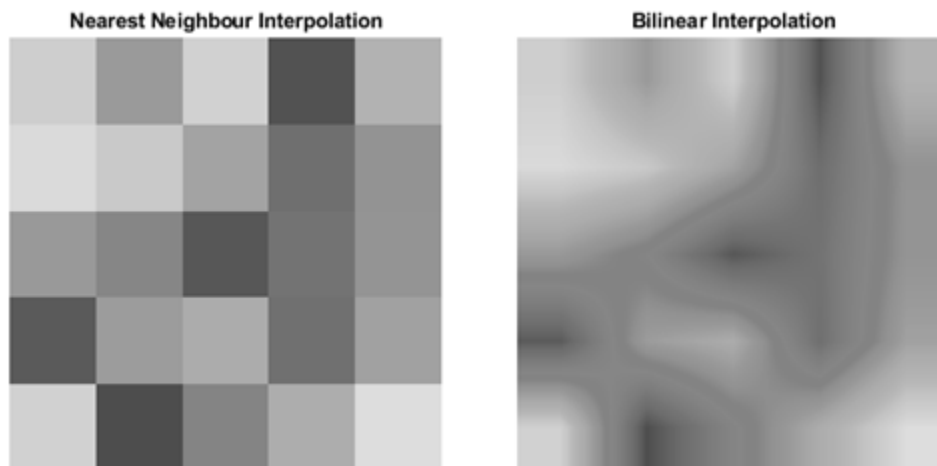
Perform antialiasing on downscale — Apply antialiasing filter before resizing

on (default) | off

When you set this property to true, the object applies an antialiasing filter to the input image before the resize operation. The antialiasing filter is a separable 5-by-5 filter. The coefficients are [1 4 6 4 1]/16.

Data Types

Rounding mode — Rounding method for internal fixed-point calculations

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

Specify a rounding method for internal fixed-point calculations.

Saturate on integer overflow — Overflow action for internal fixed-point calculations

off (default) | on

When you clear this parameter, fixed-point and integer values wrap around to zero when the value overflows what is representable with that data type. When you select this parameter, the value saturates at the maximum representable value.

Coefficients — Interpolation filter coefficients data type

fixdt(0,16) (default) | <data type expression>

Specify the data type of the filter coefficients as a data type expression in the form fixdt(0,WL), where the type is unsigned and *WL* is a word length in the range 2 to 17. The block computes the binary point scaling to accommodate the interpolation coefficients. When converting the coefficients to the specified data type, the block uses the settings of the **Rounding method** and **Overflow action** parameters.

The interpolation filter also limits the internal pixel data types to 24 bits wide so that the filter maps to DSP blocks on your FPGA.

Output — Output data type

Inherit: Same as first input (default) | fixdt(0,16,0) | <data type expression>

Specify the data type of the output pixels as Inherit: Same as first input or as a data type expression in the form fixdt(0,WL,0), where the type is unsigned, and *WL* is a word length in the range 8 to 16.

- Inherit: Same as first input — Sets the data type of the output pixels to match the data type of the input **pixel** port.
- <data type expression> — Sets the data type of the output pixels to this data type.

When converting the pixel values to the specified data type, the block uses the settings of the **Rounding method** and **Overflow action** parameters.

Algorithms

The resize architecture consists of a scale calculator, buffer, interpolator, and optional antialiasing filter and output buffer. The scale calculator converts the specified input and output sizes into accumulators that control the selection of pixels from the buffer. If you select the antialiasing filter, the input stream is filtered with a 5-by-5 fixed kernel before entering the input buffer. The input buffer consists of circular buffers that form the vertical kernels required for the interpolation filter. The interpolator contains vertical and horizontal filters that use the interpolation method that you selected. The antialiasing filter and the interpolation filters use symmetric padding. At the output of

the block, the `pixelcontrol` bus control signals are regenerated, and data is buffered again if necessary to maintain the specified minimum horizontal blanking interval.

Version History
Introduced in R2023a

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 one 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 for this block inside a Resettable Synchronous Subsystem.

See Also

visionhdl.ImageResizer

Image Statistics

Calculate mean, variance, and standard deviation
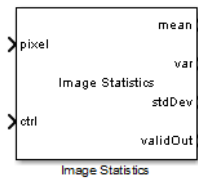
Libraries:
Vision HDL Toolbox / Statistics

Description

The Image Statistics block calculates the mean, variance, and standard deviation of streaming video data. Each calculation is performed over all pixels in the input region of interest (ROI). The block implements the calculations by using hardware-efficient algorithms.

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Ports

Input

pixel — Input pixel stream
scalar

Single image pixel in a pixel stream, specified as an unsigned scalar value that represents grayscale intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `fixdt(0,W,0)` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Output

The block uses full-precision arithmetic for internal calculation. At the output ports, intermediate data is cast back to the input type by using the following fixed-point settings: **Rounding method** =

Nearest, and **Overflow action** = Saturate. The table shows the output word length for each calculation, relative to the input word length (*IWL*).

Mean	Variance	Std. Deviation
IWL	$2{\times}IWL$	$2{\times}IWL$

mean — Mean of most recent input frame completed
scalar

Mean of the most recent input frame completed, returned as a scalar. The data type of **mean** is same as the data type of **pixel**.

Data Types: uint8 | uint16 | fixdt(0,W,0) | double | single

var — Variance of most recent input frame completed
scalar

Variance of the most recent input frame completed, returned as a scalar. The data type of **var** is same as the data type of **pixel**.

Data Types: uint8 | uint16 | fixdt(0,W,0) | double | single

stdDev — Standard deviation of most recent input frame completed
scalar

Standard deviation of the most recent input frame completed, returned as a scalar. The data type of **stdDev** is same as the data type of **pixel**.

Data Types: uint8 | uint16 | fixdt(0,W,0) | double | single

validOut — Computations completed
boolean

The block sets this output to true when the statistic outputs for a frame are ready.

Data Types: Boolean

Parameters

Enable mean output — Calculate the mean of each input frame

off (default) | on

Select this parameter to enable the **mean** port and calculate the mean of each input frame.

Enable variance output — Calculate the variance of each input frame

off (default) | on

Select this parameter to enable the **var** port and calculate the variance of each input frame.

Enable std. deviation — Calculate the standard deviation of each input frame

off (default) | on

Select this parameter to enable the **stdDev** port and calculate the standard deviation of each input frame.

Tips

- To change the size and dimensions of the ROI, you can manipulate the input video stream control signals. See "Regions of Interest" on page 1-137.
- The number of valid pixels in the input image affects the accuracy of the mean approximation. To avoid approximation error, use an image that contains a multiple of 64^n pixels, up to 64^{n+1} pixels, where n equals 0, 1, 2, or 3. For details of the mean approximation, see "Algorithm" on page 1-134.
- The block calculates statistics over frames up to 64^4 (16, 777, 216) pixels in size. This size supports HD frames.

Algorithms

Architecture

The calculations of mean, variance, and standard deviation build off each other. For hardware efficiency, the calculation logic is shared as shown.
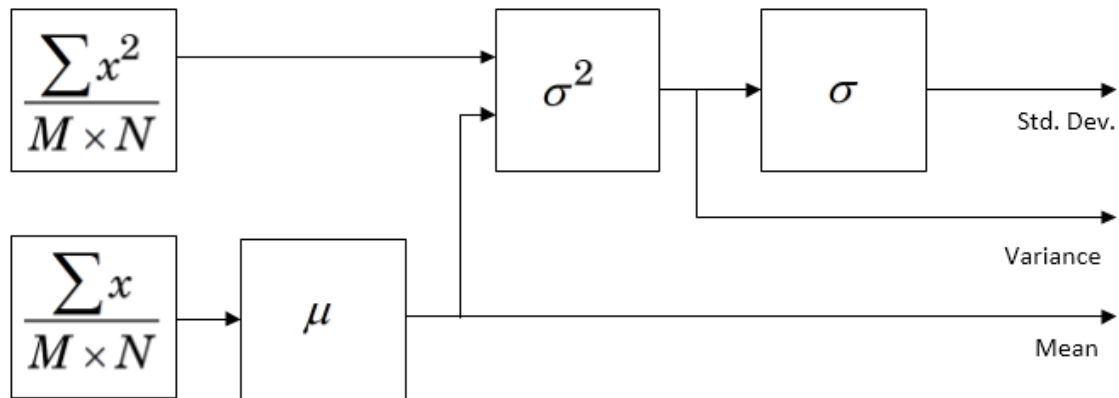
Mean

The equation to calculate the precise mean pixel value requires large internal word lengths and expensive division logic.

$$\mu = \frac{1}{M * N} \sum_{i=1}^{M} \sum_{j=1}^{N} x_{ij}$$

Instead of using this equation, the block calculates the mean by a series of four accumulators that compute the mean of a segment of pixels. First, the block finds the sum of a window of 64 pixels, and normalizes.

$$\mu_{L1} = \frac{1}{64} \sum_{n=1}^{64} x_n$$

Then it accumulates 64 of the previous windows, and normalizes.

$$\mu_{L2} = \frac{1}{64} \sum_{n=1}^{64} \mu_{nL1}$$

A third accumulator sums 64 of the 64×64 windows, and normalizes the same way.

$$\mu_{L3} = \frac{1}{64} \sum_{n=1}^{64} \mu_{nL2}$$

The fourth accumulator sums 64 of the 64×64×64 windows and normalizes.

$$\mu_{L4} = \frac{1}{64} \sum_{n=1}^{64} \mu_{nL3}$$

Each valid pixel is accumulated as it arrives. Its location within a line or frame does not affect the accumulation logic.

When `vEnd` is received, the block promotes any remaining data in the four levels of mean calculation to calculate the final output. If an accumulator counter is not at 64 when `vEnd` arrives, that level normalizes by the actual value of the counter. The constants for this multiplication are stored in a lookup table (LUT). The four accumulators share a single LUT and multiplier. The values in the LUT are in `ufix18_en17` data type so that the multiplier fits into a DSP block on an FPGA.
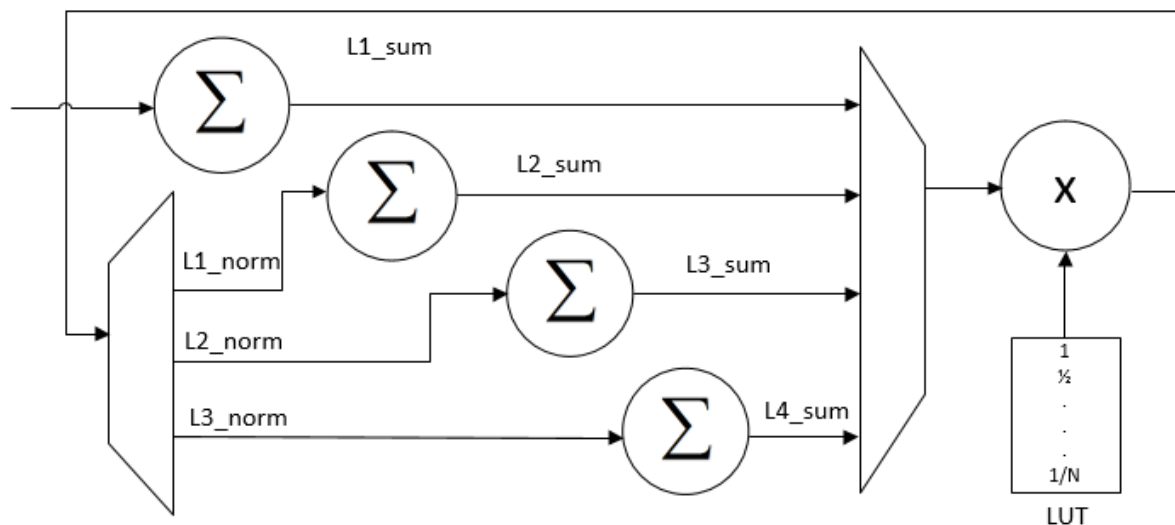
This method of mean calculation is accurate when the number of pixels in the frame aligns `vEnd` with the final accumulator rollover. This alignment occurs at level 2 when the frame contains a multiple of 64 pixels, and fewer than 64^2(4096) pixels. It occurs at level 3 when the frame contains a multiple of 4096 pixels. It occurs at level 4 when the frame contains a multiple of 64^3 pixels. This method is also accurate when the frame has fewer than 64 pixels, because only the first accumulator is needed.

However, when the number of pixels in the frame does not fit these conditions, the block must normalize the final accumulation before the counter reaches 64. This normalization introduces an error in the calculation at subsequent levels. The figure shows the normalization error introduced in the mean calculation by image sizes under 4096 pixels. The spikes occur where an image size is just over a multiple of 64 pixels.

For images larger than 4096 pixels, the same effect occurs at multiples of 4096 pixels, and at multiples of 64^3 pixels.
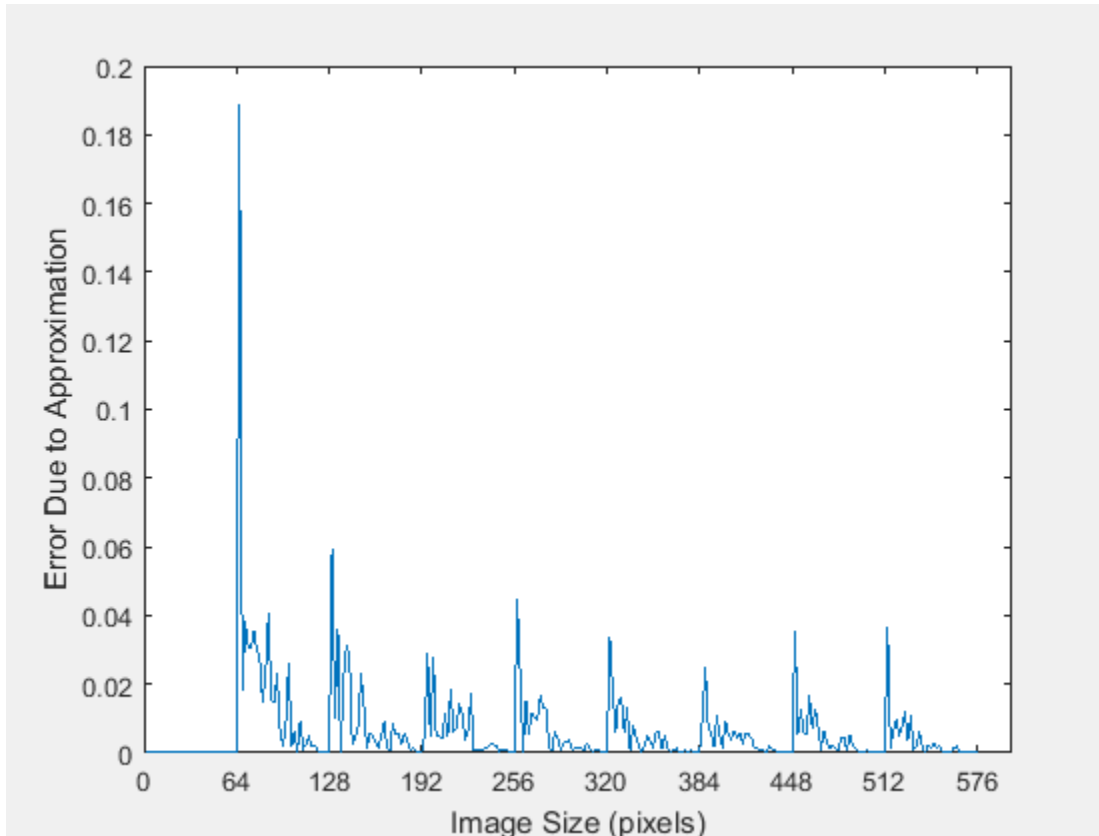
Variance

The block calculates the variance of the input pixels by using this equation:

$$\sigma^2 = (\frac{1}{M*N} \sum_{i=1}^{M} \sum_{j=1}^{N} x_{ij}^2) - \mu^2$$

The mean and the mean of the squared input are calculated in parallel. The block calculates the mean of squares using the same approximation method used to calculate the mean, as described in the previous section.

Standard Deviation

The block calculates the square root of the variance by using a pipelined bit-set-and-check algorithm. This algorithm computes the square root by using addition and shifts, rather than multipliers. For an N-bit input, the result has N bits of accuracy.

This method is hardware efficient for general inputs. If your data has known characteristics that allow for a more efficient square root implementation, you can disable the calculation in this block and construct your own logic from HDL-supported blocks.

Regions of Interest

Statistics are often calculated on small regions of interest (ROI) rather than an entire video frame. This block performs calculations on all valid pixels between the `vStart` and `vEnd` signals in the **ctrl** bus and does not track pixel location within the frame. You can use the ROI Selector block to select a smaller region of the frame before passing the video stream to this block. For an example that selects multiple small ROIs from a larger image and compares the statistics of each region, see "Multi-Zone Metering".

The Image Statistics block calculates statistics over frames up to 64^4 (16,777,216) pixels in size. If you provide an image with more than 64^4 pixels, the block calculates the requested statistics on only the first 16,777,216 pixels and then asserts `validOut`. The block ignores extra pixels until it receives a `vEnd` signal.

Latency

The latency from `vEnd` to **validOut** depends on the calculations you select.

When the block receives a `vEnd` signal that is `true`, it combines the remaining data in the four levels of mean calculation to calculate the final output. This final step takes four cycles per level, resulting in a maximum of 16 cycles of latency between the input `vEnd` signal and the **validOut** signal. Once the mean is available, the variance calculation takes four cycles. The square root logic requires input word length (IWL) cycles of latency.

If a calculation is not selected, and is not needed for other selected calculations, that logic is excluded from the generated HDL code.

The table shows the calculation logic and latency for various block configurations.

Mean	Variance	Std. Deviation	Logic Excluded From HDL	Latency (cycles)
✓	✓	✓		$[4n]+4+$IWL, (where n is the number of accumulator levels required for the input size)
✓			variance and square root	$[4n]$
	✓		square root	$[4n]+4$
		✓		$[4n]+4+$IWL
✓	✓		square root	$[4n]+4$
✓		✓		$[4n]+4+$IWL
	✓	✓		$[4n]+4+$IWL

Note You must have a vertical blanking interval of at least 16 cycles between the `vEnd` of one input frame and the `vStart` of the next input frame. This period lets the block finish processing the current frame before the new one arrives. For more information, see "Configure Blanking Intervals".

Version History

Introduced in R2015a

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 one 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

2-D Mean | 2-D Variance | 2-D Standard Deviation | Frame To Pixels | `visionhdl.ImageStatistics`

Topics
"Multi-Zone Metering"

Lookup Table

Map input pixel to output pixel using custom rule
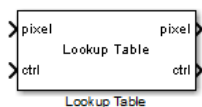
Libraries:
Vision HDL Toolbox / Conversions

Description

The Lookup Table block provides a custom one-to-one map between an input pixel value and an output pixel value.

Ports

This block uses a streaming pixel interface with a `pixelcontrol` bus for frame control signals. This interface enables the block to operate independently of image size and format. All Vision HDL Toolbox blocks use the same streaming interface. The block accepts and returns a scalar pixel value and a bus that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar | vector

For single pixel streaming, specify **pixel** as a scalar intensity value or a row vector of **Number of components** values. For multipixel streaming, specify **pixel** as a column vector of **Number of pixels** pixel intensity values. **Number of pixels** can be two, four, or eight. This block does not support multipixel-multicomponent streaming.

Integer and fixed-point data types larger than 16 bits are not supported. Signed data types are not supported.

Data Types: uint8 | uint16 | fixdt(0,N,F), N <= 16 | Boolean

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: bus

Output

pixel — Output pixel stream
scalar | vector

The dimensions of the output **pixel** match the dimensions of the input **pixel**. The block returns the output **pixel** as a scalar, a row vector of **Number of components** values, or a column vector of **Number of pixels** values.

The data type is specified by the **Table data** parameter.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

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

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Parameters

Table data — Map between input pixel and output pixel

`uint8(0:1:255)` (default) | vector

Map between input pixel and output pixel, specified as a row or column vector of any data type. The data type of the table data determines the data type of the output **pixel**. This mapping determines the one-to-one correspondence between an input **pixel** value and an output **pixel** value.

- The length of the vector must be $2^{WordLength}$, where *WordLength* is the size, in bits, of the input **pixel**. The Lookup Table block does not perform interpolation. Every input value must have a corresponding output value in the table.

- The smallest representable value of the input data type maps to the first element of the table, the second smallest value maps to the second element, and so on. For example, if the input **pixel** has a data type of `fixdt(0,3,1)`, the input value 0 maps to the first element of the table, 0.5 maps to the second element, 1 maps to the third element, and so on.

When the input **pixel** is a vector, the block replicates the lookup table for each element of the vector. In that case, the tables use $VectorSize \times 2^{WordLength}$ memory locations.

Example: `uint8(linspace(255,0,256))`

Algorithms

Latency

The latency of the Lookup Table block is 2 cycles.

Version History
Introduced in R2015a

R2023a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2020b: Multipixel streaming

The Lookup Table block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel. You cannot use multicomponent and multipixel together.

For multipixel streaming, the blocks support input and output column vectors of 4 or 8 pixels. The **ctrl** port remains scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the vector.

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 one 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).
LUTRegisterResetType	The reset type of the lookup table output register. Select none to synthesize the lookup table to a ROM when your target is an FPGA. See also "LUTRegisterResetType" (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
Frame To Pixels

Objects
visionhdl.LookupTable

Measure Timing

Measure timing of pixel control bus input
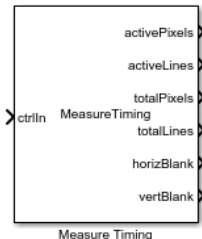
Libraries:
Vision HDL Toolbox / Utilities

Description

The Measure Timing block measures the timing parameters of a video stream. This block is for use in Simulink simulation only and does not support HDL code generation.

The Vision HDL Toolbox streaming pixel protocol implements the timing of a video system, including inactive intervals between frames. These inactive intervals are called *blanking intervals*. Many Vision HDL Toolbox blocks require minimum blanking intervals. For more information, see "Configure Blanking Intervals". You can use the timing parameter measurements from this block to check that your video stream meets these requirements. If you manipulate the control signals of your video stream, you can use this block to verify the resulting signals.

To determine the parameters of each frame, the block measures the time steps between the control signals on the bus.
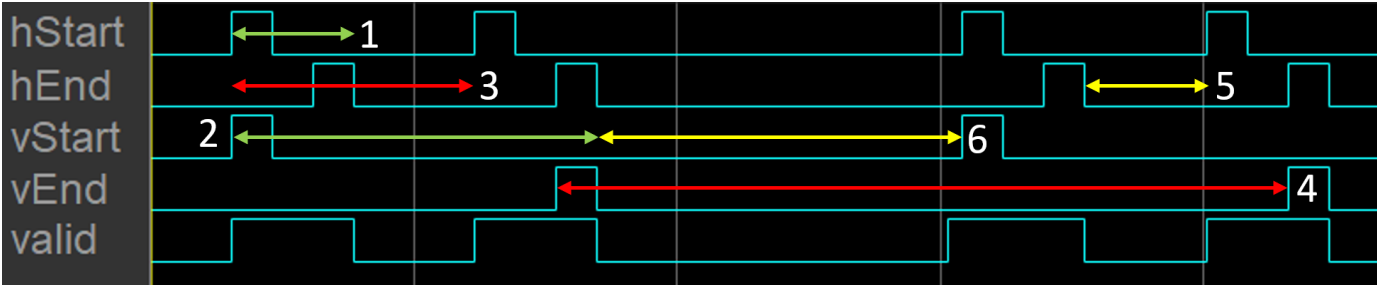
- 1 — Active pixels per line
- 2 — Active lines per frame (count `hStart` pulses)
- 3 — Total pixels per line
- 4 — Total lines per frame (cycles divided by total pixels per line)
- 5 — Horizontal blanking
- 6 — Vertical blanking (cycles, minus horizontal blanking, divided by total pixels per line)

For details on the pixel control bus and the dimensions of a video frame, see "Streaming Pixel Interface".

Note Measurements from the first simulated frame are incorrect because some parameters require measurements between frames. Simulate at least two frames before using the results.

Ports

Input

ctrlIn — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: bus

Output

activePixels — Number of pixels in each line of active video frame
scalar

Number of pixels in each line of the active video frame, returned as a scalar. This value is measured by counting valid pixels between hStart and hEnd. See marker 1 in the diagram.

Data Types: double

activeLines — Number of lines in active video frame
scalar

Number of lines in the active video frame, returned as a scalar. This value is measured by counting the hStart pulses between vStart and vEnd. See marker 2 in the diagram.

Data Types: double

totalPixels — Number of pixels in each line, including the horizontal blanking interval
scalar

Number of pixels in each line, including the horizontal blanking interval, returned as a scalar. This value is measured by counting the number of valid pixels between two hStart pulses. See marker 3 in the diagram.

Data Types: double

totalLines — Number of lines in the frame, including the vertical blanking interval
scalar

Number of lines in the frame, including the vertical blanking interval, returned as a scalar. This value is measured by counting the number of valid pixels between two vEnd pulses, divided by totalPixels. See marker 4 in the diagram.

Data Types: double

horizBlank — Number of inactive pixels between lines of a frame
scalar

Number of inactive pixels between lines of a frame, returned as a scalar. This value is measured by counting invalid cycles between hEnd and the next hStart. See marker 5 in the diagram.

Data Types: `double`

vertBlank — Number of inactive lines between frames
scalar

Number of inactive lines between frames, returned as a scalar. This value is measured by counting invalid cycles between `vEnd` and the next `vStart`, subtracting `horizBlank`, and then dividing by `totalPixels`. See marker `6` in the diagram.

Data Types: `double`

Version History
Introduced in R2016b

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 can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

See Also
Frame To Pixels | `visionhdl.MeasureTiming`

Topics
"Streaming Pixel Interface"
"Configure Blanking Intervals"

Median Filter

2-D median filtering
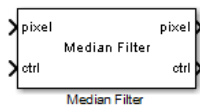
Libraries:
Vision HDL Toolbox / Analysis & Enhancement
Vision HDL Toolbox / Filtering

Description

The Median Filter block replaces each input pixel with the median value of a specified surrounding N-by-N neighborhood. The median is less sensitive to extreme values than the mean. You can use this block to remove salt-and-pepper noise from an image without significantly reducing the sharpness of the image. You can specify the neighborhood size and padding values for edges of the input image.

Ports

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of M pixels per clock cycle to support high-frame-rate or high-resolution formats. The M value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar intensity value. For multipixel streaming, specify a vector of two, four, or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

This block does not support multicomponent streaming. To process multicomponent streams, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect a single bus to multiple replicated blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Output

pixel — Output pixel stream
scalar | vector

This block supports single pixel streaming or multipixel streaming. When using single pixel streaming, the block returns a single pixel as a scalar intensity value. When using multipixel streaming, the block returns a vector of intensity values. This vector is the same size as the input **pixel** vector. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

The output pixel data type is the same as the data type of the input pixels.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Parameters

Neighborhood size — Size of image region used to compute median

3×3 (default) | 5×5 | 7×7

Select the neighborhood size, in pixels, as 3×3, 5×5, or 7×7.

You cannot use a neighborhood size of 7×7 when you use multipixel streaming and set **Padding method** to None.

Padding method — Method for padding boundary of input image

Symmetric (default) | Constant | Replicate | Reflection | None

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- `Constant` — Interpret pixels outside the image frame as having a constant value.
- `Replicate` — Repeat the value of pixels at the edge of the image.
- `Symmetric` — Set the value of the padding pixels to mirror the edge of the image.
- `Reflection` — Set the value of the padding pixels to reflect around the pixel at the edge of the image.
- `None` — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

You cannot use a neighborhood size of 7×7 when you set **Padding method** to `None`.

Padding value — Value used to pad boundary of input image

`0` (default) | integer

Specify an integer to pad the boundary of the input image. The block casts this value to the same data type as the input pixel.

Dependencies

To enable this parameter, set the **Padding method** parameter to `Constant`.

Line buffer size — Size of line buffer

`2048` (default) | integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (N— 1) -by-**Line buffer size** memory locations to store the pixels used to compute the median value. N is the dimension of the square region specified by the **Neighborhood size** parameter.

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

When you use multipixel streaming, the block uses a single line memory and implements one filter kernel for each of the M input pixels, in parallel. This increase in hardware resources is a trade off for increasing throughput compared to single-pixel streaming.

Latency

The latency of the block is the line buffer latency plus the latency of the kernel calculation. The line buffer latency includes edge padding by default. The latency of the padding operation depends on the size of the kernel. If edge padding is not necessary for your design, you can reduce the latency by setting the **Padding method** parameter to None. When you use this option, the block latency does not depend on your kernel size. To determine the exact latency for any configuration of the block, measure the number of time steps between the input and output control signals.
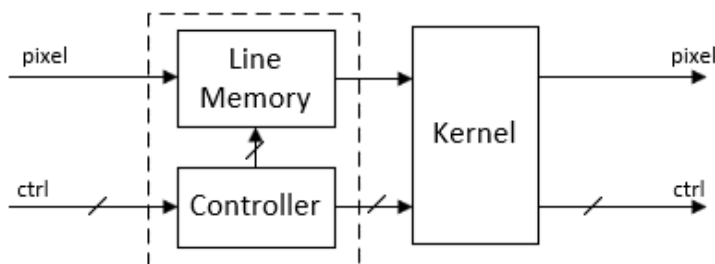
To find the median value of each neighborhood, the kernel logic compares the pixel values to each other. The latency of the filter kernel depends on the neighborhood size. The table shows the number of comparisons needed to find the median value of each size of neighborhood.

Neighborhood Size	Number of Comparisons Performed
3×3	11
5×5	75
7×7	230

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch** + **Back porch**. For more information, see "Configure Blanking Intervals".

Performance

For the Median Filter block, the comparison logic that finds the median pixel value of each neighborhood is typically the critical path that determines the achievable clock frequency after synthesis. As the size of a neighborhood increases, the length of this path increases.

Version History
Introduced in R2015a

R2023a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020a: Option to omit padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

R2019b: Multipixel streaming

The Median Filter block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

For multipixel streaming, the block supports input and output column vectors of 4 or 8 pixels. The **ctrl** port remains scalar, and the control signals in the pixelcontrol bus apply to all pixels in the vector.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 if it is inside a Resettable Synchronous Subsystem.

See Also
Median Filter | Frame To Pixels | `visionhdl.MedianFilter`

Opening

Morphological opening of binary pixel data

Libraries:
Vision HDL Toolbox / Morphological Operations

Description

The Opening block performs morphological erosion followed by morphological dilation by using the same neighborhood for both calculations. The block operates on a stream of binary intensity values. You can specify a neighborhood or structuring element of up to 32-by-32 pixels.

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Ports

Input

pixel — Boolean input pixel or multipixel vector
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar Boolean value. For multipixel streaming, specify a vector of four or eight Boolean values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

Data Types: `Boolean`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: bus

Output

pixel — Boolean output pixel or multipixel vector
scalar | vector

This block supports single pixel streaming or multipixel streaming. When using single pixel streaming, the block returns a single pixel as a scalar Boolean value. When using multipixel streaming, the block returns a vector of Boolean values. This vector is the same size as the input **pixel** vector. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

Data Types: Boolean

ctrl — Control signals associated with pixel stream
pixelcontrol bus

The pixelcontrol bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one valid signal, the pixels in the vector must be either all valid or all invalid. The hStart and vStart signals apply to the pixel with the lowest index in the vector. The hEnd and vEnd signals apply to the pixel with the highest index in the vector.

Data Types: bus

Parameters

Neighborhood — Pixel neighborhood

[0,1,0; 1,1,1; 0,1,0] (default) | vector or matrix of 1s and 0s

Pixel neighborhood, specified as a vector or matrix of binary values.

The block supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the **Neighborhood** parameter to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function..

When you use multipixel vector input, the neighborhood must be at least two pixels in each dimension.

Line buffer size — Size of line memory buffer

2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The block allocates (n – 1)-by-**Line buffer size** memory locations to store the pixels, where n is the number of lines in the **Neighborhood** parameter value.

Padding method — Method for padding

Constant (default) | None

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- Constant — The block pads the image with zeros for the dilation operation and with ones for the erosion operation. These values prevent opening at the boundaries of the active frame.

- None — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

Tips

- When you use a block with an internal line buffer inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

Latency

The total latency of the block is the line buffer latency plus the latency of the kernel calculation. Morphological opening is a compound operation. Therefore, this block contains a second line buffer between the erosion kernel and the dilation kernel. To determine the exact latency for any configuration of the block, monitor the number of time steps between the input control signals and the output control signals.

The latency of the line memory includes edge padding. The latency of the kernel depends on the neighborhood size.

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch + Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2015a

R2020a: Option to omit padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

R2020a: Multipixel streaming

The Opening block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

For multipixel streaming, the blocks support input and output column vectors of 4 or 8 pixels. The **ctrl** port remains scalar, and the control signals in the pixelcontrol bus apply to all pixels in the vector.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this block now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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 if it is inside a Resettable Synchronous Subsystem.

See Also

Closing | Dilation | Erosion | Frame To Pixels | `visionhdl.Opening`

Topics

"Types of Morphological Operations" (Image Processing Toolbox)
"Structuring Elements" (Image Processing Toolbox)

Pixel Control Bus Creator

Create control signal bus for use with Vision HDL Toolbox blocks
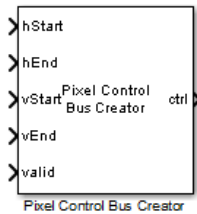
Libraries:
Vision HDL Toolbox / Utilities

Description

The Pixel Control Bus Creator block creates a `pixelcontrol` bus. See "Pixel Control Bus".

The block is an implementation of the Simulink Bus Creator block. See Bus Creator for more information.

Troubleshooting When you generate HDL code from a Simulink model that uses the `pixelcontrol` bus, you might encounter this error.

```
Cannot resolve variable 'pixelcontrol'
```

To avoid this issue, use the `pixelcontrolbus` function to create an instance of the bus type in the base workspace. Then try generating HDL code again.

The Vision HDL Toolbox model template includes this line in the `InitFcn` callback.

```
evalin('base','pixelcontrolbus')
```

Ports

Input

hStart — First pixel in a horizontal line of a frame
scalar

First pixel in a horizontal line of a frame, specified as a `Boolean` scalar.

Data Types: `Boolean`

hEnd — Last pixel in a horizontal line of a frame
scalar

Last pixel in a horizontal line of a frame, specified as a `Boolean` scalar.

Data Types: `Boolean`

vStart — First pixel in the first (top) line of a frame
scalar

First pixel in the first (top) line of a frame, specified as a `Boolean` scalar.

Data Types: `Boolean`

vEnd — Last pixel in the last (bottom) line of a frame
scalar

Last pixel in the last (bottom) line of a frame, specified as a `Boolean` scalar.

Data Types: `Boolean`

valid — Valid pixel indicator
scalar

Valid pixel indicator, specified as a `Boolean` scalar.

Data Types: `Boolean`

Output

ctrl — Control signals accompanying pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Version History
Introduced in R2015a

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.

To learn more about using buses for HDL code generation, see "Buses" (HDL Coder) and "Use Bus Signals to Improve Readability of Model and Generate HDL Code" (HDL Coder).

HDL Architecture

This block has one 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

Pixel Control Bus Selector | Frame To Pixels | Pixels To Frame

Topics
"Streaming Pixel Interface"

Pixel Control Bus Selector

Select signals from control signal bus used by Vision HDL Toolbox blocks
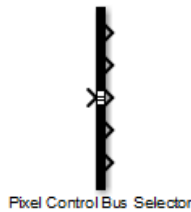
Libraries:
Vision HDL Toolbox / Utilities

Description

The Pixel Control Bus Selector block selects signals from the `pixelcontrol` bus. See "Pixel Control Bus".

The block is an implementation of the Simulink Bus Selector block. See Bus Selector for more information.

Ports

Input

Port_1 — Control signals accompanying pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

Data Types: `bus`

Output

Port_1 — First pixel in a horizontal line of a frame
scalar

First pixel in a horizontal line of a frame, returned as a `Boolean` scalar. This port returns the `hStart` signal from the input bus.

Data Types: `Boolean`

Port_2 — Last pixel in a horizontal line of a frame
scalar

Last pixel in a horizontal line of a frame, returned as a `Boolean` scalar. This port returns the `hEnd` signal from the input bus.

Data Types: `Boolean`

Port_3 — First pixel in the first (top) line of a frame
scalar

First pixel in the first (top) line of a frame, returned as a `Boolean` scalar. This port returns the `vStart` signal from the input bus.

Data Types: `Boolean`

Port_4 — Last pixel in the last (bottom) line of a frame
scalar

Last pixel in the last (bottom) line of a frame, returned as a `Boolean` scalar. This port returns the `vEnd` signal from the input bus.

Data Types: `Boolean`

Port_5 — Valid pixel indicator
scalar

Valid pixel indicator, returned as a `Boolean` scalar. This port returns the `valid` signal from the input bus.

Data Types: `Boolean`

Version History
Introduced in R2015a

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.

To learn more about using buses for HDL code generation, see "Buses" (HDL Coder) and "Use Bus Signals to Improve Readability of Model and Generate HDL Code" (HDL Coder).

HDL Architecture

This block has one 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

Pixel Control Bus Creator | Frame To Pixels | Pixels To Frame

Topics
"Streaming Pixel Interface"

Pixels To Frame

Convert pixel stream to frame-based video
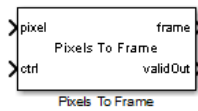
Libraries:
Vision HDL Toolbox / I/O Interfaces

Description

The Pixels To Frame block converts a color or grayscale pixel stream and control signals to frame-based video. The control signal bus indicates the validity of each pixel and its location within the frame. The pixel stream format can include padding pixels around the active frame. You can configure the frame and padding dimensions by selecting a common video format or by specifying custom dimensions. The pixel input can support single pixel streaming, or multipixel streaming. Multipixel streaming accepts 2, 4, or 8 pixels each clock cycle to support high-rate or high-resolution formats. For details of the pixel stream format, see "Streaming Pixel Interface".

This block itself does not support HDL code generation. However, you can use this block to convert the output of a separate subsystem, targeted for HDL code generation, back to frames.

If your model converts frames to a pixel stream and later converts the stream back to frames, specify the same video format for the Frame To Pixels block and the Pixels To Frame block.

Ports

Input

pixel — Input pixel stream
scalar | vector | matrix

For single pixel streams, specify **pixel** as a scalar or a vector of 1-by-**Number of components** values that represent a single pixel. For multipixel streams, specify **pixel** as a matrix of **Number of pixels**-by-**Number of components** pixel intensity values. **Number of pixels** can be two, four, or eight.**Number of components** can be from one to four.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `single` | `double`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Output

frame — Full image frame
array

Full image frame, returned as an **Active pixels per line**-by-**Active video lines**-by-*N* array. The first two array dimensions are the height and width of the active image specified by the **Video format** parameter. *N* is the **Number of components** used to express a single pixel. The output data type is the same as the data type of the **pixel** port.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `single` | `double`

validOut — Indicates when output frame is ready
scalar

This signal is 1 (`true`) when the block has successfully recompiled an output frame from the input stream.

Data Types: `Boolean`

Parameters

Number of components — Number of values used to represent each pixel

1 (default) | 2 | 3 | 4

Number of values used to represent each pixel. Each pixel can have from 1 to 4 components.

- For grayscale video, set this parameter to 1.
- For color video, for example, {R,G,B} or {Y,Cb,Cr}, set this parameter to 3.
- For color video with an alpha channel for transparency, set this parameter to 4.

To process multicomponent streams for blocks that do not support multicomponent input, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect the **ctrl** output of the Frame To Pixels block to each replicated block.

Number of pixels — Number of pixels streamed per time step

1 (default) | 2 | 4 | 8

Number of pixels transferred on the streaming interface for each time step, specified as 1, 2, 4, or 8. To enable multipixel streaming and increase throughput for high-resolution or high-frame-rate video, set this parameter to 2, 4 or 8. Multipixel streaming processes more pixels with the same clock frequency as a single-pixel streaming interface.

Video Resolution	Clock Frequency Required for Single-Pixel Streaming at 60 fps	Clock Frequency Required for Multipixel Streaming at 60 fps (4 pixels)
1080p	150 MHz	37.5 MHz
4k UHD	600 MHz	150 MHz
8k UHD	1200 MHz	300 MHz

Video format — Dimensions of active regions in a video frame

1080p (default) | 240p | 480p | 480pH | 576p | 720p | 768p | 1024p | 1200p | 2KCinema | 4KUHDTV | 8KUHDTV | Custom

Dimensions of active regions in a video frame. To select a predefined format, use the **Video format** pull-down menu. The active frame dimensions are set as shown in the table. For a custom format, select Custom, then specify the active frame dimensions as integers.

Video Format	Active Pixels Per Line	Active Video Lines
240p	320	240
480p	640	480
480pH	720	480
576p	720	576
720p	1280	720
768p	1024	768
1024p	1280	1024
1080p (default)	1920	1080
1200p	1600	1200
2KCinema	2048	1080
4KUHDTV	3840	2160
8KUHDTV	7680	4320
Custom	User-defined	User-defined

Version History

Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2020a: Multipixel-multicomponent streaming

The block now supports multipixel-multicomponent streams. The block accepts an input matrix of *NumPixels*-by-*NumComponents* pixels. The **ctrl** port remains scalar, and the control signals in the pixelcontrol bus apply to all pixels in the matrix. The "MultiPixel-MultiComponent Video Streaming" example shows how to use a multipixel and multicomponent pixel stream to process a high-resolution color image.

R2019b: Multipixel streaming

The block now accepts pixel streams that have 4 or 8 pixels per clock cycle. You can set the **Number of pixels** parameter to 4 or 8. With this setting, the block expects an input vector of 4 or 8 pixel

values on each time step. For an example of how to use this feature, see "Filter Multipixel Video Streams".

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.

See Also

Blocks
Frame To Pixels

Objects
`visionhdl.PixelsToFrame`

Topics
"Streaming Pixel Interface"

Pixel Stream FIFO

Buffer input stream to create image lines that have contiguous valid pixels

Libraries:
Vision HDL Toolbox / Utilities

Description

The Pixel Stream FIFO block stores incoming valid pixels and accompanying control signals and returns the same pixel stream without gaps between the valid pixels of each line. The block preserves the total line size and total frame size of the video stream, including invalid cycles.

Use the Pixel Stream FIFO block to buffer video sources. The waveform shows a direct memory access (DMA) video source, where pixels are read in bursts, and a Camera Link® video source, where pixels are valid every second clock cycle. To create contiguous video lines, a Pixel Stream FIFO block buffers the input pixels and control signals of each source.

Ports

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

This block also supports multipixel-multicomponent streams, where the pixel input is a matrix of *M*-by-*N* values. *M* is number of pixels and *N* is number of components. These values correspond to the **Number of pixels** and **Number of components** parameters of the Frame To Pixels block.

Input

pixel — Input pixel stream
scalar | vector | matrix

For single pixel streams, specify **pixel** as a scalar or a vector of 1-by-**Number of components** values that represent a single pixel. For multipixel streams, specify **pixel** as a matrix of **Number of pixels**-by-**Number of components** pixel intensity values. **Number of pixels** can be two, four, or eight.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `single` | `double`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Output

pixel — Output pixel stream
scalar | vector | matrix

Each line of the output stream has contiguous valid pixels. The dimensions and data type of the output **pixel** stream match those of the input **pixel** stream. The block returns the output pixel stream as a scalar, a vector of 1-by-**Number of components** values, or a matrix of **Number of pixels**-by-**Number of components** values.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `single` | `double`

ctrl — Control signals accompanying pixel stream
`pixelcontrol` bus

Control signals accompanying pixel stream, returned as a `pixelcontrol` bus containing five signals. The signals indicate the validity of the pixel and its location in the frame. For more information on the signal protocol, see "Pixel Control Bus".

The Pixel Stream FIFO block returns each line of pixels with the **ctrl**.`valid` signal continuously `true` between **ctrl**.`hStart` and **ctrl**.`hEnd`. The block buffers the entire input line before it returns the modified line.

Data Types: `bus`

Parameters

Line buffer size — Size of line memory buffer

2048 (default) | positive integer

Specify a power of two that accommodates the number of active pixels in a single horizontal line. If you specify a value that is not a power of two, the block uses the next largest power of two.

Algorithms

The Pixel Stream FIFO block contains a memory controller, read and write counters, and two RAMs. One RAM stores the incoming control signals, and the other stores the incoming pixel stream. The block stores valid pixels and their accompanying control signals for each line, as determined by the input **ctrl**.hStart and **ctrl**.hEnd signals. The buffering removes any bursty behavior of the input stream. Once a full line of valid pixels is stored, the block returns the new continuous version of the line.
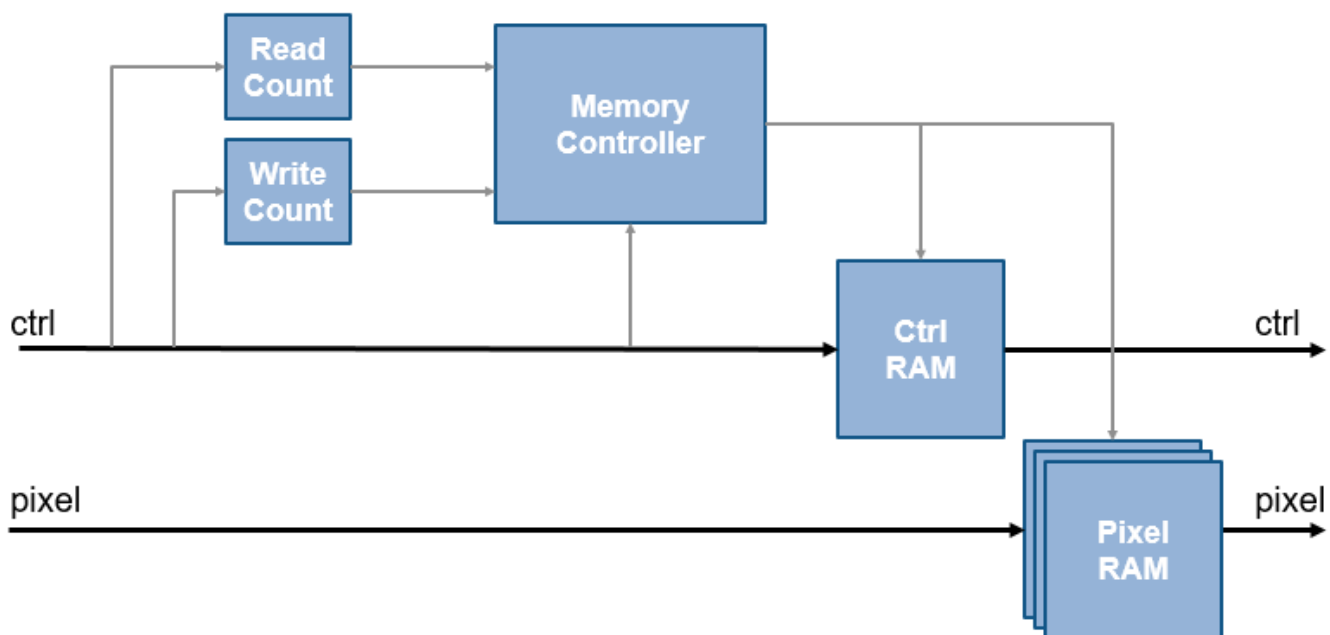
When the input **pixel** is a vector or a matrix, the block replicates the Pixel RAM for each element. The diagram shows three Pixel RAMs, to represent a three-component pixel stream. For multipixel streaming, the block adjusts the line buffer size to store 1/**Number of pixels** pixels. For instance, with a 4-by-3 input stream, each buffer stores ¼ of the pixels for a line, so there are 12 Pixel RAMs, each with 2^{m-2} locations.

Version History
Introduced in R2018a

R2022a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2021a: Multipixel-multicomponent streaming

The block now supports streams that are both multipixel and multicomponent. The HDL implementation replicates the algorithm for each pixel and component in parallel.

The block supports input and output matrices of *NumPixels*-by-*NumComponents* pixels. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

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 one 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 for this block inside an Enabled Subsystem, Resettable Synchronous Subsystem, or a Triggered Subsystem.

See Also

Blocks
Frame To Pixels | Line Buffer

ROI Selector

Select a region of interest (ROI) from pixel stream
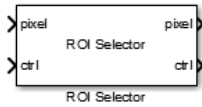
Libraries:
Vision HDL Toolbox / Utilities

Description

The ROI Selector block selects a portion of the active frame from a video stream. The size of the inactive frame remains the same. The output control signals indicate a new active region of the frame. You can use this block with a single pixel, multipixel, or multipixel-multicomponent pixel stream. This diagram shows the inactive pixel regions in blue and the requested output region outlined in orange.
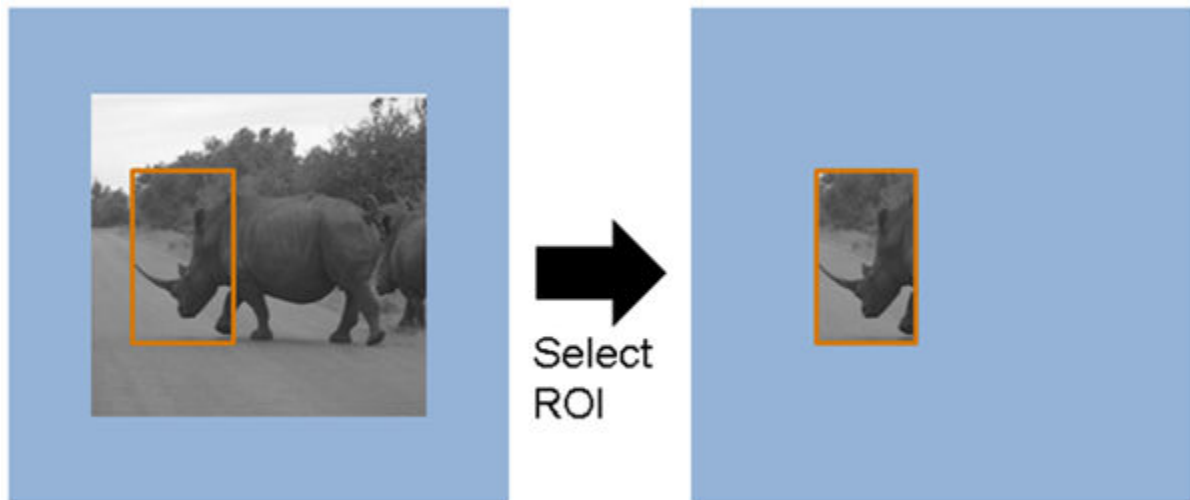
You can specify a fixed size and location for the ROI, or you can select the region location dynamically by using an input port. You can select more than one region. Define each region by specifying its upper-left corner coordinates and dimensions. By default, the block returns one set of pixels and control signals for each region you specify. The block sets the inactive pixels in the output frame to zero.

Regions are independent from each other, so they can overlap. If you specify a region that includes the edge of the active frame, the block returns only the active portion of the region. This diagram shows the output frames for three requested regions. The second output region does not include the inactive area above the image.

The block also provides a mode for vertical reuse. In this mode, you must specify regions that have no vertical overlap and are aligned in columns. Each column of regions shares one output pixel stream. The control signals define each region in the stream. This arrangement enables parallel processing of each column, and the reuse of downstream processing logic for each region in a column. To use this mode, select the **Reuse output ports for vertically aligned regions** parameter. Use the `visionhdlframetoregions` function to divide a frame into tiled regions for vertical reuse.
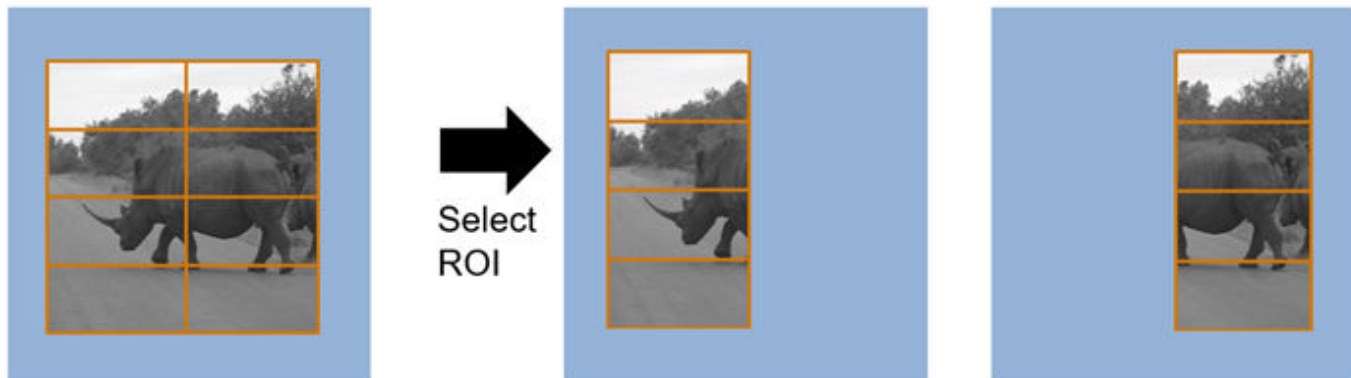
Ports

This image shows the additional ports on the block when you configure three regions of interest from input ports.

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of M pixels per clock cycle to support high-frame-rate or high-resolution formats. The M value corresponds to the

Number of pixels parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

This block also supports multipixel-multicomponent streams, where the pixel input is a matrix of *M*-by-*N* values. *M* is number of pixels and *N* is number of components. These values correspond to the **Number of pixels** and **Number of components** parameters of the Frame To Pixels block.

Input

pixel — Input pixel stream
scalar | vector | matrix

For single pixel streams, specify **pixel** as a scalar or a vector of 1-by-**Number of components** values. For multipixel streams, specify **pixel** as a matrix of **Number of pixels**-by-**Number of components** pixel intensity values. **Number of pixels** can be two, four, or eight.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

region1,...,regionR — Region of interest
four-element row vector

Region of interest, specified as a row vector of four positive integers that define the coordinates of the top-left corner and dimensions of each desired output frame. The vector must have the form [*hPos vPos hSize vSize*]. The block has *R* region ports, where *R* is the **Number of regions** parameter value.

For multipixel streaming the regions must meet these conditions, where *M* is **Number of pixels**.

- `mod(`*hPos*`, M) = 1` — For example, if your pixel stream has four pixels per cycle, 129 is a valid value of *hPos* (`mod(129,4) = 1`), while 128 is invalid (`mod(128,4) = 0`).

- `mod(`*hSize*`, M) = 0` — For example, if your pixel stream has four pixels per cycle, 120 is a valid *hSize* (`mod(120,4) = 0`), while 118 is invalid (`mod(118,4) = 2`).

Dependencies

To enable this port, clear the **Reuse output ports for vertically aligned regions** parameter and set the **Regions source** parameter to `Input port`.

Data Types: `int` | `uint`

Output

pixel1,...,pixelR — Output pixel stream
scalar | vector | matrix

Output pixel, returned as a single pixel or multipixel stream. The output dimensions and data type match those of the input **pixel** port. By default, the block has R output **pixel** ports, where R is the **Number of regions** parameter, or the number of rows in the **Regions** parameter matrix.

When you select **Reuse output ports for vertically aligned regions**, the block has one output **pixel** port for each column of vertically-aligned regions.

Data Types: `uint` | `int` | `fixed point` | `Boolean` | `double` | `single`

ctrl1,...,ctrlR — Control signals
pixelcontrol bus

Control signals, returned as a `pixelcontrol` bus that contains five signals. By default, the block has R output **ctrl** ports, where R is the **Number of regions** parameter, or the number of rows in the **Regions** parameter matrix.

When you select **Reuse output ports for vertically aligned regions**, the block has one output **ctrl** port for each column of aligned regions.

Data Types: `bus`

Parameters

Reuse output ports for vertically aligned regions — Option to return one pixel stream for each column of regions

`off` (default) | `on`

When you select this parameter, you must specify regions that have no vertical overlap and that are aligned in columns. The block has one output pixel stream for each column of regions. This arrangement enables parallel processing of each column, and the reuse of downstream processing logic for each region in a column. Use the `visionhdlframetoregions` function to divide a frame into tiled regions for vertical reuse.

Dependencies

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

Regions source — Location of output region definitions

`Property` (default) | `Input port`

Specify the location of the output region definitions. You can set **Regions source** to one of these values:

- `Property` — Specify the regions by using the **Regions** parameter.
- `Input port` — Specify the regions by using input ports. Each input port corresponds to one region. The block samples the **region** input ports when `vStart` is `true` in the input control bus.

Dependencies

To enable this parameter, clear the **Reuse output ports for vertically aligned regions** parameter.

Regions — Rectangular ROIs to select from input frame

R-by-4 matrix

Specify rectangular ROIs to select from the input frame as an *R*-by-4 matrix.

R is the number of regions. The four elements that define each region are the top-left starting coordinates and the dimensions of the region and must be of the form [*hPos vPos hSize vSize*]. The coordinates count from the upper-left corner of the active frame, defined as [1,1]. *hSize* must be greater than 1.

For multipixel streaming the regions must meet these conditions, where *M* is **Number of pixels**.

- `mod(`*hPos*`, M) = 1` — For example, if your pixel stream has four pixels per cycle, `129` is a valid value of *hPos* (`mod(129,4) = 1`), while `128` is invalid (`mod(128,4) = 0`).
- **`mod(`*hSize*`, M) = 0`** — For example, if your pixel stream has four pixels per cycle, `120` is a valid *hSize* (`mod(120,4) = 0`), while `118` is invalid (`mod(118,4) = 2`).

When you clear the **Reuse output ports for vertically aligned regions** parameter, the regions can overlap, and *R* must not be greater than 16.

When you select **Reuse output ports for vertically aligned regions**, the regions in this matrix must have no vertical overlap and be aligned in columns. This diagram shows two examples of invalid regions outlined with a blue dashed line. The region on the left overlaps another region vertically. The two regions at the bottom do not align with the other regions in their columns. The regions do not have to cover the entire frame. Pixels outside of the tiled regions are marked as inactive pixels. The number of columns must not be greater than 16.
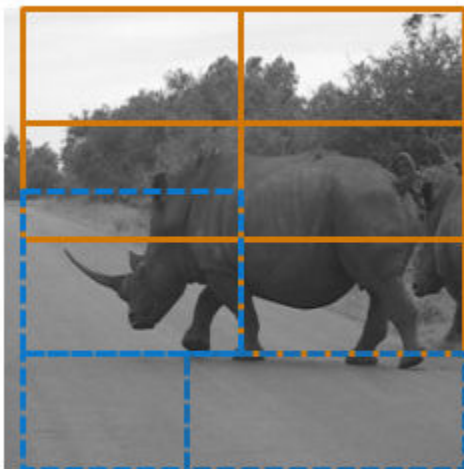
Dependencies

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

Number of regions — Number of region input ports

integer in the range [1, 16]

Specify the number of region input ports as an integer in the range [1, 16].

Dependencies

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

Algorithms

The generated HDL code for the ROI Selector block uses two 32-bit counters. The block does not use additional counters for additional regions.

Latency

The block has a latency of three cycles.

Version History
Introduced in R2016a

R2023a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2021a: Multipixel-multicomponent streaming

The block now supports streams that are both multipixel and multicomponent. The HDL implementation replicates the algorithm for each pixel and component in parallel.

The block supports input and output matrices of *NumPixels*-by-*NumComponents* pixels. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

R2021a: Resource sharing

The block now provides an option to share hardware resources when you select vertically aligned regions. Regions in the same column share the same `pixelcontrol` bus output.

Select the **Reuse output ports for vertically aligned regions** checkbox, and provide a set of regions that are aligned in columns and do not overlap vertically within each column. You can specify up to 1024 regions per column. To divide a frame into tiled regions that are compatible with vertical reuse, use the `visionhdlframetoregions` function.

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 one 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
Frame To Pixels

Objects
`visionhdl.ROISelector`

Functions
`visionhdlframetoregions`

Topics
"Lane Detection"

Pixel Stream Aligner

Align two streams of pixel data
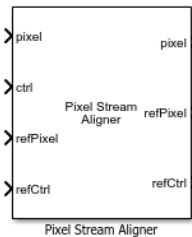
Libraries:
Vision HDL Toolbox / Utilities

Description

The Pixel Stream Aligner block synchronizes two pixel streams by delaying one stream to match the timing of a reference stream. Many Vision HDL Toolbox algorithms delay the pixel stream, and the amount of delay can change as you adjust algorithm parameters. You can use this block to align streams for overlaying, comparing, or combining two streams such as in a Gaussian blur operation. Connect the delayed stream to the **refPixel** and **refCtrl** input ports, and the earlier stream to the **pixel** and **ctrl** input ports.

This waveform diagram shows the input streams, `pixelIn` and `refPixelIn`, and their associated control signals. The reference input frame starts later than the `pixelIn` frame. The output signals show that the block delays `pixelIn` to match the reference stream, and that both output streams share control signals. The waveform shows the short latency between the input `refCtrl` and the output `refCtrl`. In this simulation, to accommodate the delay of four lines between the input streams, the **Maximum number of lines** parameter must be set to at least 4.
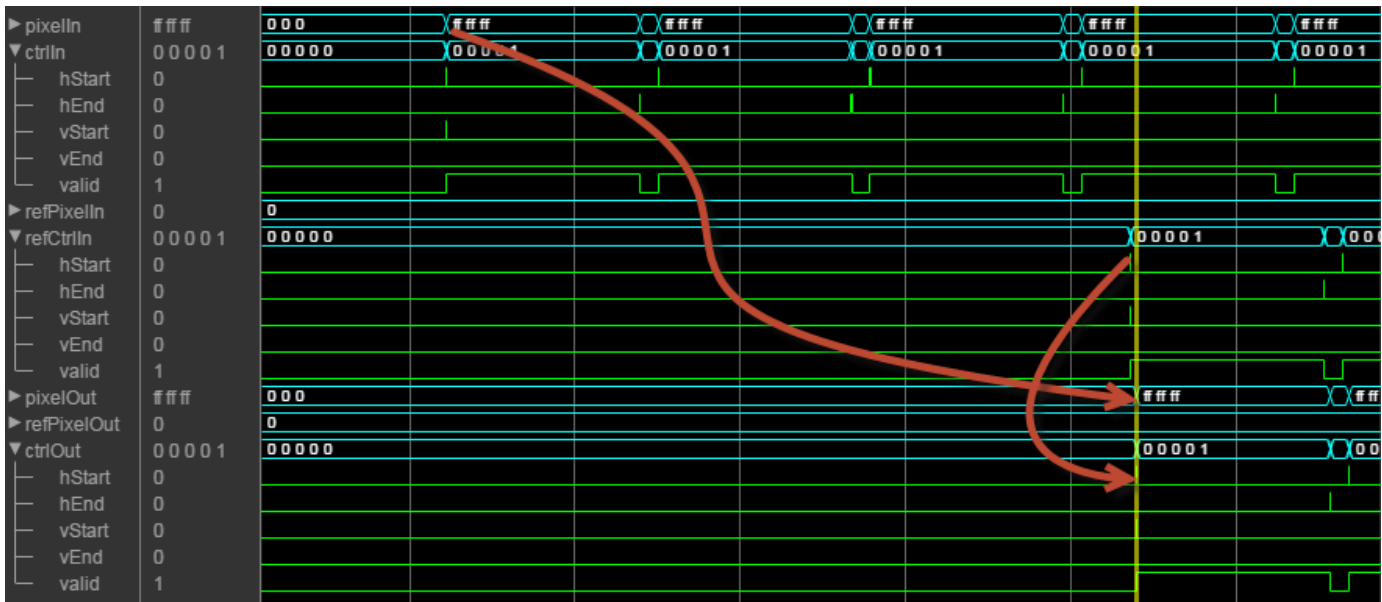
Ports

Input

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

This block also supports multipixel-multicomponent streams, where the pixel input is a matrix of *M*-by-*N* values. *M* is number of pixels and *N* is number of components. These values correspond to the **Number of pixels** and **Number of components** parameters of the Frame To Pixels block.

pixel — Input pixel stream
scalar | vector | matrix

For single pixel streams, specify **pixel** as a scalar or a vector of 1-by-**Number of components** values. For multipixel streams, specify **pixel** as a matrix of **Number of pixels**-by-**Number of components** pixel intensity values. **Number of pixels** can be two, four, or eight.

Because the block delays this pixel stream to match the control signals of the reference stream, **refPixel**, **pixel** must be the earlier of the two streams.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `fixed point` | `uint` | `int` | `Boolean` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

refPixel — Reference pixel stream
scalar | vector | matrix

Specify **refPixel** as a scalar, a vector of 1-by-**Number of components** values, or a matrix of **Number of pixels**-by-**Number of components** pixel intensity values.

The allowed dimensions for **refPixel** depend on the dimensions of the input **pixel** stream. The **Number of pixels** must be the same.

Input pixel Dimensions	Allowed refPixel Dimensions
1-by-*N*	1-by-1, 1-by-3, or 1-by-4
2-by-*N*	2-by-1, 2-by-3, or 2-by-4
4-by-*N*	4-by-1, 4-by-3, or 4-by-4
8-by-*N*	8-by-1, 8-by-3, or 8-by-4

Because the block delays the **pixel** input stream to match the reference control signals, **refPixel** must be the later of the two streams. The reference data and its control signals pass through the block with a small delay.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `fixed point` | `uint` | `int` | `Boolean` | `double` | `single`

refCtrl — Reference pixel stream control signals
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

The block uses these control signals for the aligned output stream.

Data Types: `bus`

Output

pixel — Aligned pixel stream
scalar | vector | matrix

The timing of the output **pixel** stream matches the timing of the output **refPixel** stream. The dimensions and data type of the output **pixel** stream match those of the input **pixel** stream.

Data Types: `fixed point` | `uint` | `int` | `Boolean` | `double` | `single`

refPixel — Reference pixel stream
scalar | vector | matrix

The dimensions and data type of the output **refPixel** stream match those of the input **refPixel** stream. The block passes this stream through the block unchanged.

Data Types: `fixed point` | `uint` | `int` | `Boolean` | `double` | `single`

refCtrl — Reference control signals
`pixelcontrol` bus

The block passes the input **refCtrl** signals through the block with a small delay.

Data Types: `bus`

Parameters

Line buffer size — Size of the line memory buffer

2048 (default) | positive integer

Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the block uses the next largest power of two. The block implements a circular buffer of 2^m pixels, where m is **Maximum number of lines** + $\log_2$(**Line buffer size**).

Maximum number of lines — Buffer depth that accommodates the timing offset between input streams

10 (default) | positive integer

The block implements a circular buffer of 2^m pixels, where m is **Maximum number of lines** + $\log_2$(**Line buffer size**), and a line address buffer of **Maximum number of lines** locations. The circular memory stores the earlier input lines until the reference control signals arrive. The line address buffer stores the address of the start of each line. When the reference control signals arrive, the block uses the stored address to read and send the delayed line. This parameter must accommodate the difference in timing between the two input streams, including the internal latency before the block reads the first line. During simulation, the block warns when an overflow occurs. To avoid the overflow condition, increase **Maximum number of lines**. The delay between streams cannot exceed an entire frame.

When the input **pixel** is a vector or a matrix, the block replicates the circular buffer for each element. For multipixel streaming, the block adjusts the line buffer size to store 1/**Number of pixels** pixels. For instance, with a 4-by-3 input stream, each buffer stores ¼ of the pixels for a line, and the buffer is $12 \times 2^{m-2}$ locations. The line address buffer remains the same size.

Algorithms

The block stores the data from the **pixel** input port to a circular buffer and then reads the lines out to align with the reference control signals. The block also stores the address of the start of each line. To match the added processing delay of the buffer data path, the block delays the reference pixel data and control signals for around 10 cycles.

The diagram shows the architecture of the block. The RAM Circular Buffer is a memory of 2^m pixels, where m is **Maximum number of lines** + $\log_2$(**Line buffer size**). The Line Address Buffer has **Maximum number of lines** locations. If the number of `ctrl.Hstart` assertions before the first `refCtrl.Hstart` assertion is greater than the size of the Line Address Buffer, the block overwrites both buffers and corrupts the output. In this diagram, **Maximum number of lines** is four, so four locations exist to store a line address in the buffer. The Frame Start Buffer is a single location to store `ctrl.Vstart`. The delay between the two streams must be less than the time between frame starts.
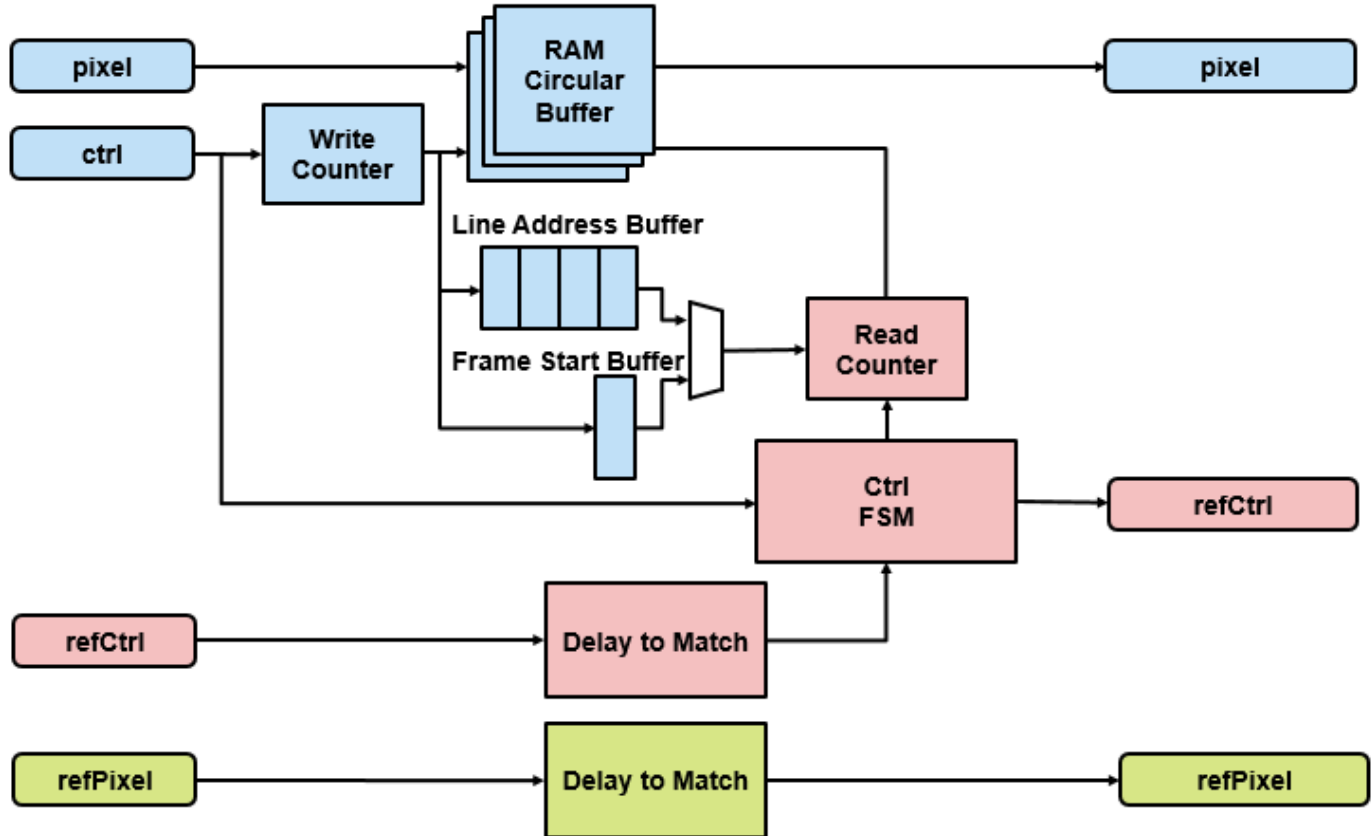
When the input **pixel** is a vector or a matrix, the block replicates the RAM Circular Buffer for each element. The diagram shows three RAM Circular Buffers, to represent a three-component pixel stream. Each RAM Circular Buffer is 2^m memory locations. For multipixel streaming, the block adjusts the line buffer size to store 1/**Number of pixels** pixels. For instance, with a 4-by-3 input stream, each buffer stores ¼ of the pixels for a line, and there are 12 RAM Circular Buffers, each with 2^{m-2} locations. The line address buffer remains the same size.

Version History
Introduced in R2017a

R2022b: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2021a: Multipixel-multicomponent streaming

The block now supports streams that are both multipixel and multicomponent. The HDL implementation replicates the algorithm for each pixel and component in parallel.

The block supports input and output matrices of *NumPixels*-by-*NumComponents* pixels, where *NumPixels* is 4 or 8. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

R2020b: Multipixel streaming

The Pixel Stream Aligner block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel. You cannot use multicomponent and multipixel together.

For multipixel streaming, the block supports input and output column vectors of 4 or 8 pixels. The **ctrl** port remains scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the vector.

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 one 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 for this block inside a Resettable Synchronous Subsystem.

See Also

Blocks
Frame To Pixels

Objects
visionhdl.PixelStreamAligner

Line Buffer

Store video lines and return neighborhood pixels
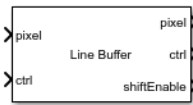
Libraries:
Vision HDL Toolbox / Utilities

Description

The Line Buffer block selects neighborhood pixels from streaming image data. It handles video control signals and edge padding, and is pipelined for high-speed video designs. The block outputs one column of the neighborhood at a time. To compose a neighborhood for further processing, use the **shiftEnable** signal to store the output columns, including padding, in a shift register. This block allows you to share the line buffer resources when your design performs multiple operations on the same neighborhood.

The following waveform shows the Line Buffer block returning 5-by-1 pixel columns that make up a 5-by-5 neighborhood. The time frame shown is at the beginning (top-left corner) of an input frame. The output starts after the block has stored two (`floor(M/2)`) lines and is receiving the start of the third line. The **shiftEnable** signal is asserted two cycles earlier than the output **ctrl**.`valid` signal, which indicates that the first two (`floor(M/2)`) columns are exclusively padding pixels. Similarly, **shiftEnable** stays high for two extra cycles at the end of the line.

Ports

This block uses a streaming pixel interface with a bus for frame control signals. This interface enables the block to operate independently of image size and format. The **pixel** ports on this block support single pixel streaming or multipixel streaming. Single pixel streaming accepts and returns a single pixel value each clock cycle. Multipixel streaming accepts and returns a vector of *M* pixels per clock cycle to support high-frame-rate or high-resolution formats. The *M* value corresponds to the **Number of pixels** parameter of the Frame To Pixels block. Along with the pixel, the block accepts

and returns a `pixelcontrol` bus that contains five control signals. The control signals indicate the validity of each pixel and their location in the frame. For multipixel streaming, one set of control signals applies to all pixels in the vector. To convert a frame (pixel matrix) into a serial pixel stream and control signals, use the Frame To Pixels block. For a full description of the interface, see "Streaming Pixel Interface".

Input

pixel — Input pixel stream
scalar | vector

This block supports single pixel streaming or multipixel streaming. For single pixel streaming, specify a single input pixel as a scalar intensity value. For multipixel streaming, specify a vector of two, four, or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see "Filter Multipixel Video Streams".

This block does not support multicomponent streaming. To process multicomponent streams, replicate the block for each component. The `pixelcontrol` bus for all components is identical, so you can connect a single bus to multiple replicated blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `int` | `uint` | `fixed point` | `Boolean` | `double` | `single`

ctrl — Control signals associated with pixel stream
`pixelcontrol` bus

The `pixelcontrol` bus contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see "Pixel Control Bus".

For multipixel streaming, each vector of pixel values has one set of control signals. Because the vector has only one `valid` signal, the pixels in the vector must be either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

Output

pixel — Column of neighborhood
column vector | matrix

Column of neighborhood pixel values, returned as a of 1-by-*H* vector, where *H* is the vertical neighborhood dimension. When using multipixel streaming, the block returns a *M*-by-*H* matrix, where *M* is the number of pixels in the input vector. The output pixel data type is the same as the data type of the input pixels.

To compose a neighborhood for further processing, use the **shiftEnable** signal to store the output columns for each kernel region, including padding, in a shift register. For multipixel streaming, use shift registers to construct *M* regions, each of *W*-by-*H* pixels, where *W* is the horizontal kernel dimension. Then, perform the kernel operation *M* times in parallel.

Note HDL code generation is supported for small matrices, but matrix operations can impact hardware performance and resource usage. Therefore, minimize how much your design operates on an *W*-by-*H* neighborhood directly. You can :

- Separate a filter into vertical and horizontal components.
- Concatenate the neighborhood pixels into an (*W**H)-by-1 vector.
- Use a For Each subsystem to store and operate on each pixel row.

These design suggestions provide opportunities to add pipelining around each adder or multiplier to increase synthesized clock speed and fit the design to DSP blocks on an FPGA.

Data Types: `int` | `uint` | `fixed point` | `Boolean` | `double` | `single`

ctrl — Control signals accompanying output column
`pixelcontrol` bus

Control signals accompanying the output column, returned as a `pixelcontrol` bus containing five signals. The signals describe the validity of the center pixel of the column, and the location of that pixel within the frame. See "Pixel Control Bus".

Columns that contain only padding pixels do not assert **ctrl**.`valid`. The **shiftEnable** signal is asserted for both padding and active columns.

Note For most of the frame, the block returns the input control signals that arrived with the bottom pixel of the column. However, for the final `floor(`*H*`/2)` lines of each frame, the bottom pixel of the column is a padding pixel, so the block generates output control signals. The block generates a contiguously asserted **ctrl**.`valid` signal for the valid pixels in each line.

For multipixel streaming, there is one set of control signals for each vector of pixel values. Since there is only one `valid` signal for the vector, the pixels in the vector are either all valid or all invalid. The `hStart` and `vStart` signals apply to the pixel with the lowest index in the vector. The `hEnd` and `vEnd` signals apply to the pixel with the highest index in the vector.

Data Types: `bus`

shiftEnable — Valid signal for neighborhood column
scalar

Valid signal for neighborhood columns, returned as a `Boolean` scalar. Use this signal to control a shift register that composes a neighborhood from the output columns. This signal is `true` (1) for any neighborhood column, indicating that the column contains valid image pixels or that the column was added for edge padding.

Data Types: `Boolean`

Parameters

Neighborhood size — Size of output neighborhood

[3 3] (default) | 2-element row vector of integer dimensions

Size of output neighborhood to be formed, specified as a 2-element row vector of integer dimensions [*H W*], where *H* is the vertical dimension and *W* is the horizontal dimension. The block returns a column vector of *H* elements. The *W* dimension is used to determine padding.

Padding method — Method for padding the boundary of input image

Symmetric (default) | Constant | Replicate | Reflection | None

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- Constant — Interpret pixels outside the image frame as having a constant value.
- Replicate — Repeat the value of pixels at the edge of the image.
- Symmetric — Set the value of the padding pixels to mirror the edge of the image.
- Reflection — Set the value of the padding pixels to reflect around the pixel at the edge of the image.
- None — Exclude padding logic. The block does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. For details, see "Increase Throughput by Omiting Padding".

Padding value — Value used to pad boundary of input image

0 (default) | integer

Specify an integer to pad the boundary of the input image. The block casts this value to the same data type as the input pixel.

Dependencies

To enable this parameter, set the **Padding method** parameter to Constant.

Line buffer size — Size of line memory buffer

2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

If you specify a value that is not a power of two, the block uses the next largest power of two.

Tips

- When you use this block inside an Enabled Subsystem, the enable signal pattern must maintain the timing of the pixel stream, including the minimum blanking intervals. If the enable pattern corrupts the timing of the pixel stream, you might see partial output frames, corrupted pixel stream control signals, or mismatches between Simulink and HDL simulation results. You may need to extend the blanking intervals to accommodate for cycles when the enable is low. For more information, see "Configure Blanking Intervals".

Algorithms

The block stores $H - 1$ lines of valid pixels, as specified by the neighborhood size. It adds padding bits at the edge of the frame. The block returns the first output column once it can form a complete neighborhood column, which occurs at the start of input line `floor(H/2)`.
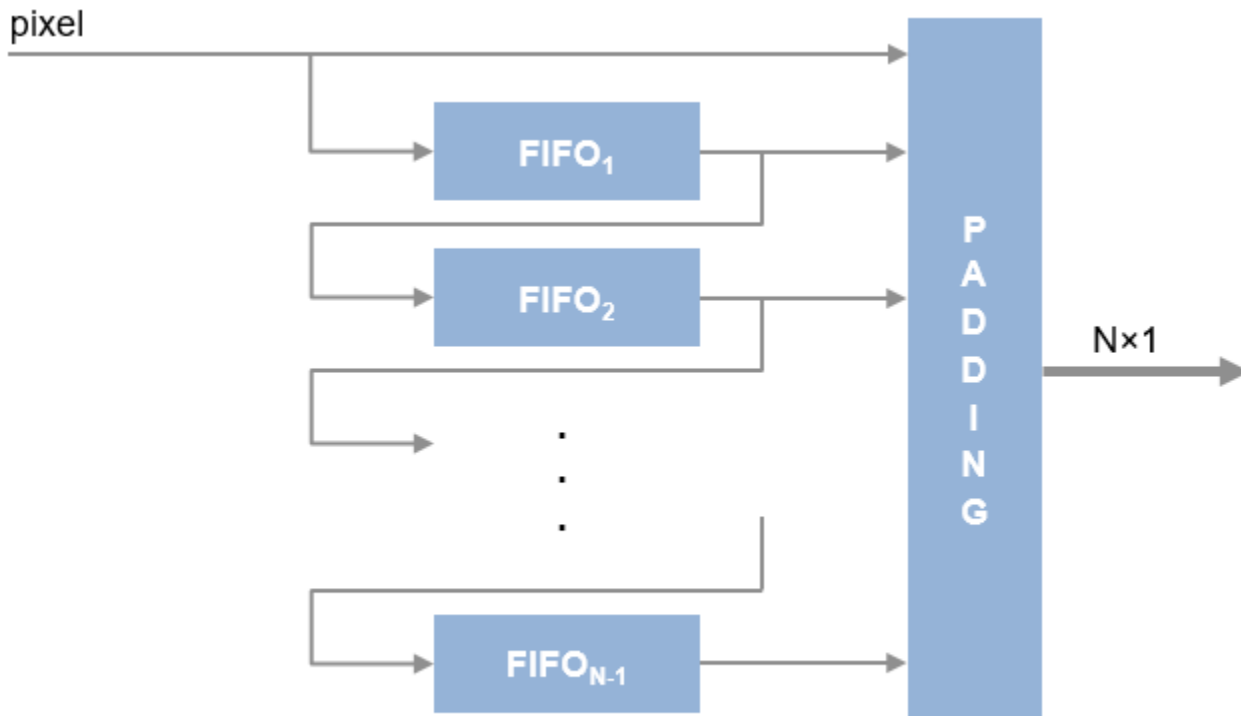
Performance

These resource and performance data are the synthesis results from the generated HDL targeted to a Xilinx® Zynq®-7000 ZC706 FPGA. The synthesis results were generated using Xilinx Vivado® 2017.4. The Line Buffer block is configured with symmetric padding and a line buffer size of `2048`. The table shows both odd and even neighborhood sizes.

	5-by-5 Kernel	6-by-6 Kernel
Clock frequency	300 MHz, 0.5 slack	300 MHz, 0.31 slack
LUT	647	790
Slice registers	1452	1844
BRAM	4	5

Latency

Note When you use edge padding, use a horizontal blanking interval greater than twice the kernel width. This interval lets the block finish processing one line before it starts processing the next one, including adding padding pixels before and after the active pixels in the line. Standard streaming

video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the filters applied to each frame. When you disable edge padding, the horizontal blanking interval must be at least 12 cycles and is independent of the kernel size. If you are using a custom video format, set the horizontal blanking interval by using the Frame To Pixels block parameters. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line** or, equivalently, **Front porch** + **Back porch**. For more information, see "Configure Blanking Intervals".

Version History
Introduced in R2017b

R2022a: Two pixels-per-clock streaming

The block now supports multipixel streams that have 2 pixels per clock cycle.

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020a: Option to omit padding

You can now configure the block to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to None. For an example, see "Increase Throughput by Omiting Padding".

R2019b: Multipixel streaming

The Line Buffer block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

For multipixel streaming, the block accepts an input vector of 4 or 8 pixels and returns a *KernelHeight*-by-*NumberOfPixels* matrix. The **ctrl** port remains scalar, and the control signals in the pixelcontrol bus apply to all pixels in the vector.

R2018b: Improved line buffer
Behavior changed in R2018b

The line buffer now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and blocks that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models. A best practice is to synchronize parallel paths in

your models by using the pixel stream control signals rather than by inserting a specific number of delays.

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 one 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
Frame To Pixels

Objects
visionhdl.LineBuffer

System Objects

visionhdl.BilateralFilter

Package: visionhdl

Perform 2-D filtering of a pixel stream

Description

The `visionhdl.BilateralFilter` object filters images while preserving edges. Some applications of bilateral filtering are denoising while preserving edges, separating texture from illumination, and cartooning to enhance edges. The filter replaces each pixel at the center of a neighborhood by an average that is calculated using spatial and intensity Gaussian filters. The object determines the filter coefficients from:

- Spatial location in the neighborhood (similar to a Gaussian blur filter)
- Intensity difference from the neighborhood center value

The object provides two standard deviation parameters for independent control of the spatial and intensity coefficients.

To perform bilateral filtering of a pixel stream:

1   Create the `visionhdl.BilateralFilter` 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

```
filt2d = visionhdl.BilateralFilter(Name,Value)
```

Description

`filt2d = visionhdl.BilateralFilter(Name,Value)` returns a bilateral filter System object™. Set properties using name-value pairs. Enclose each property name in single quotes.

For example:

```
filt2d = visionhdl.BilateralFilter('CoefficientsDataType','Custom',...
                    'CustomCoefficientsDataType',numerictype(0,18,17))
```

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.

NeighborhoodSize — Size of image region to average
'3×3' (default) | '5×5' | '7×7' | '9×9' | '11×11' | '13×13' | '15×15'

Size of the image region used to compute the average, specified as an *N*-by-*N* pixel square.

SpatialStdDev — Spatial standard deviation target
0.5 (default) | positive real number

Spatial standard deviation target used to compute coefficients for the spatial Gaussian filter, specified as a positive real number. This parameter has no limits, but recommended values are from 0.1 to 10. At the high end, the distribution becomes flat and the coefficients are small. At the low end, the distribution peaks in the center and has small coefficients in the rest of the neighborhood. These boundary values also depend on the neighborhood size and the data type used for the coefficients.

IntensityStdDev — Intensity standard deviation target
0.5 (default) | positive real number

Intensity standard deviation target used to compute coefficients for the intensity Gaussian filter, specified as a positive real number. This parameter has no limits, but recommended values are from 0.1 to 10. At the high end, the distribution becomes flat and the coefficients are small. At the low end, the distribution peaks in the center and has small coefficients in the rest of the neighborhood. These boundary values also depend on the neighborhood size and the data type used for the coefficients.

When the intensity standard deviation is large, the bilateral filter acts more like a Gaussian blur filter, because the intensity Gaussian has a lower peak. Conversely, when the intensity standard deviation is smaller, edges in the intensity are preserved or enhanced.

PaddingMethod — Method for padding boundary of input image
'Constant' (default) | 'Symmetric' | 'Replicate' | 'Reflection' | 'None'

Methods for padding the boundary of the input image, specified as one of these values.

- 'Constant' — Interpret pixels outside the image frame as having a constant value.
- 'Replicate' — Repeat the value of pixels at the edge of the image.
- 'Symmetric' — Set the value of the padding pixels to mirror the edge of the image.
- 'Reflection' — Set the value of the padding pixels to reflect around the pixel at the edge of the image.
- 'None' — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the *n*/2 pixels around the edge of the frame for downstream operations. *n* is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

For more information about these methods, see "Edge Padding".

PaddingValue — Value used to pad boundary of input image
0 (default) | integer

Value used to pad the boundary of the input image, specified as an integer. The object casts this value to the same data type as the input pixel.

Dependencies

This parameter applies when you set PaddingMethod to 'Constant'.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

RoundingMethod — Rounding mode for fixed-point operations
'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Zero'

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is single or double.

OverflowAction — Overflow mode used for fixed-point operations
'Saturate' (default) | 'Wrap'

Overflow mode used for fixed-point operations. When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This option does not apply when the input data type is single or double.

CoefficientsDataType — Method to determine data type of filter coefficients
'Same as first input' (default) | 'Custom'

Method for determining the data type of the filter coefficients. The coefficients usually require a data type with more precision than the input data type.

- 'Custom' — Sets the data type of the coefficients to match the data type defined in the CustomCoefficientsDataType property.
- 'Same as first input'' — Sets the data type of the coefficients to match the data type of the pixelin argument.

CustomCoefficientsDataType — Data type for the filter coefficients
numerictype(0,16,15) (default) | numerictype(0,WL,FL)

Data type for the filter coefficients, specified as numerictype(0,WL,FL), where *WL* is the word length and *FL* is the fraction length in bits.

Specify an unsigned data type that can represent values less than 1. The coefficients usually require a data type with more precision than the input data type. The object calculates the coefficients based on the neighborhood size and the values of IntensityStdDev and SpatialStdDev. Larger neighborhoods spread the Gaussian function such that each coefficient value is smaller. A larger standard deviation flattens the Gaussian so that the coefficients are more uniform in nature, and a smaller standard deviation produces a peaked response.

Note If you try a data type and after quantization, more than half of the coefficients become zero, the object issues a warning. If all the coefficients are zero after quantization, the object issues an error. These messages mean that the object was unable to express the requested filter by using the data

type specified. To avoid this issue, choose a higher-precision coefficient data type or adjust the standard deviation parameters.

Dependencies

This property applies when you set `CoefficientsDataType` to `'Custom'`.

OutputDataType — Method to determine data type of output pixels
`'Same as first input'` (default) | `'Custom'`

Method to determine data type of output pixels.

- `'Same as first input''` — Sets the data type of the output pixels to match the data type of `pixelin`.
- `'Custom'` — Sets the data type of the output pixels to match the data type defined in the `CustomOutputDataType` property.

CustomOutputDataType — Data type for the output pixels
`numerictype(1,16,15)` (default) | `numerictype(S,WL,FL)`

Data type for the output pixels, specified as `numerictype(S,WL,FL)`, where *S* is `1` (`true`) for signed and `0` (`false`) for unsigned, *WL* is the word length, and *FL* is the fraction length in bits. The filtered pixel values are cast to this data type.

Dependencies

This property applies when you set `OutputDataType` to `'Custom'`.

Usage

Syntax

`[pixelout,ctrlout] = filt2d(pixelin,ctrlin)`

Description

`[pixelout,ctrlout] = filt2d(pixelin,ctrlin)` returns the filtered pixel value and accompanying control signals.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel stream
scalar

Single image pixel in a pixel stream, specified as a scalar value representing intensity. Integer and fixed-point data types larger than 16 bits are not supported.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fi` | `double` | `single`

ctrlin — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

pixelout — Output pixel stream
scalar

Single image pixel in a pixel stream, returned as a scalar value representing intensity. Integer and fixed-point data types larger than 16 bits are not supported.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fi` | `double` | `single`

ctrlout — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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

Create Bilateral Filter for HDL Generation

Load input image and create serializer and deserializer objects.

```
frmOrig = imread('rice.png');
frmActivePixels = 48;
frmActiveLines = 32;
frmIn = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmIn,'InitialMagnification',300)
title 'Input Image'

frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+10,...
      'TotalVideoLines',frmActiveLines+10,...
      'StartingActiveLine',6,...
      'FrontPorch',5);
[~,~,numPixPerFrm] = getparamfromfrm2pix(frm2pix);

pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+10);
```


Input Image

Write a function that creates and calls the System object™. You can generate HDL from this function.

```
function  [pixOut,ctrlOut] = BilatFilt(pixIn,ctrlIn)
```

```
%bilatFilt
% Filters one pixel according to the default spatial and intensity standard
% deviation, 0.5.
% pixIn and pixOut are scalar intensity values.
% ctrlIn and ctrlOut are structures that contain control signals associated
% with the pixel.
% You can generate HDL code from this function.

  persistent filt2d;
  if isempty(filt2d)
    filt2d = visionhdl.BilateralFilter(...
      'CoefficientsDataType','Custom',...
      'CustomCoefficientsDataType',numerictype(0,18,17));

  end
 [pixOut,ctrlOut] = filt2d(pixIn,ctrlIn);
end
```

Filter the image by calling the function for each pixel.

```
pixOutVec = zeros(numPixPerFrm,1,'uint8');
ctrlOutVec = repmat(pixelcontrolstruct,numPixPerFrm,1);

[pixInVec,ctrlInVec] = frm2pix(frmIn);
for p = 1:numPixPerFrm
    [pixOutVec(p),ctrlOutVec(p)] = BilatFilt(pixInVec(p),ctrlInVec(p));
end
[frmOut,frmValid] = pix2frm(pixOutVec,ctrlOutVec);

if frmValid
    figure;
    imshow(frmOut,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image

Version History

Introduced in R2017b

R2022a: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 2, 4, or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
Bilateral Filter

Objects
`visionhdl.FrameToPixels`

visionhdl.BirdsEyeView

Package: `visionhdl`

Transform front-facing camera image into top-down view

Description

The `visionhdl.BirdsEyeView` System object warps a front-facing camera image into a top-down view. It uses a hardware-efficient architecture that supports HDL code generation.

You must provide the homography matrix that describes the transform. This matrix can be calculated from physical camera properties, or empirically derived by analyzing an image of a grid pattern taken by the camera. The object uses the matrix to compute the transformed coordinates of each pixel. The transform does not interpolate between pixel locations. Instead it rounds the result to the nearest coordinate.

The object operates on a trapezoidal region of the input image below the vanishing point. These images show the input region selected for transformation and the resulting top-down view.

You can specify the number of lines in the transformed region and the size of the output frame. If the specified homography matrix cannot map from the requested number of lines to the requested output size, the object returns a warning.

Because the object replicates lines from the input region to create the larger output frame, it cannot complete the transform of one frame before the next frame arrives. The object ignores any new input

frames while it is still transforming the previous frame. Therefore, depending on the stored lines and output size, the object can drop input frames. This timing also enables the object to maintain the blanking intervals of the input pixel stream.

To transform a front-facing camera image to top-down view:

1   Create the `visionhdl.BirdsEyeView` 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

`birdsEyeXfrm = visionhdl.BirdsEyeView(hM,MaxBufferSize,Name,Value)`

Description

`birdsEyeXfrm = visionhdl.BirdsEyeView(hM,MaxBufferSize,Name,Value)` returns a bird's-eye transform System object, with the homography matrix set to `hM`, and a buffer size of `MaxBufferSize` pixels. You can optionally set additional properties using 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.

HomographyMatrix — Transfer function derived from camera parameters
[0.000100990123328 0 0;0.000412396945637 0.001302203393162
1.293171994e-06;-0.103019798961327 -0.255811259450009 -0.000222053779501]
(default) | 3-by-3 matrix

Transfer function derived from camera parameters, specified as a 3-by-3 matrix.

The homography matrix, *h*, is derived from four intrinsic parameters of the physical camera setup: the focal length, pitch, height, and principal point (from a pinhole camera model). The default value is the matrix for the camera setup used in the "Lane Detection" example.

This matrix can be calculated from physical camera properties, or empirically derived by analyzing an image of a grid test pattern taken by the camera. See `estimateGeometricTransform` or "Using the Single Camera Calibrator App" (Computer Vision Toolbox).

MaxBufferSize — Number of input pixels to buffer
40000 (default) | integer

Number of input pixels to buffer, specified as an integer. Compute this value from MaxSourceLinesBuffered*ActivePixelsPerLine. The object uses a memory of this size to store the input pixels. If you specify a value that is not a power of two, the object uses the next largest power of two.

MaxSourceLinesBuffered — Number of lines to transform
54 (default) | integer

Number of lines to transform, specified as an integer. The object stores and transforms this number of lines into the output bird's-eye view image, starting at the vanishing point as determined by the HomographyMatrix.

Storing the full input frame uses too much memory to implement the algorithm without off-chip storage. Therefore, for a hardware implementation, choose a smaller region to store and transform, one that generates an acceptable output frame size.

For example, using the default HomographyMatrix with an input image of 640-by-480 pixels, the full-sized transform results in a 900-by-640 output image. Analysis of the input-to-output *x*-coordinate mapping shows that around 50 lines of the input image are required to generate the top 700 lines of the bird's-eye view output image. This number of input lines can be stored using on-chip memory. The vanishing point for the default camera setup is around line 200, and lines above that point do not contribute to the resulting bird's-eye view. Therefore, the object can store only input lines 200–250 for transformation.

BirdEyeActivePixels — Horizontal size of output frame
640 (default) | integer

Horizontal size of output frame, specified as an integer. This parameter is the number of active pixels in each output line.

BirdEyeActiveLines — Vertical size of output frame
700 (default) | integer

Vertical size of output frame, specified as an integer. This parameter is the number of active lines in each output frame.

Usage

Syntax

[pixelout,ctrlout] = birdsEyeXfrm(pixelin,ctrlin)

Description

[pixelout,ctrlout] = birdsEyeXfrm(pixelin,ctrlin) returns the bird's-eye view transformation of the input stream. The frame size of the output stream corresponds to the size you configured in the BirdsEyeViewPixels and BirdsEyeViewLines properties.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the

`visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel stream
scalar

Single image pixel in a pixel stream, specified as a scalar value representing intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fi` | `logical` | `double` | `single`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel stream
scalar

Single image pixel in a pixel stream, returned as a scalar value representing intensity.

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

Data Types: `uint` | `int` | `fi` | `logical` | `double` | `single`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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

The transform from input pixel coordinate *(x,y)* to the bird's-eye pixel coordinate is derived from the homography matrix, *h*. The homography matrix is based on physical parameters and therefore is a constant for a particular camera installation.

$$(\widehat{x}, \widehat{y}) = round\left(\frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}}, \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}}\right)$$

The implementation of the bird's-eye transform in hardware does not directly perform this calculation. Instead, the object precomputes lookup tables for the horizontal and vertical aspects of the transform.
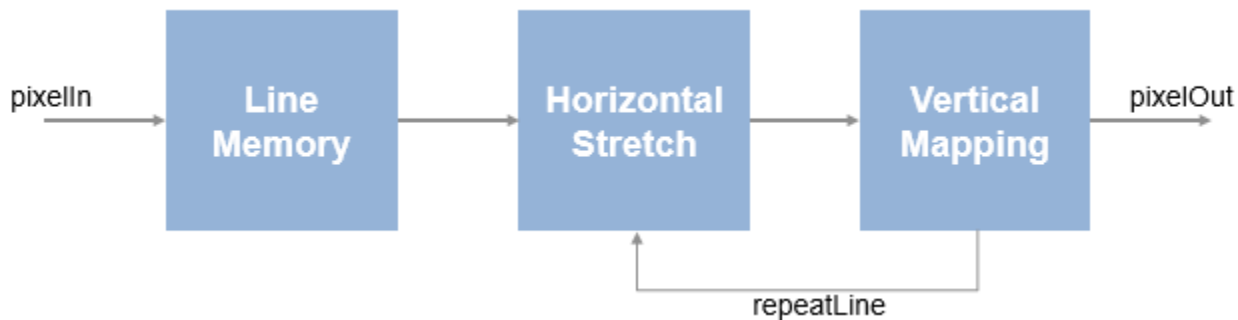
First, the object stores the input lines starting from the precomputed vanishing point. The stored pixels form a trapezoid, with short lines near the vanishing point and wider lines near the camera. This storage uses `MaxBufferSize` memory locations.

The horizontal lookup table contains interpolation parameters that describe the stretch of each line of the trapezoidal input region to the requested width of the output frame. Lines that fall closer to the vanishing point are stretched more than lines nearer to the camera.

The vertical lookup table contains the *y*-coordinate mapping, and how many times each line is repeated to fill the requested height of the output frame. Near the vanishing point, one input line maps to many output lines, while each line nearer the camera maps to a diminishing number of output lines.

The lookup tables use 3*`MaxSourceLinesBuffered` memory locations.

Version History
Introduced in R2017b

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. For more information about acceleration, see "Accelerate Pixel-Streaming

Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
Birds-Eye View

Objects
`visionhdl.FrameToPixels`

Functions
`imwarp` | `estimateGeometricTransform`

Topics
"Using the Single Camera Calibrator App" (Computer Vision Toolbox)

visionhdl.ChromaResampler

Package: visionhdl

Downsample or upsample chrominance component

Description

The visionhdl.ChromaResampler System object downsamples or upsamples a pixel stream.

- Downsampling reduces bandwidth and storage requirements in a video system by combining pixel chrominance components over multiple pixels. You can specify a filter to prevent aliasing, by selecting the default filter or by entering coefficients.
- Upsampling restores a signal to its original rate. You can use interpolation or replication to calculate the extra sample.

The object accepts luma and the chrominance components. The object does not modify the luma component and applies delay to align with the resampled chrominance outputs. The rate of the output luma component is the same as the input.

To resample the chrominance component of a pixel stream:

1 Create the visionhdl.ChromaResampler 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

CR = visionhdl.ChromaResampler(Name,Value)

Description

CR = visionhdl.ChromaResampler(Name,Value) returns a chroma resampler System object that resamples the chrominance component of a pixel stream. Set 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.

Resampling — Type of resampling
'4:4:4 to 4:2:2' (default) | '4:2:2 to 4:4:4'

Resampling format, specified as one of these values.

- '4:4:4 to 4:2:2' — Perform a downsampling operation.
- '4:2:2 to 4:4:4' — Perform an upsampling operation.

AntialiasingFilterSource — Lowpass filter to follow downsample operation
'Auto' (default) | 'Property' | 'None'

Lowpass filter to accompany a downsample operation, specified as one of these values.

- 'Auto' — Use the built-in lowpass filter.
- 'Property' — Filter using the coefficients in theHorizontalFilterCoefficients property.
- 'None' — Do not filter the input signal.

Dependencies

This property applies when you set Resampling to '4:4:4 to 4:2:2'.

HorizontalFilterCoefficients — Coefficients for the antialiasing filter
[0.2,0.6,0.2] (default) | vector

Coefficients for the antialiasing filter, specified as a vector.

Dependencies

This property applies when you set Resampling to '4:4:4 to 4:2:2' and
AntialiasingFilterSource to 'Property'.

InterpolationFilter — Interpolation method for an upsample operation
'Linear' (default) | 'Pixel replication'

Interpolation method for an upsample operation, specified as one of these values.

- 'Linear' — Use linear interpolation to calculate the missing values.
- 'Pixel replication' — Repeat the chrominance value of the preceding pixel to create the missing pixel.

Dependencies

This property applies when you set Resampling to '4:2:2 to 4:4:4'.

RoundingMethod — Rounding mode for fixed-point operations
'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Zero'

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is single or double.

OverflowAction — Overflow mode used for fixed-point operations
'Wrap' (default) | 'Saturate'

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is single or double.

CustomCoefficientsDataType — Data type for antialiasing filter coefficients
numerictype(1,16,15) (default) | numerictype(S,WL,FL)

Data type for the antialiasing filter coefficients, specified as numerictype(S,WL,FL), where *S* is 1 (true) for signed and 0 (false) for unsigned, *WL* is the word length, and *FL* is the fraction length in bits.

Dependencies

This parameter applies when you set AntialiasingFilterSource to 'Property' or 'Auto'.

Usage

Syntax

[pixelout,ctrlout] = CR(pixelin,ctrlin)

Description

[pixelout,ctrlout] = CR(pixelin,ctrlin) computes the next output pixel, pixelout, in the resampled video stream. The pixel data arguments, pixelin and pixelout, are vectors of three values representing a pixel in Y'CbCr color space. The object passes through the luma component and control signals, ctrlin, aligned with the output pixel stream.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a three-component vector that represents a single pixel and a structure that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the visionhdl.FrameToPixels object. For a full description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel
vector

Input pixel in gamma-corrected Y'CbCr color space, specified as a vector of three values.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: uint8 | uint16 | fixdt(0,N,0), where N = 8,9,...,16 | single | double

ctrlin — Control signals accompanying input pixel stream
pixelcontrol structure

Control signals accompanying the input pixel stream, specified as a pixelcontrol structure containing five logical data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: struct

Output Arguments

`pixelout` — Output pixel
vector

Output pixel in gamma-corrected Y'CbCr color space, returned as a vector of three values.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `fixdt(0,N,0), where N = 8,9,...,16` | `single` | `double`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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 Y'CbCr Image

Resample a 4:4:4 Y'CbCr image to 4:2:2. The example also shows how to convert an R'G'B' input image to Y'CbCr color space.

Prepare a test image by selecting a portion of an image file.

```
frmActivePixels = 64;
frmActiveLines = 48;
frmOrig = imread('fabric.png');
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels,:);
```

Create a serializer and specify the size of inactive pixel regions. The number of padding pixels on each line must be greater than the latency of each pixel-processing object.

```
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',3, ...
      'VideoFormat','custom', ...
```

```
        'ActivePixelsPerLine',frmActivePixels, ...
        'ActiveVideoLines',frmActiveLines, ...
        'TotalPixelsPerLine',frmActivePixels+40, ...
        'TotalVideoLines',frmActiveLines+10, ...
        'StartingActiveLine',6, ...
        'FrontPorch',5);
```

Create a color space converter and resampler, using the default property values. The default conversion is `'RGB to YCbCr'`. The default resampling mode is `'4:4:4 to 4:2:2'`. The default antialiasing filter is a 29-tap lowpass filter. This filter gives the object a latency of 30 cycles.

```
convert2ycbcr = visionhdl.ColorSpaceConverter();
downsampler = visionhdl.ChromaResampler();
```

Serialize the test image using the serializer object. `pixIn` is a `numPixelsPerFrame` -by-3 matrix. `ctrlIn` is a vector of control signal structures. Preallocate vectors for the output signals.

```
[pixIn,ctrlIn] = frm2pix(frmInput);

[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pix444 = zeros(numPixelsPerFrame,3,'uint8');
ctrl444  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
pix422 = zeros(numPixelsPerFrame,3,'uint8');
ctrl422 = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the stream, convert to Y'CbCr, then downsample.

```
for p = 1:numPixelsPerFrame
    [pix444(p,:),ctrl444(p)] = convert2ycbcr(pixIn(p,:),ctrlIn(p));
    [pix422(p,:),ctrl422(p)] = downsampler(pix444(p,:),ctrl444(p));
end
```

Create deserializers with a format matching that of the serializer. Convert the 4:4:4 and 4:2:2 pixel streams back to image frames.

```
pix2frm444 = visionhdl.PixelsToFrame( ...
        'NumComponents',3, ...
        'VideoFormat','custom', ...
        'ActivePixelsPerLine',frmActivePixels, ...
        'ActiveVideoLines',frmActiveLines, ...
        'TotalPixelsPerLine',frmActivePixels+40);

pix2frm422 = visionhdl.PixelsToFrame( ...
        'NumComponents',3, ...
        'VideoFormat','custom', ...
        'ActivePixelsPerLine',frmActivePixels, ...
        'ActiveVideoLines',frmActiveLines, ...
         'TotalPixelsPerLine',frmActivePixels+40);

[frm444,frmValid1] = pix2frm444(pix444,ctrl444);
[frm422,frmValid2] = pix2frm422(pix422,ctrl422);
```

The 4:2:2 and 4:4:4 pixel streams and frames have the same number of pixels. To examine the resampled data, regroup the pixel data for the first 8 pixels of the first line. The first row is the Y elements of the pixels, the second row is the Cb elements, and the third row is the Cr elements. In the 4:2:2 data, the Cb and Cr elements change only every second sample.

```
YCbCr444 = [frm444(1,1:8,1); frm444(1,1:8,2); frm444(1,1:8,3)]
```

```
YCbCr444 = 3x8 uint8 matrix

  132   134   129   124   125   122   118   119
  116   118   119   122   122   121   123   123
  135   131   125   121   119   116   118   118
```

```
YCbCr422 = [frm422(1,1:8,1); frm422(1,1:8,2); frm422(1,1:8,3)]
```

```
YCbCr422 = 3x8 uint8 matrix

  132   134   129   124   125   122   118   119
  116   116   120   120   122   122   123   123
  135   135   126   126   119   119   118   118
```

```
figure
imshow(frm422,'InitialMagnification',300)
title '4:2:2'
```
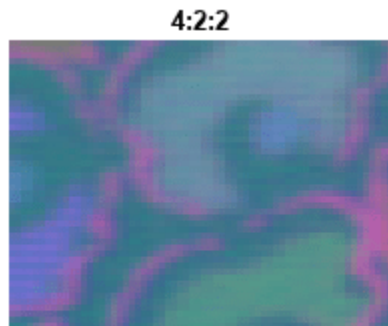


```
figure
imshow(frm444,'InitialMagnification',300)
title '4:4:4'
```
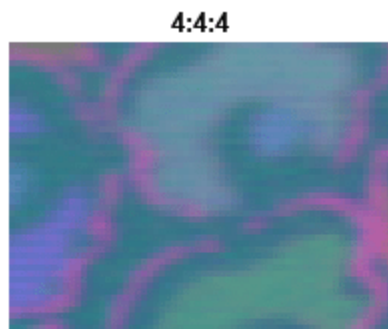
Algorithms

This object implements the algorithms described on the Chroma Resampler block reference page.

Version History

Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The object now supports multipixel streams that have 2 pixels per clock cycle.

R2021b: Multipixel streaming

The object now supports multipixel streams. The object accepts input matrices of *NumPixels*-by-3 values, and output matrices of *NumPixels*-by-*NumComponents* values, where *NumComponents* is 3 or 1 and *NumPixels* is 4 or 8. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also
Chroma Resampler | `vision.ChromaResampler` | `visionhdl.FrameToPixels`

visionhdl.ColorSpaceConverter

Package: visionhdl

Convert color information between color spaces

Description

The `visionhdl.ColorSpaceConverter` System object converts between R'G'B' and Y'CbCr color spaces, and also converts R'G'B' to intensity.

Note The `ColorSpaceConverter` System object operates on gamma-corrected color spaces. However, to simplify use of the System object, the property arguments do not include the prime notation.

To convert color information between color spaces:

1   Create the `visionhdl.ColorSpaceConverter` 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

```
CSC = visionhdl.ColorSpaceConverter(Name,Value)
```

Description

`CSC = visionhdl.ColorSpaceConverter(Name,Value)` returns a System object that converts between color spaces. Set 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.

Conversion — Type of color space conversion
`'RGB to YCbCr'` (default) | `'YCbCr to RGB'` | `'RGB to intensity'`

Type of color space conversion. The object accepts input as a vector of three values representing a single pixel. If you choose `'RGB to intensity'`, the output is a scalar value. Otherwise, the output is a vector of three values.

`ConversionStandard` — Conversion equation
`'Rec. 601 (SDTV)'` (default) | `'Rec. 709 (HDTV)'`

Conversion equation to use on the input video stream.

Dependencies

This property does not apply when you set `Conversion` to `'RGB to intensity'`.

`ScanningStandard` — HDTV scanning standard
`'1250/50/2:1'` (default) | `'1125/60/2:1'`

Scanning standard to use for HDTV conversion.

Dependencies

This property applies when you set `ConversionStandard` to `'Rec. 709 (HDTV)'`.

Usage

Syntax

`[pixelout,ctrlout] = CSC(pixelin,ctrlin)`

Description

`[pixelout,ctrlout] = CSC(pixelin,ctrlin)` converts a single pixel from one color space to another. The input `pixelin` is a vector of three values representing one pixel in R'G'B' or Y'CbCr color space. If the `Conversion` property is set to `'RGB to YCbCr'` or `'YCbCr to RGB'`, then the output `pixelout` is a vector of three values representing one pixel. If the `Conversion` property is set to `'RGB to intensity'`, then the output `pixelout` is a scalar value representing one pixel.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a three-component vector that represents a single pixel and a structure that contains five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a full description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel
vector

Input pixel in gamma-corrected R'G'B' or Y'CbCr color space, specified as a vector of three unsigned integer values.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `fixdt(0,N,0), where N = 8,9,...,16` | `single` | `double`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel
scalar | vector

Output pixel returned as a vector of three unsigned integer values, or a scalar unsigned integer value. The data type of the output pixel is the same as the input pixel.

- If you set the `Conversion` property to `'RGB to YCbCr'` or `'YCbCr to RGB'`, then `pixelout` is a vector representing the pixel in gamma-corrected color space.

- If you set the `Conversion` property to `'RGB to intensity'`, then `pixelout` is a scalar representing pixel intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `fixdt(0,N,0), where N = 8,9,...,16` | `single` | `double`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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

Convert Color Image to Grayscale

This example shows how to convert pixel stream data to a different color space.

Set the dimensions of the test image and load a color source image. Select a portion of the image matching the desired test size.

```
frmActivePixels = 64;
frmActiveLines = 48;
frmOrig = imread('fabric.png');
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels,:);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Input Image

Create a serializer object and specify the size of the inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',3, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',6, ...
      'FrontPorch',5);
```

Create a color space converter object. Select a conversion from R'G'B' to grayscale.

```
convertrgb2gray = visionhdl.ColorSpaceConverter( ...
      'Conversion','RGB to intensity');
```

Serialize the test image. `pixIn` is a numPixelsPerFrame-by-3 matrix. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Set up variables, and convert each pixel in the stream to the new color space.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
for p = 1:numPixelsPerFrame
    [pixOut(p),ctrlOut(p)] = convertrgb2gray(pixIn(p,:),ctrlIn(p));
end
```

Create a deserializer object with a format that matches the format of the serializer. Convert the pixel stream to an image frame, and display the grayscale output image.

```
pix2frm = visionhdl.PixelsToFrame( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image
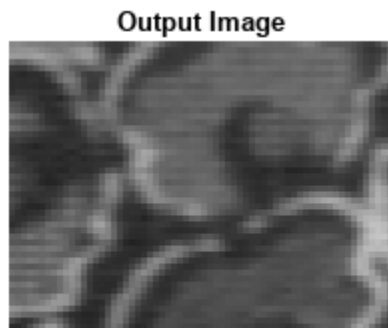
Algorithms

This System object implements the algorithms described on the Color Space Converter block reference page.

Version History
Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The object now supports multipixel streams that have 2 pixels per clock cycle.

R2021b: Multipixel streaming

The object now supports multipixel streams. The object accepts input matrices of *NumPixels*-by-3 values, and output matrices of *NumPixels*-by-*NumComponents* values, where *NumComponents* is 3 or 1 and *NumPixels* is 4 or 8. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also
Color Space Converter | rgb2ycbcr | ycbcr2rgb | rgb2gray | visionhdl.FrameToPixels

visionhdl.CornerDetector

Package: visionhdl

Finds corners using FAST or Harris algorithm

Description

The visionhdl.CornerDetector System object detects corners by using the features-from-accelerated-segment test (FAST) algorithm or by the intersecting edges (Harris) algorithm. For each pixel, if the pixel is a corner, the object returns the corner metric. If the pixel is not a corner, the object returns a pixel value of zero.

The FAST algorithm tests a circular area around the potential center of the corner. The test detects a corner if a contiguous section of pixels are either brighter than the center plus a threshold or darker than the center minus a threshold. You can specify a minimum contrast threshold as a property or argument and select from three rules to define a corner. These rules specify how many pixels in a circle of pixels must exceed the minimum contrast for the center pixel to be considered a corner. The object performs parallel tests of all combinations of contiguous pixels around the circle. The FAST algorithm uses few hardware resources.

The Harris algorithm computes horizontal and vertical gradients, filters the gradient components with a circular 5-by-5 Gaussian filter, and computes a metric that represents the strength of the corner. You can specify a threshold that determines the level at which corners are detected. The object returns a corner for pixels where the metric exceeds this threshold. The Harris algorithm uses more hardware resources than the FAST algorithm, but can detect corners that the FAST algorithm might not find.

To detect corners in a pixel stream:

1   Create the visionhdl.CornerDetector 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

corners = visionhdl.CornerDetector(Name,Value)

Description

corners = visionhdl.CornerDetector(Name,Value) returns a corner detector System object. Set properties using one or more name-value pairs. Enclose each property name in single quotes.

For example:

```
corners = visionhdl.CornerDetector('Method','FAST 7 of 12',...
              'MinContrastSource','Input port',...
              'PaddingMethod','None')
```

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.

Method — Detection method
`'FAST 5 of 8'` (default) | `'FAST 7 of 12'` | `'FAST 9 of 16'` | `'Harris'`

Select the Harris algorithm, or the size of the circle for the FAST algorithm. The three FAST options specify how many contiguous pixels on the circle must exceed the threshold for the center pixel to be marked as a corner. For the circles of 8, 12, and 16 pixels shown in these figures, the object detects a corner when 5, 7, or 9 contiguous pixels,respectively, are above the threshold. The object checks for this number of contiguous pixels anywhere on the circle. The object uses a kernel of size 3-by-3 for `'FAST 5 of 8'`, 5-by-5 for `'FAST 7 of 12'`, and 7-by-7 for `'FAST 9 of 16'`.

MinContrastSource — Source for threshold that indicates corner for FAST algorithm
`'Property'` (default) | `'Input port'`

Specify `'Property'` to set the contrast threshold by using the `MinContrast` property. Specify `'Input port'` to enable the `thresh` input argument.

Dependencies

To enable this property, set the `Method` property to one of the FAST options.

MinContrast — Threshold that indicates corner for FAST algorithm
20 (default) | nonnegative integer

Each pixel in the circle is subtracted from the center pixel value, and only those differences that exceed this threshold are used to determine if the center pixel is a corner and to calculate the output metric. This value is cast to the same data type as `pixelin`.

Dependencies

To enable this property, set the `Method` property to one of the FAST options and set the `MinContrastSource` property to `'Property'`.

ThresholdSource — Source for threshold that indicates corner for Harris algorithm

`'Property'` (default) | `'Input port'`

Specify `'Property'` to set the threshold by using the `Threshold` property. Specify `'Input port'` to enable the `thresh` input argument.

Dependencies

To enable this property, set the `Method` property to `'Harris'`.

Threshold — Threshold that indicates corner for Harris algorithm

`15000` (default) | nonnegative integer

This value represents an approximation of the eigenvalues of the Harris matrix calculated from the gradient results. The object returns a corner for pixels where the metric exceeds this threshold. This value is cast to the same data type as the output `corner`. The corner metric is in the range of a fourth power of the input pixel values. For example, for an 8-bit input pixel, the full-precision output size is 44 bits.

Dependencies

To enable this property, set the `Method` property to `'Harris'` and set the `ThresholdSource` property to `'Property'`.

LineBufferSize — Size of line buffer
`2048` (default) | integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (*N* – 1)-by-`LineBufferSize` memory locations to store the pixels. *N* is the size of the kernel.

The object uses a kernel of size 3-by-3 when the `Method` property is `'FAST 5 of 8'`, 5-by-5 when `Method` is `'FAST 7 of 12'`, and 7-by-7 when `Method` is `'FAST 9 of 16'`.

When you set `Method` to `'Harris'`, the object uses one 1-by-3 kernel and one 3-by-1 kernel for the gradient calculation and three 5-by-5 kernels for the circular window filter on the three gradient components.

PaddingMethod — Method for padding
`'Symmetric'` (default) | `'Replicate'` | `'Reflection'` | `'None'`

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- **'Symmetric'** — Set the value of the padding pixels to mirror the edge of the image. This option prevents corners from being detected at the boundaries of the active frame.

- **'Replicate'** — Repeat the value of pixels at the edge of the image.

- **'Reflection'** — Set the value of the padding pixels to reflect around the pixel at the edge of the image.

- **'None'** — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the *n*/2 pixels around the edge of the frame for downstream operations. *n* is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

The circular window filters used in the Harris algorithm pad the edges of the frame using 'Replicate' padding.

RoundingMethod — Rounding mode for fixed-point operations
'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Zero'

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is single or double.

OverflowAction — Overflow mode used for fixed-point operations
'Wrap' (default) | 'Saturate'

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is single or double.

OutputDataType — Method to determine data type of output corner metric
'Same as first input' (default) | 'Custom' | 'Full precision'

Method to determine data type of output corner metric.

- **'Same as first input'** — Sets the data type of the corner metric to match the data type of pixelin.

- **'Custom'** — Sets the data type of the corner metric to match the data type defined in the CustomOutputDataType property.

- **'Full precision'** — Computes internal and output data types using full precision rules. These rules provide accurate fixed-point numerics and prevent quantization within the object. Bits are added, as needed, to prevent rounding and overflow.

CustomOutputDataType — Data type for the output corner metric
numerictype(0,8,0) (default) | numerictype(S,WL,FL)

Data type for the output corner metric, specified as numerictype(S,WL,FL), where *S* is 1 (true) for signed and 0 (false) for unsigned, *WL* is the word length, and *FL* is the fraction length in bits. The object cases the corner metric to this data type.

Dependencies

This property applies when you set OutputDataType to 'Custom'.

Usage

Syntax

```
[corner,ctrlout] = corners(pixelin,ctrlin)
[corner,ctrlout] = corners(pixelin,thresh,ctrlin)
```

Description

`[corner,ctrlout] = corners(pixelin,ctrlin)` returns the metric that indicates whether each pixel is a corner

`[corner,ctrlout] = corners(pixelin,thresh,ctrlin)` specifies the threshold that determines a corner as an input argument. This syntax applies for both Harris and FAST algorithms. The threshold is interpreted differently for each algorithm. To enable this argument when using the FAST algorithm, set the `MinContrastSource` property to `'Input port'`. To enable this argument when using the Harris algorithm, set the `ThresholdSource` property to `'Input port'`.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel stream
scalar

Single image pixel in a pixel stream, specified as a scalar value representing intensity.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fi` | `double` | `single`

ctrlin — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

thresh — Threshold that indicates a corner
nonnegative integer

Threshold that indicates a corner, specified as a nonnegative integer.

When you choose the FAST algorithm, each pixel in the circle is subtracted from the center pixel value. Only those differences that exceed this threshold are used to determine if the center pixel is a corner and to calculate the output metric. This value is cast to the same data type as `pixelin`.

When you choose the Harris algorithm, this value represents an approximation of the eigenvalues of the Harris matrix calculated from the gradient results. The object returns a corner for pixels where the eigenvalue metric exceeds this threshold. The Harris corner metrics are in the range of a fourth power of the input pixel values. For example, for an 8-bit input pixel, the full-precision output size is 44 bits. This value is cast to the same data type as the output `corner`.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Dependencies

To enable this argument when you choose the FAST algorithm, set the `MinContrastSource` property to `'Input port'`.

To enable this argument when you choose the Harris algorithm, set the `ThresholdSource` property to `'Input port'`.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

corner — Metric that indicates whether pixel is corner
scalar

Metric that indicates whether the pixel is a corner, returned as a scalar. The type is determined by the `OutputDataType` property. When the pixel is not a corner, the object returns zero for this value.

For corners detected with the FAST algorithm, this value is a contrast metric computed by using the sum-of-absolute-difference (SAD) of the pixels around the circle with the center pixel value. The metric includes only those differences that exceed the threshold.

For corners detected with the Harris algorithm, this value is the approximation of the eigenvalues of the Harris matrix calculated from the gradient results.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fi` | `double` | `single`

ctrlout — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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 object implements the algorithms described on the Corner Detector block reference page.

Version History

Introduced in R2020a

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020b: Harris corner detection

The object now provides a choice between the FAST algorithm and the Harris and Stephens interconnecting edges algorithm.

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
Corner Detector

Objects
visionhdl.FrameToPixels

visionhdl.Closing

Package: visionhdl

Morphologically closes binary pixel stream

Description

The visionhdl.Closing System object morphologically closes a binary pixel stream. This operation morphological dilates and then morphologically erodes each pixel, by using the same neighborhood for both calculations. The object operates on a stream of binary intensity values.

To morphologically close a binary pixel stream:

1 Create the visionhdl.Closing 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

```
closer = visionhdl.Closing
closer = visionhdl.Closing(Name,Value)
```

Description

closer = visionhdl.Closing returns a System object that morphologically closes a binary pixel stream.

closer = visionhdl.Closing(Name,Value) sets properties using one or more name-value arguments. For example, 'Neighborhood',getnhood(strel('disk',4)) specifies a 4-by-4 disk-pattern neighborhood.

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.

Neighborhood — Pixel neighborhood
[0,1,0;1,1,1;0,1,0] (default) | vector or matrix of binary values

Pixel neighborhood, specified as a vector or matrix of binary values.

The object supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the Neighborhood property to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (*n* – 1)-by-LineBufferSize memory locations to store the pixels, where *n* is the number of lines in the Neighborhood property value.

PaddingMethod — Method for padding boundary of input image
'Constant' (default) | 'None'

Method for padding the boundary of the input image, specified as one of these values.

- 'Constant' — The object pads the image with zeros for the dilation operation and with ones for the erosion operation. These values prevent closing at the boundaries of the active frame.

- 'None' — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the *n*/2 pixels around the edge of the frame for downstream operations. *n* is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

For more information about these methods, see "Edge Padding".

Usage

Syntax

[pixelout,ctrlout] = closer(pixelin,ctrlin)

Description

[pixelout,ctrlout] = closer(pixelin,ctrlin) returns the next binary pixel value, pixelout, resulting from morphologically closing the neighborhood around each input binary pixel, pixelin.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the visionhdl.FrameToPixels object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel
`0` or `false` | `1` or `true`

Input pixel, specified as a `logical` value of `0`(`false`) or `1` (`true`).

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Data Types: `logical`

ctrlin — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

pixelout — Output pixel
`0` | `1`

Output pixel that is transformed by a morphological operation, returned as a `logical` value.

Data Types: `logical`

ctrlout — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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

Morphologically Close Image

Load a source image from a file. Select a portion of the image that matches the desired test size. This source image contains pixel intensity values of `uint8` data type. Apply a threshold to convert the pixel data to binary values.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
frmInput = frmInput>128;
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Input Image

Create a serializer System object™ and define inactive pixel regions. Set the number of inactive pixels following each active line to at least double the horizontal size of the neighborhood. Set the number of lines following each frame to at least double the vertical size of the neighborhood.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',3, ...
      'FrontPorch',10);
```

Create a morphological close System object.

```
 closing = visionhdl.Closing(...
          'Neighborhood',getnhood(strel('disk',4)));
```

Serialize the test image by calling the serializer object. `pixin` is a vector of intensity values. `ctrlin` is a vector of control signal structures.

```
[pixin,ctrlin] = frm2pix(frmInput);
```

Prepare to process the pixel stream by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixout = false(numPixelsPerFrame,1);
ctrlout  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlin(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixout(p),ctrlout(p)] = closing(pixin(p),ctrlin(p));
    if (ctrlout(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
objLatency_cycles = foundValOut - foundValIn

objLatency_cycles = 540
```

Create a deserializer System object with a format that matches the serializer format. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+20);
[frmOutput,frmValid] = pix2frm(pixout,ctrlout);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```

Output Image
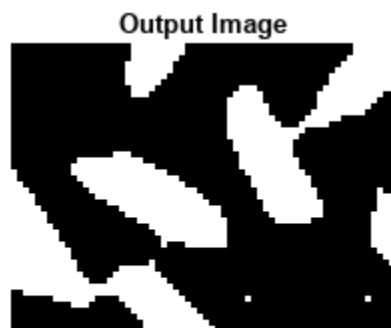
Algorithms

This object implements the algorithms described on the Closing block reference page.

Version History

Introduced in R2015a

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

R2020a: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Objects
visionhdl.Erosion | visionhdl.Opening | visionhdl.Dilation |
visionhdl.FrameToPixels

Blocks
Closing

Functions
imclose

visionhdl.GrayscaleClosing

Package: visionhdl

Morphologically close grayscale pixel stream

Description

The `visionhdl.GrayscaleClosing` System object morphologically closes a grayscale pixel stream. This operation morphologically dilates and then morphologically erodes each pixel, by using the same neighborhood for both calculations. The object operates on a stream of grayscale pixel intensity values.

You can specify a neighborhood, or structuring element, of up to 32-by-32 pixels. For line, square, or rectangle structuring elements that are more than 8 pixels wide, the object uses the Van Herk algorithm to find the maximum and minimum. For structuring elements that are less than 8 pixels wide, or that contain zero elements, the object implements a pipelined comparison tree to find the maximum and minimum.

To morphologically close a grayscale pixel stream:

1   Create the `visionhdl.GrayscaleClosing` 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

```
closer = visionhdl.GrayscaleClosing
closer = visionhdl.GrayscaleClosing(Name,Value)
```

Description

`closer = visionhdl.GrayscaleClosing` returns a System object that morphologically closes a grayscale pixel stream.

`closer = visionhdl.GrayscaleClosing(Name,Value)` sets properties using one or more name-value arguments. For example, `'Neighborhood',getnhood(strel('disk',4))` specifies a 4-by-4 disk-pattern neighborhood.

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.

Neighborhood — Pixel neighborhood
ones(3,3) (default) | vector or matrix of binary values

Pixel neighborhood, specified as a vector or matrix of binary values.

The object supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the Neighborhood property to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function. The minimum neighborhood size is a 2-by-2 matrix, or a 2-element column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (*n* – 1)-by-LineBufferSize memory locations to store the pixels, where *n* is the number of lines in the Neighborhood property value.

Usage

Syntax

[pixelout,ctrlout] = closer(pixelin,ctrlin)

Description

[pixelout,ctrlout] = closer(pixelin,ctrlin) returns the next pixel intensity value, pixelout, resulting from morphologically closing the neighborhood around each input pixel intensity value, pixelin.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the visionhdl.FrameToPixels object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel
unsigned integer

Input pixel, specified as an unsigned integer.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt(0,N,0)`

ctrlin — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

pixelout — Output pixel
unsigned integer

Output pixel that is transformed by a morphological operation, returned as an unsigned integer.

The data type is the same as the data type of `pixelin`.

ctrlout — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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
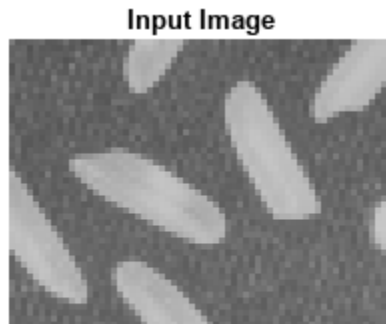
Examples

Grayscale Morphological Closing

Perform morphological closing on a grayscale thumbnail image.

Load a source image from a file. Select a portion of the image matching the desired test size.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
```

```
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```

Input Image

Create a serializer object and define inactive pixel regions. Set the number of inactive pixels following each active line to at least double the horizontal size of the neighborhood. Set the number of lines following each frame to at least double the vertical size of the neighborhood.

```
frm2pix = visionhdl.FrameToPixels(...
        'NumComponents',1, ...
        'VideoFormat','custom',...
        'ActivePixelsPerLine',frmActivePixels, ...
        'ActiveVideoLines', frmActiveLines,...
        'TotalPixelsPerLine',frmActivePixels+20, ...
        'TotalVideoLines',frmActiveLines+20, ...
        'StartingActiveLine',3, ...
        'FrontPorch',10);
```

Create a filter object.

```
mclose = visionhdl.GrayscaleClosing( ...
            'Neighborhood',ones(5,5));
```

Serialize the test image by calling the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = uint8(zeros(numPixelsPerFrame,1));
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
```

```
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = mclose(pixIn(p),ctrlIn(p));
    if (ctrlOut(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
objLatency_cycles = foundValOut-foundValIn

objLatency_cycles = 384
```

Create a deserializer object with a format matching that of the serializer. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image

Algorithms

This object implements the algorithms described on the Grayscale Closing block reference page.

Version History
Introduced in R2016a

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Objects
visionhdl.GrayscaleErosion | visionhdl.GrayscaleDilation | visionhdl.GrayscaleOpening | visionhdl.FrameToPixels

Blocks
Grayscale Closing

Functions
imclose

visionhdl.DemosaicInterpolator

Package: visionhdl

Construct full RGB pixel data from Bayer pattern pixel stream

Description

The `visionhdl.DemosaicInterpolator` System object constructs the full RGB pixel values from a Bayer pattern pixel stream. You can select a low-complexity bilinear interpolation or a moderate-complexity gradient-corrected bilinear interpolation. The object implements the calculations using hardware-efficient algorithms for HDL code generation.

- The object performs bilinear interpolation on a 3-by-3 pixel window using only additions and bit shifts.
- The object performs gradient correction on a 5-by-5 pixel window. The object implements the calculation using bit shifts, additions, and low-order canonical signed digit (CSD) multiplication.

To construct full RGB pixel data from a Bayer pattern pixel stream:

1 Create the `visionhdl.DemosaicInterpolator` 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

```
bayerInterpolator = visionhdl.DemosaicInterpolator
bayerInterpolator = visionhdl.DemosaicInterpolator(Name,Value)
```

Description

`bayerInterpolator = visionhdl.DemosaicInterpolator` creates a System object that interpolates RGB data from a Bayer pattern pixel stream.

`bayerInterpolator = visionhdl.DemosaicInterpolator(Name,Value)` sets properties using one or more name-value arguments. For example, `'SensorAlignment','RGGB'` specifies the pattern of the RGB pixels in the input stream.

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.

InterpolationAlgorithm — Algorithm to calculate the missing pixel values
'Gradient-corrected linear' (default) | 'Bilinear'

Algorithm to calculate the missing pixel values, specified as one of these values.

• 'Bilinear' — Average of the pixel values in the surrounding 3-by-3 neighborhood

• 'Gradient-corrected linear' (default) — Bilinear average, corrected for intensity gradient

SensorAlignment — Color sequence of the input pixels
'RGGB' (default) | 'GBRG' | 'GRBG' | 'BGGR'

Color sequence of the input pixels, specified as 'RGGB', 'GBRG', 'GRBG', or 'BGGR'. These values consist of four letters, R, G, or B, that correspond to the 2-by-2 block of pixels in the top-left corner of the input image. Specify the sequence in left-to-right, top-to-bottom order. For example, the default value, 'RGGB', represents an image with this pattern.

R	G
G	B

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

• When you set the InterpolationAlgorithm property to 'Bilinear', the object allocates 2-by-LineBufferSize memory locations.

• When you set the InterpolationAlgorithm property to 'Gradient-corrected linear', the object allocates 4-by-LineBufferSize memory locations.

Usage

Syntax

[pixelout,ctrlout] = bayerInterpolator(pixelin,ctrlin)

Description

[pixelout,ctrlout] = bayerInterpolator(pixelin,ctrlin) interpolates the missing color values of a Bayer pattern input pixel stream, and returns the next pixel value, pixelout, as a vector of RGB values. pixelin represents one pixel in a Bayer pattern image.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its

location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel stream
scalar

Single input pixel, specified as an unsigned scalar integer. Images in the Bayer format have one color component for each pixel location. Select the sequence of R, G, and B pixels by using the `SensorAlignment` property.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `fi(0,W,F)` | `single` | `double`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel stream
scalar

Single output pixel, returned as a three-element vector of values in RGB color space.

The data type of `pixelout` is the same as the data type of `pixelin`.

Data Types: `uint` | `fi(0,W,F)` | `single` | `double`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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

Demosaic Bayer Pattern Image

This example shows how to construct full RGB pixel data from a Bayer pattern thumbnail image.

Set the dimensions of the test image. Load the source image file. This image is represented in a Bayer pattern, meaning each pixel is represented by one value, alternating green values with red and blue values. Select a portion of the image matching the desired test size. These offsets select the face of the person in the image.

```
frmActivePixels = 256;
frmActiveLines = 192;
frmOrig = imread('mandi.tif');
frmInput = frmOrig(900:899+frmActiveLines,2350:2349+frmActivePixels);
figure
imshow(frmInput)
title 'Input Image'
```


Input Image

Create a serializer object and specify the size of the inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10, ...
      'TotalVideoLines',frmActiveLines+10, ...
```

```
        'StartingActiveLine',6, ...
        'FrontPorch',5);
```

Create an interpolator object. Specify the sequence of color values matching the 2-by-2 pixels in the top-left corner of the image.

```
BayerInterpolator = visionhdl.DemosaicInterpolator(...
    'SensorAlignment','RGGB');
```

Serialize the test image. `pixIn` is a vector of pixel values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Set up variables, and then generate the RGB triplet for each pixel in the stream. This example prints a progress message every 32 lines.

```
[pixels,lines,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
pixOut = zeros(numPixelsPerFrame,3,'uint8');
lineCount = 1;
for p = 1:numPixelsPerFrame
    if ctrlIn(p).hEnd
        lineCount = lineCount+1;
        if mod(lineCount,32)==0
            fprintf('Processing line #%d\n',lineCount)
        end
    end
    [pixOut(p,:),ctrlOut(p)] = BayerInterpolator(pixIn(p),ctrlIn(p));
end
```

```
Processing line #32
Processing line #64
Processing line #96
Processing line #128
Processing line #160
Processing line #192
```

Create a deserializer object with a format that matches that of the serializer. Convert the pixel stream to an image frame, and then display the result.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',3, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput)
    title 'Output Image'
end
```

Output Image

Algorithms

This object implements the algorithms described on the Demosaic Interpolator block reference page.

Version History
Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The object now supports multipixel streams that have 2 pixels per clock cycle.

R2021b: Multipixel streaming

The object now supports multipixel streams. The object accepts an input vector of *NumPixels*-by-1 values and returns an output matrix of *NumPixels*-by-3 values, where *NumPixels* is 4 or 8. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
Demosaic Interpolator

Functions
demosaic

Objects
visionhdl.FrameToPixels

visionhdl.Dilation

Package: visionhdl

Morphologically dilate binary pixel stream

Description

The `visionhdl.Dilation` System object morphologically dilates a binary pixel stream. This operation replaces each pixel with the local maximum of the neighborhood around the pixel. The object operates on a stream of binary pixel intensity values.

To morphologically dilate a binary pixel stream:

1   Create the `visionhdl.Dilation` 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

```
dilator = visionhdl.Dilation
dilator = visionhdl.Dilation(Name,Value)
```

Description

`dilator = visionhdl.Dilation` creates a System object that morphologically dilates a binary pixel stream.

`dilator = visionhdl.Dilation(Name,Value)` sets properties using one or more name-value arguments. For example, `'Neighborhood',getnhood(strel('disk',4))` specifies a 4-by-4 disk-pattern neighborhood.

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.

Neighborhood — Pixel neighborhood
[0,1,0;1,1,1;0,1,0] (default) | vector or matrix of binary values

Pixel neighborhood, specified as a vector or matrix of binary values.

The object supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the `Neighborhood` property to `getnhood(strel(shape))`, where *shape* is specified by the input arguments to the `strel` function.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (*n* – 1)-by-`LineBufferSize` memory locations to store the pixels, where *n* is the number of lines in the `Neighborhood` property value.

PaddingMethod — Method for padding boundary of input image
'Constant' (default) | 'None'

Method for padding the boundary of the input image, specified as one of these values.

- `'Constant'` — The object pads the image with zeros. These values prevent dilation at the boundaries of the active frame.
- `'None'` — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the *n*/2 pixels around the edge of the frame for downstream operations. *n* is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

For more information about these methods, see "Edge Padding".

Usage

Syntax

```
[pixelout,ctrlout] = dilator(pixelin,ctrlin)
```

Description

`[pixelout,ctrlout] = dilator(pixelin,ctrlin)` returns the next binary pixel value, `pixelout`, resulting from morphologically dilating the neighborhood around each input binary pixel, `pixelin`.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel
`0` or `false` | `1` or `true`

Input pixel, specified as a `logical` value of `0`(`false`) or `1` (`true`).

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Data Types: `logical`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel
`0` | `1`

Output pixel that is transformed by a morphological operation, returned as a `logical` value.

Data Types: `logical`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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)
```
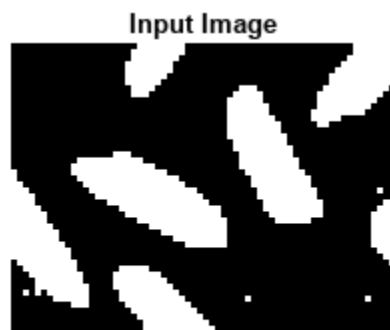
Common to All System Objects
step   Run System object algorithm
release Release resources and allow changes to System object property values and input characteristics
reset   Reset internal states of System object

Examples

Morphologically Dilate an Image

Load a source image from a file. Select a portion of the image that matches the desired test size. This source image contains pixel intensity values of `uint8` data type. Apply a threshold to convert the pixel data to binary values.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
frmInput = frmInput>128;
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Input Image

Create a serializer System object™ and define inactive pixel regions. Set the number of inactive pixels following each active line to at least double the horizontal size of the neighborhood. Set the number of lines following each frame to at least double the vertical size of the neighborhood.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',3, ...
      'FrontPorch',10);
```

Create a morphological dilation System object.

```
 dilator = visionhdl.Dilation(...
      'Neighborhood',getnhood(strel('disk',3)));
```

Serialize the test image by calling the serializer object. `pixin` is a vector of intensity values. `ctrlin` is a vector of control signal structures.

```
[pixin,ctrlin] = frm2pix(frmInput);
```

Prepare to process the pixel stream by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixout = false(numPixelsPerFrame,1);
ctrlout  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlin(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixout(p),ctrlout(p)] = dilator(pixin(p),ctrlin(p));
    if (ctrlout(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
objLatency_cycles = foundValOut - foundValIn
```

```
objLatency_cycles = 184
```

Create a deserializer System object with a format that matches the serializer format. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20);
[frmOutput,frmValid] = pix2frm(pixout,ctrlout);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image
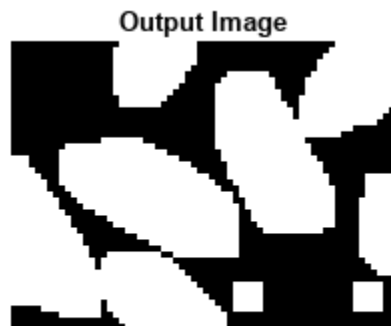
Algorithms

This object implements the algorithms described on the Dilation block reference page.

Version History

Introduced in R2015a

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

R2020a: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
Dilation

Functions
imdilate

Objects
visionhdl.FrameToPixels | visionhdl.Erosion

visionhdl.GrayscaleDilation

Package: visionhdl

Morphologically dilate grayscale pixel stream

Description

The `visionhdl.GrayscaleDilation` System object morphologically dilates a grayscale pixel stream. This operation replaces each pixel with the local maximum of the neighborhood around the pixel. The object operates on a stream of grayscale pixel intensity values.

You can specify a neighborhood, or structuring element, of up to 32-by-32 pixels. For line, square, or rectangle structuring elements that are more than 8 pixels wide, the object uses the Van Herk algorithm to find the maximum. This algorithm uses only three comparators to find the maximums of all the rows and then uses a comparison tree to find the maximum of the row results.

For structuring elements that are less than 8 pixels wide, or that contain zero elements, the object implements a pipelined comparison tree for each row of the neighborhood. An additional comparison tree finds the maximum value of the row results. If the structuring element contains zeros that mask off pixels, the algorithm saves hardware resources by not implementing comparators for those pixel locations.

To morphologically dilate a grayscale pixel stream:

1 Create the `visionhdl.GrayscaleDilation` 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

```
dilator = visionhdl.GrayDilation
dilator = visionhdl.GrayDilation(Name,Value)
```

Description

`dilator = visionhdl.GrayDilation` creates a System object that morphologically dilates a grayscale pixel stream.

`dilator = visionhdl.GrayDilation(Name,Value)` sets properties using one or more name-value arguments. For example, `'Neighborhood',getnhood(strel('disk',4))` specifies a 4-by-4 disk-pattern neighborhood.

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.

Neighborhood — Pixel neighborhood
ones(5,5) (default) | vector or matrix of binary values

Pixel neighborhood, specified as a vector or matrix of binary values.

The object supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the Neighborhood property to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function. The minimum neighborhood size is a 2-by-2 matrix, or a 2-element column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (n – 1)-by-LineBufferSize memory locations to store the pixels, where n is the number of lines in the Neighborhood property value.

Usage

Syntax

[pixelout,ctrlout] = dilator(pixelin,ctrlin)

Description

[pixelout,ctrlout] = dilator(pixelin,ctrlin) returns the next grayscale pixel value, pixelout, resulting from morphologically dilating the neighborhood around each input grayscale pixel, pixelin.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the visionhdl.FrameToPixels object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel
unsigned integer

Input pixel, specified as an unsigned integer.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt(0,N,0)`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel
unsigned integer

Output pixel that is transformed by a morphological operation, returned as an unsigned integer.

The data type is the same as the data type of `pixelin`.

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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
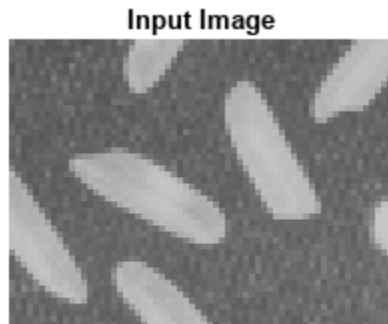
Examples

Morphologically Dilate Grayscale Image

Load a source image from a file. Select a portion of the image that matches the desired test size.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
```

```
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```

Input Image

Create a serializer System object™ and define the inactive pixel regions. Set the number of inactive pixels following each active line to at least double the horizontal size of the neighborhood. Set the number of lines following each frame to at least double the vertical size of the neighborhood.

```
frm2pix = visionhdl.FrameToPixels(...
        'NumComponents',1, ...
        'VideoFormat','custom', ...
        'ActivePixelsPerLine',frmActivePixels, ...
        'ActiveVideoLines',frmActiveLines, ...
        'TotalPixelsPerLine',frmActivePixels+20, ...
        'TotalVideoLines',frmActiveLines+20, ...
        'StartingActiveLine',3, ...
        'FrontPorch',10);
```

Create a morphological dilation System object.

```
dilator = visionhdl.GrayscaleDilation(...
            'Neighborhood',ones(4,4));
```

Serialize the test image by calling the serializer object. `pixin` is a vector of intensity values. `ctrlin` is a vector of control signal structures.

```
[pixin,ctrlin] = frm2pix(frmInput);
```

Prepare to process the pixel stream by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixout = uint8(zeros(numPixelsPerFrame,1));
ctrlout = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
```

```
    if (ctrlin(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixout(p),ctrlout(p)] = dilator(pixin(p),ctrlin(p));
    if (ctrlout(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
objLatency_cycles = foundValOut - foundValIn
```

```
objLatency_cycles = 104
```

Create a deserializer System object with a format that matches the serializer format. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20);
[frmOutput,frmValid] = pix2frm(pixout,ctrlout);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image

Algorithms

This object implements the algorithms described on the Grayscale Dilation block reference page.

Version History
Introduced in R2016a

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Objects
visionhdl.GrayscaleErosion | visionhdl.FrameToPixels

Blocks
Grayscale Dilation

Functions
imdilate

visionhdl.EdgeDetector

Package: visionhdl

Find edges of objects

Description

The `visionhdl.EdgeDetector` System object finds the edges in a grayscale pixel stream using the Sobel, Prewitt, or Roberts method. The object convolves the input pixels with derivative approximation matrices to find the gradient of pixel magnitude along two orthogonal directions. It then compares the sum of the squares of the gradients to a configurable threshold to determine if the gradients represent an edge. The Sobel and Prewitt methods calculate the gradient in horizontal and vertical directions. The Roberts method calculates the gradients at 45 and 135 degrees.

The object returns a binary image as a stream of pixel values. A pixel value of 1 indicates that the pixel is an edge. You can optionally enable output of the gradient values in the two orthogonal directions at each pixel.

To find the edges of objects:

1   Create the `visionhdl.EdgeDetector` 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

```
detectEdges = visionhdl.EdgeDetector
detectEdges = visionhdl.EdgeDetector(Name,Value)
```

Description

`detectEdges = visionhdl.EdgeDetector` returns a System object that detects edges using the Sobel method.

`detectEdges = visionhdl.EdgeDetector(Name,Value)` returns an edge detector System object with properties specified 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.

Method — Edge detection algorithm
'Sobel' (default) | 'Prewitt' | 'Roberts'

When you select 'Sobel' or 'Prewitt', the object calculates horizontal and vertical gradients, Gv and Gh. When you select 'Roberts', the object calculates orthogonal gradients, G45 and G135. For details of each method, see "Algorithms" on page 1-58.

Note When you select 'Prewitt', the full-precision internal data type is large due to the 1/6 coefficient. Consider setting the GradientComponentOutputPorts property to true so that you can customize the data type to a smaller size.

BinaryImageOutputPort — Enable edge output argument
true (default) | false

When this property is true, the object returns a binary pixel value representing whether an edge was detected at each location in the frame.

You must set at least one of BinaryImageOutputPort or GradientComponentOutputPorts to true. These properties specify the output arguments that return the edge results. If both properties are false, the object cannot return the results.

GradientComponentOutputPorts — Enable gradient output arguments
false (default) | true

When this property is true, the object returns the G1 and G2 arguments representing the gradients calculated in two orthogonal directions at each pixel. When you set the Method property to 'Sobel' or 'Prewitt', the outputs are horizontal and vertical gradients, Gv and Gh. When you set the Method property to 'Roberts', the outputs are diagonal orthogonal gradients, G45 and G135. Set the data type for these arguments in the GradientDataType property.

You must set at least one of BinaryImageOutputPort and GradientComponentOutputPorts to true. These properties specify the output arguments that return the edge results. If both properties are false, the object cannot return the results.

ThresholdSource — Source for gradient threshold that indicates an edge
'Property' (default) | 'Input port'

Set this property to 'Input port' to set the threshold as an input argument to the object. When this property is set to 'Property', set the threshold in the Threshold property.

Threshold — Gradient threshold value that indicates an edge
20 (default) | scalar

The object compares the square of this value to the sum of the squares of the gradients. The object casts this value to the data type of the gradients. This property applies when you set ThresholdSource to 'Property'.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: single | double | int | uint | fi

LineBufferSize — Size of line memory buffer
2048 (default) | integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (N – 1)-by-`LineBufferSize` memory locations to store the pixels, where N is the number of lines in the differential approximation matrix. If you set the `Method` property to `'Sobel'` or `'Prewitt'`, then N is 3. If you set the `Method` property to `'Roberts'`, then N is 2.

PaddingMethod — Method for padding
`'Symmetric'` (default) | `'None'`

Select one of these methods for padding the boundary of the input image. For more information about these methods, see "Edge Padding".

- `'Symmetric'` — Set the value of the padding pixels to mirror the edge of the image. This option prevents edges from being detected at the boundaries of the active frame.

- `'None'` — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the $n/2$ pixels around the edge of the frame for downstream operations. n is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

RoundingMethod — Rounding mode for fixed-point operations
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is `single` or `double`.

OverflowAction — Overflow mode for fixed-point operations
`'Wrap'` (default) | `'Saturate'`

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 data type is `single` or `double`.

GradientDataType — Method to determine data type for gradient output arguments
`'Full precision'` (default) | `'custom'`

- `'Full precision'` (default) — Use full-precision rules based on the data type of the `pixelIn` argument, and the coefficients of the derivative approximation matrices.

- `'custom'` — Use the data type defined in the `CustomGradientDataType` property.

CustomGradientDataType — Data type for gradient outputs
numerictype(1,8,0) (default)

Data type for the gradient output values, specified as `numerictype(signed,WL,FL)`, where *WL* is the word length and *FL* is the fraction length in bits.

Usage

Syntax

```
[edge,ctrlout] = detectEdges(pixelin,ctrlin)
[G1,G2,ctrlout] = detectEdges(pixelin,ctrlin)
[edge,ctrlout] = detectEdges(pixelin,ctrlin,thresh)
```

Description

`[edge,ctrlout] = detectEdges(pixelin,ctrlin)` detects edges in the neighborhood of `pixelin` by computing the gradient in two orthogonal directions. The `edge` output argument is a binary value indicating whether the sum of the squares of the gradients for the input pixel is above the threshold indicating an edge.

`[G1,G2,ctrlout] = detectEdges(pixelin,ctrlin)` detects edges in the neighborhood of `pixelIn` by computing the gradient in two orthogonal directions. Use this syntax when you set the `GradientComponentOutputPorts` property to `true`. The `G1` and `G2` output arguments are the gradients calculated in the two orthogonal directions. When you set the `Method` property to `'Sobel'` or `'Prewitt'`, the first argument is the vertical gradient, and the second argument is the horizontal gradient. When you set the `Method` property to `'Roberts'`, the first argument is the 45-degree gradient, and the second argument is the 135-degree gradient.

`[edge,ctrlout] = detectEdges(pixelin,ctrlin,thresh)` detects edges in the neighborhood of `pixelin` by computing the gradient in two orthogonal directions. Use this syntax when you set the `ThresholdSource` property to `'InputPort'`. The `edge` output argument is a binary value indicating whether the sum of the squares of the gradients is above the threshold, `thresh`, squared.

You can use any combination of the optional argument syntaxes.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel
scalar

Intensity of a single pixel, specified as a scalar value.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Data Types: `int` | `uint` | `fi` | `single` | `double`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

`thresh` — Threshold
scalar

Gradient threshold value that indicates an edge, specified as a scalar numeric value.

The object compares this value squared to the sum of the squares of the gradients.

Dependencies

To enable this argument, set `ThresholdSource` property to `'InputPort'`.

Data Types: `int` | `uint` | `fi` | `single` | `double`

Output Arguments

`edge` — Edge locations
scalar

Pixel value indicating an edge at this pixel, returned as a scalar binary value.

`G1` — Gradient in first direction
scalar

Gradient calculated in the first direction, returned as a scalar value.

If you set the `Method` property to `'Sobel'` or `'Prewitt'`, this argument is the vertical gradient. When you set the `Method` property to `'Roberts'`, this argument is the 45-degree gradient.

Configure the data type of the gradients by using the `GradientComponentDataType` and `CustomGradientComponent` properties.

Dependencies

To enable this argument, set the `GradientComponentOutputPorts` property to `true`

`G2` — Gradient in second direction
scalar

Gradient calculated in the second direction, returned as a scalar value.

If you set the `Method` property to `'Sobel'` or `'Prewitt'`, this argument is the horizontal gradient. When you set the `Method` property to `'Roberts'`, this argument is the 135-degree gradient.

Configure the data type of the gradients by using the `GradientComponentDataType` and `CustomGradientComponent` properties.

Dependencies

To enable this argument, set the `GradientComponentOutputPorts` property to `true`.

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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
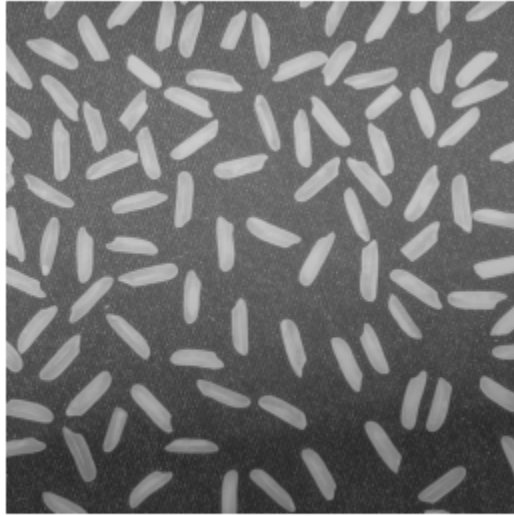reset   Reset internal states of System object

Examples

Edge Detection Using Sobel Method

Detect edges in a thumbnail image by using the Sobel method.

Import a test image.

```
frmInput = imread('rice.png');
[frmActivePixels,frmActiveLines] = size(frmInput);
figure
imshow(frmInput)
title 'Input Image'
```

Input Image

Create a serializer and specify the size of inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',6, ...
      'FrontPorch',5);
```

Create an edge detection object with the default property values. The default detection method is Sobel.

```
edgeDetectSobel = visionhdl.EdgeDetector();
```

Speed up simulation by enabling a simulation mode that uses code generation for the System object.

```
edgeDetectSobel.simulateUsing('Code generation');
```

Serialize the test image by using the object you created. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures. Preallocate vectors for the output signals.

```
[pixIn,ctrlIn] = frm2pix(frmInput);

[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
edgeOut = false(numPixelsPerFrame,1);
```

For each pixel in the stream, compute whether it represents an edge.

```
for p = 1:numPixelsPerFrame
    [edgeOut(p),ctrlOut(p)] = edgeDetectSobel(pixIn(p),ctrlIn(p));
end
```

Create a deserializer with a format matching that of the serializer. Use the deserializer to convert the output pixel stream to an image frame.

```
pix2frm = visionhdl.PixelsToFrame( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10);

[frmOutput,frmValid] = pix2frm(edgeOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput)
    title 'Output Image'
end
```

Output Image
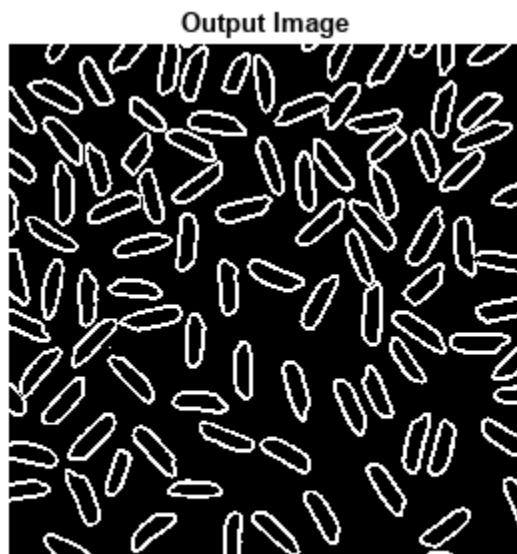
Algorithms

This object implements the algorithms described on the Edge Detector block reference page.

Version History

Introduced in R2015a

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

R2019b: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your design. A best practice is to synchronize parallel paths in your design by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Edge Detector | `edge` | `visionhdl.FrameToPixels`

Topics
"Enhanced Edge Detection from Noisy Color Video"

visionhdl.Erosion

Package: visionhdl

Morphologically erode binary pixel stream

Description

The `visionhdl.Erosion` System object morphologically erodes a binary pixel stream. This operation replaces each pixel with the local minimum of the neighborhood around the pixel. The object operates on a stream of binary intensity values.

To morphologically erode a binary pixel stream:

1　Create the `visionhdl.Erosion` 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

```
eroder = visionhdl.Erosion
eroder = visionhdl.Erosion(Name,Value)
```

Description

`eroder = visionhdl.Erosion` creates a System object that morphologically erodes a binary pixel stream.

`eroder = visionhdl.Erosion(Name,Value)` sets properties using one or more name-value arguments. For example, `'Neighborhood',getnhood(strel('disk',4))` specifies a 4-by-4 disk-pattern neighborhood.

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.

Neighborhood — Pixel neighborhood
`ones(3,3)` (default) | vector or matrix of binary values

Pixel neighborhood, specified as a vector or matrix of binary values.

The object supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the `Neighborhood` property to `getnhood(strel(`*shape*`))`, where *shape* is specified by the input arguments to the `strel` function.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (*n* – 1)-by-`LineBufferSize` memory locations to store the pixels, where *n* is the number of lines in the `Neighborhood` property value.

PaddingMethod — Method for padding boundary of input image
'Constant' (default) | 'None'

Method for padding the boundary of the input image, specified as one of these values.

- **'Constant'** — The object pads the image with ones. These values prevent erosion at the boundaries of the active frame.

- **'None'** — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the *n*/2 pixels around the edge of the frame for downstream operations. *n* is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

For more information about these methods, see "Edge Padding".

Usage

Syntax

`[pixelout,ctrlout] = eroder(pixelin,ctrlin)`

Description

`[pixelout,ctrlout] = eroder(pixelin,ctrlin)` returns the next binary pixel value, `pixelout`, resulting from morphologically eroding the neighborhood around each input binary pixel value, `pixelin`.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel
`0` or `false` | `1` or `true`

Input pixel, specified as a `logical` value of `0`(`false`) or `1` (`true`).

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Data Types: `logical`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel
`0` | `1`

Output pixel that is transformed by a morphological operation, returned as a `logical` value.

Data Types: `logical`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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)
```
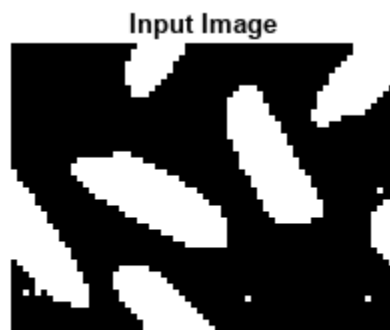
Common to All System Objects

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

Morphologically Erode an Image

Load a source image from a file. Select a portion of the image that matches the desired test size. This source image contains pixel intensity values of `uint8` data type. Apply a threshold to convert to the pixel data to binary values.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
frmInput = frmInput>128;
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Create a serializer System object™ and define inactive pixel regions. Set the number of inactive pixels following each active line to at least double the horizontal size of the neighborhood. Set the number of lines following each frame to at least double the vertical size of the neighborhood.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',3, ...
      'FrontPorch',10);
```

Create a morphological erode object.

```
 eroder = visionhdl.Erosion(...
          'Neighborhood',ones(2,7));
```

Serialize the test image by calling the serializer object. `pixin` is a vector of intensity values. `ctrlin` is a vector of control signal structures.

```
[pixin,ctrlin] = frm2pix(frmInput);
```

Prepare to process the pixel stream by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixout = false(numPixelsPerFrame,1);
ctrlout  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlin(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixout(p),ctrlout(p)] = eroder(pixin(p),ctrlin(p));
    if (ctrlout(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
objLatency_cycles = foundValOut - foundValIn
```

```
objLatency_cycles = 102
```

Create a deserializer System object with a format that matches the serializer format. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20);
[frmOutput,frmValid] = pix2frm(pixout,ctrlout);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image
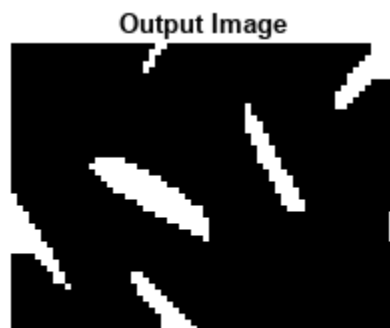
Algorithms

This object implements the algorithms described on the Erosion block reference page.

Version History

Introduced in R2015a

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

R2020a: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
Erosion

Functions
imerode

Objects
visionhdl.Dilation | visionhdl.FrameToPixels

visionhdl.GrayscaleErosion

Package: `visionhdl`

Morphologically erode grayscale pixel stream

Description

The `visionhdl.GrayscaleErosion` System object morphologically erodes a grayscale pixel stream. This operation replaces each pixel with the local minimum of the neighborhood around the pixel. The object operates on a stream of grayscale pixel intensity values.

You can specify a neighborhood, or structuring element, of up to 32-by-32 pixels. For line, square, or rectangle structuring elements that are more than 8 pixels wide, the object uses the Van Herk algorithm to find the maximum. This algorithm uses only three comparators to find the maximums of all the rows, then uses a comparison tree to find the maximum of the row results.

For structuring elements that are less than 8 pixels wide, or that contain zero elements, the object implements a pipelined comparison tree for each row of the neighborhood. An additional comparison tree finds the maximum value of the row results. If the structuring element contains zeros that mask off pixels, the algorithm saves hardware resources by not implementing comparators for those pixel locations.

To morphologically erode a grayscale pixel stream:

1 Create the `visionhdl.GrayscaleErosion` 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

```
eroder = visionhdl.GrayErosion
eroder = visionhdl.GrayErosion(Name,Value)
```

Description

`eroder = visionhdl.GrayErosion` creates a System object that morphologically erodes a grayscale pixel stream.

`eroder = visionhdl.GrayErosion(Name,Value)` sets properties using one or more name-value arguments. For example, `'Neighborhood',getnhood(strel('disk',4))` specifies a 4-by-4 disk-pattern neighborhood.

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.

Neighborhood — Pixel neighborhood
ones(3,3) (default) | vector or matrix of binary values

Pixel neighborhood, specified as a vector or matrix of binary values.

The object supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the Neighborhood property to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function. The minimum neighborhood size is a 2-by-2 matrix, or a 2-element column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (n – 1)-by-LineBufferSize memory locations to store the pixels, where n is the number of lines in the Neighborhood property value.

Usage

Syntax

[pixelout,ctrlout] = eroder(pixelin,ctrlin)

Description

[pixelout,ctrlout] = eroder(pixelin,ctrlin) returns the next pixel intensity value, pixelout, resulting from morphologically eroding the neighborhood around each input pixel intensity value, pixelin.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the visionhdl.FrameToPixels object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel
unsigned integer

Input pixel, specified as an unsigned integer.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `uint64` | `fixdt(0,N,0)`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel
unsigned integer

Output pixel that is transformed by a morphological operation, returned as an unsigned integer.

The data type is the same as the data type of `pixelin`.

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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
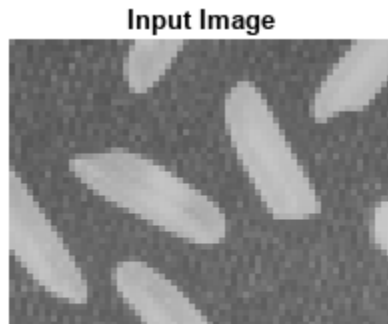
Examples

Morphologically Erode Grayscale Image

Load a source image from a file. Select a portion of the image that matches the desired test size.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
```

```
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```

Input Image

Create a serializer System object™ and define the inactive pixel regions. Set the number of inactive pixels following each active line to at least double the horizontal size of the neighborhood. Set the number of lines following each frame to at least double the vertical size of the neighborhood.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20, ...
      'TotalVideoLines',frmActiveLines+20, ...
      'StartingActiveLine',3, ...
      'FrontPorch',10);
```

Create a morphological erode System object.

```
eroder = visionhdl.GrayscaleErosion(...
          'Neighborhood',ones(2,5));
```

Serialize the test image by calling the serializer object. `pixin` is a vector of intensity values. `ctrlin` is a vector of control signal structures.

```
[pixin,ctrlin] = frm2pix(frmInput);
```

Prepare to process the pixel stream by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixout = uint8(zeros(numPixelsPerFrame,1));
ctrlout = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
```

```
    if (ctrlin(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixout(p),ctrlout(p)] = eroder(pixin(p),ctrlin(p));
    if (ctrlout(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
objLatency_cycles = foundValOut - foundValIn
```

```
objLatency_cycles = 107
```

Create a deserializer System object with a format that matches the serializer format. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
    'NumComponents',1, ...
    'VideoFormat','custom', ...
    'ActivePixelsPerLine',frmActivePixels, ...
    'ActiveVideoLines',frmActiveLines, ...
    'TotalPixelsPerLine',frmActivePixels+20);
[frmOutput,frmValid] = pix2frm(pixout,ctrlout);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image

Algorithms

This object implements the algorithms described on the Grayscale Erosion block reference page.

Version History
Introduced in R2016a

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Objects
visionhdl.GrayscaleDilation | visionhdl.FrameToPixels

Blocks
Grayscale Erosion

Functions
imerode

visionhdl.FrameToPixels

Package: `visionhdl`

Convert frame-based video to pixel stream

Description

The `visionhdl.FrameToPixels` System object converts color or grayscale frame-based video to a pixel stream and control structure. The control structure indicates the validity of each pixel and its location in the frame. The pixel stream format can include padding pixels around the active frame. You can configure the frame and padding dimensions by selecting a common video format or by specifying custom dimensions. For details about the pixel stream format, see "Streaming Pixel Interface".

Use this object to generate input for a function targeted for HDL code generation. This object itself does not support HDL code generation.

If your design converts frames to a pixel stream and later converts the stream back to frames, specify the same video format for the `FrameToPixels` object and the `PixelsToFrame` object.

To convert frame-based video to a pixel stream:

1 Create the `visionhdl.FrameToPixels` 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

```
F2P = visionhdl.FrameToPixels(Name,Value)
```

Description

`F2P = visionhdl.FrameToPixels(Name,Value)` returns a System object that serializes input frames into a pixel stream. Set 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.

NumComponents — Number of values used to represent each pixel
1 (default) | 2 | 3 | 4

Number of values used to represent each pixel, specified as 1, 2, 3, or 4 components.

- For grayscale video, set this property to 1.
- For color video, for example, {R,G,B} or {Y,Cb,Cr}, set this property to 3.
- For color video with an alpha channel for transparency, set this property to 4.

The visionhdl.FrameToPixels object returns a *P*-by-NumComponents matrix, where *P* is the total number of pixels in the padded frame.

Dependencies

When NumComponents is greater than 1, you must set the NumPixels property to 1.

NumPixels — Number of pixels streamed per time step
1 (default) | 2 | 4 | 8

Number of pixels transferred on the streaming interface for each cycle, specified as 1, 2, 4, or 8. To enable multipixel streaming and increase throughput for high-resolution or high-frame-rate video, set this property to 2, 4, or 8. The visionhdl.FrameToPixels object returns a *P*-by-NumPixels matrix, where *P* is the total number of pixels in the padded frame.

Note You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Dependencies

When NumPixels is greater than 1, you must set the NumComponents property to 1.

VideoFormat — Dimensions of active and inactive regions in video frame
'1080p' (default) | '240p' | '480p' | '480pH' | '576p' | '720p' | '768p' | '1024p' | '1200p' | '2KCinema' | '4KUHDTV' | '8KUHDTV' | 'Custom'

Dimensions of active and inactive regions of a video frame. To select a predefined format, specify the VideoFormat property as one of the options in the first column of the table. For a custom format, set VideoFormat to 'Custom', and specify the dimension properties as integers. The frame dimensions are indicated in the diagram.

Video Format	Active Pixels Per Line	Active Video Lines	Total Pixels Per Line	Total Video Lines	Starting Active Line	Ending Active Line	Front Porch	Back Porch
240p	320	240	402	324	1	240	44	38
480p	640	480	800	525	36	515	16	144
480pH	720	480	858	525	33	512	16	122
576p	720	576	864	625	47	622	12	132
720p	1280	720	1650	750	25	744	110	260
768p	1024	768	1344	806	10	777	24	296
1024p	1280	1024	1688	1066	42	1065	48	360
1080p (default)	1920	1080	2200	1125	42	1121	88	192
1200p	1600	1200	2160	1250	50	1249	64	496
2KCinema	2048	1080	2750	1125	42	1121	639	63
4KUHDTV	3840	2160	4400	2250	42	2201	88	472
8KUHDTV	7680	4320	8800	4500	42	4361	88	1032
Custom	User-defined	User-defined	User-defined	User-defined	User-defined	User-defined	User-defined	User-defined

Note When using a custom format, the properties you enter for the active and inactive dimensions of the image must add up to the total frame dimensions.

For the horizontal direction, `TotalPixelsPerLine` must be greater than or equal to `FrontPorch + ActivePixelsPerLine`. The object calculates `BackPorch = TotalPixelsPerLine − FrontPorch − ActivePixelsPerLine`.

For the vertical direction, `TotalVideoLines` must be greater than or equal to `StartingActiveLine + ActiveVideoLines − 1`. The object calculates `EndingActiveLine = StartingActiveLine + ActiveVideoLines − 1`.

If you specify a format that does not conform to these rules, the object reports an error.

Note When using a custom format, `ActivePixelsPerLine` must be greater than 1. Also, set the horizontal blanking interval, or `BackPorch + FrontPorch`, according to these guidelines.

- The total of `BackPorch + FrontPorch` must be at least 2 times the largest kernel size of the algorithm in the objects following the `visionhdl.FrameToPixels` object. If the kernel size is greater than 4, the total porch must be at least 8 pixels.
- The `BackPorch` property value must be at least 6 pixels. This property is the number of inactive pixels before the first valid pixel in a frame.

For more information on blanking intervals, see "Configure Blanking Intervals".

Note When using multipixel streaming (`NumPixels > 1`), these requirements apply.

- The video format must have horizontal dimensions divisible by the `NumPixels` property value. The horizontal dimensions are set by these properties: `ActivePixelsPerLine`, `TotalPixelsPerLine`, `FrontPorch`, and `BackPorch`. Standard video protocols 480p, 768p, 1024p, 1080p, 1200p, 4k UHD, and 8k UHD support `NumPixels` equal to 4 or 8.
- The minimum input frame size for multipixel streaming is 18 rows by 32 columns.
- Choose your kernel size and `ActivePixelsPerLine` such that `ActivePixelsPerLine/NumPixels` is at least the kernel width.

Usage

Syntax

`[pixels,ctrlOut] = F2P(frm)`

Description

`[pixels,ctrlOut] = F2P(frm)` converts the input image matrix, `frm`, to a vector of pixel values, `pixels`, and an associated vector of control structures, `ctrlOut`. The control structure indicates the validity of each pixel and its location in the frame. The output pixels include padding around the active image, specified by the `VideoFormat` property.

For details about the pixel stream format, see "Streaming Pixel Interface".

Input Arguments

`frm` — Input image
matrix

Input image, specified as an `ActiveVideoLines`-by-`ActivePixelsPerLine`-by-`NumComponents` matrix, where:

- `ActiveVideoLines` is the height of the active image.
- `ActivePixelsPerLine` is the width of the active image.
- `NumComponents` is the number of components used to express a single pixel.

Set the size of the active image using the `VideoFormat` property. If the dimensions of `frm` do not match the dimensions specified by `VideoFormat`, the object returns a warning.

Data Types: `uint` | `int` | `logical` | `fi` | `double` | `single`

Output Arguments

`pixels` — Pixel values
matrix

Pixel values, returned as a *P*-by-`NumComponents` matrix or *P*-by-`NumPixels` matrix, where:

- *P* is the total number of pixels in the padded image, which is `TotalPixelsPerLine` × `TotalVideoLines`.
- `NumComponents` is the number of components used to express a single pixel.
- `NumPixels` is the number of pixels transferred on the streaming interface per cycle. When `NumPixels` is greater than 1, you must set `NumComponents` to 1.

> **Note** You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Set the size of the padded image using the `VideoFormat` property. The data type of the pixel values is the same as `im`.

`ctrlOut` — Pixel stream control signals
vector of structures

Control structures associated with the output pixels, returned as a *P*-by-1 vector. *P* is the total number of pixels in the padded image, which is `TotalPixelsPerLine` × `TotalVideoLines`. Each structure contains five control signals indicating the validity of the pixel and its location in the frame. For multipixel streaming, the control signals apply to each set of `NumPixels` values. See "Pixel Control Structure".

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
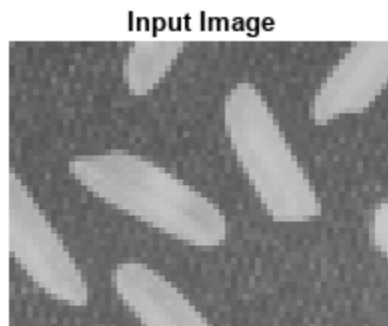reset   Reset internal states of System object

Examples

Convert Between Full-Frame and Pixel-Streaming Data

This example converts a custom-size grayscale image to a pixel stream. It uses the
`visionhdl.LookupTable` object to obtain the negative image. Then it converts the pixel stream
back to a full-frame image.

Load the source image from a file. Select a portion of the image matching the desired test size.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Create a serializer object and specify the size of the inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',6, ...
      'FrontPorch',5);
```

Create a lookup table (LUT) object to generate the negative of the input image.

```
tabledata = linspace(255,0,256);
inverter = visionhdl.LookupTable(tabledata);
```

Serialize the test image by calling the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the stream, look up the negative of the pixel value.

```
for p = 1:numPixelsPerFrame
    [pixOut(p),ctrlOut(p)] = inverter(pixIn(p),ctrlIn(p));
end
```

Create a deserializer object with a format that matches the format of the serializer. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image

Version History

Introduced in R2015a

R2022a: Change in default frame dimensions
Behavior changed in R2022a

The default values of the custom format have changed. In previous releases, the default dimensions matched the 1080p format. Starting in this release, the default values match the default values of the Frame To Pixels and Pixels To Frame blocks, which define a small custom frame as shown in this code.

```
a = visionhdl.FrameToPixels('VideoFormat','Custom')

a =

  visionhdl.FrameToPixels with properties:

           NumComponents: 1
               NumPixels: 1
             VideoFormat: 'Custom'
      ActivePixelsPerLine: 32
         ActiveVideoLines: 18
       TotalPixelsPerLine: 42
          TotalVideoLines: 28
        StartingActiveLine: 6
               FrontPorch: 5
```

This change means you might need to specify additional dimensions for custom formats where previously the default value resulted in a valid format.

The object now updates frame dimension properties when you select a standard format. This change means you can refer to the object properties to determine the dimensions being used by the format.

```
a = visionhdl.FrameToPixels('VideoFormat','480p')

a =

  visionhdl.FrameToPixels with properties:

    NumComponents: 1
        NumPixels: 1
      VideoFormat: '480p'
```

```
TPPL480 = a.TotalPixelsPerLine

ans =

   800
```

R2022a: Two pixels-per-clock streaming

The object can now generate multipixel streams that have 2 pixels per clock cycle.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2020a: Multipixel-multicomponent streaming

The object can now generate multipixel-multicomponent streams.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2019b: Multipixel streaming

The object can now generate multipixel streams. These streams have 4 or 8 pixels per clock cycle. You can set the `NumPixels` property to 4 or 8. With this setting, the output of the object is a vector of 4 or 8 pixel values on each time step.

Video formats for multipixel streams must have horizontal dimensions divisible by the `NumPixels` property value. These horizontal dimensions are set by the following properties: `ActivePixelsPerLine`, `TotalPixelsPerLine`, `FrontPorch`, and `BackPorch`. Standard video protocols 480p, 720p, 1080p, and 4k UHD support either 4 or 8 pixels at a time.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

See Also
`visionhdl.PixelsToFrame` | Frame To Pixels

Topics
"Pixel-Streaming Design in MATLAB"
"Streaming Pixel Interface"

visionhdl.GammaCorrector

Package: `visionhdl`

Apply or remove gamma correction

Description

`visionhdl.GammaCorrector` applies or removes gamma correction on a stream of pixels. Gamma correction adjusts linear pixel values so that the modified values fit a curve. The de-gamma operation performs the opposite operation to obtain linear pixel values.

To apply or remove gamma correction:

1   Create the `visionhdl.GammaCorrector` 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

```
corrector = visionhdl.GammaCorrector
corrector = visionhdl.GammaCorrector(Name,Value)
corrector = visionhdl.GammaCorrector(operation,gammaValue,Name,Value)
```

Description

`corrector = visionhdl.GammaCorrector` returns a System object that applies default gamma correction on a stream of pixels.

`corrector = visionhdl.GammaCorrector(Name,Value)` returns a gamma correction System object, `corrector`, with properties set using one or more name-value pairs. Enclose each property name in single quotes.

Example: `corrector = visionhdl.GammaCorrector('LinearSegment',false)` creates a gamma correction object that does not use a linear segment in the gamma curve.

`corrector = visionhdl.GammaCorrector(operation,gammaValue,Name,Value)` returns a gamma correction System object with the Correction property set to `operation`, the Gamma property set to `gammaValue`, and additional options specified by one or more name-value pairs.

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.

`Correction` — Direction of intensity curve adjustment
`'Gamma'` (default) | `'De-gamma'`

Direction of intensity curve adjustment, specified as either:

• `'Gamma'` — Apply gamma correction.

• `'De-gamma'` — Remove gamma correction.

Gamma — Target or current gamma value
2.2 (default) | `scalar greater than or equal to 1`

Target or current gamma value, specified as a scalar value greater than or equal to 1.

• When you set Correction to `'Gamma'`, set this property to the target gamma value of the output video stream.

• When you set Correction to `'De-gamma'`, set this property to the gamma value of the input video stream.

`LinearSegment` — Option to include a linear segment in the gamma curve
true (default) | false

Option to include a linear segment in the gamma curve, specified as `true` or `false`. When you set this property to `true`, the gamma curve has a linear portion near the origin.

`BreakPoint` — Point where the gamma curve and linear segment meet
0.018 (default) | scalar pixel value between 0 and 1, exclusive

Point where the gamma curve and linear segment meet, specified as a scalar pixel value between 0 and 1, exclusive.

Dependency

To enable this property, set LinearSegment to `true`.

Usage

Syntax

`[pixelout,ctrlout] = corrector(pixelin,ctrlin)`

Description

`[pixelout,ctrlout] = corrector(pixelin,ctrlin)` returns the intensity value of a pixel after gamma correction, and the control signals associated with the pixel. The input, `pixelin`, and output, `pixelout`, are scalar values representing a single pixel.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the

`visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel stream
scalar

Input pixel stream, specified as a scalar intensity value. Integer and fixed-point data types larger than 16 bits are not supported.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `int8` | `int16` | `fi` | `single` | `double`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel stream
scalar

Gamma-corrected intensity of a single pixel, specified as a scalar value. The data type of the output pixel is the same as the data type of `pixelin`.

Data Types: `uint8` | `uint16` | `int8` | `int16` | `fi` | `single` | `double`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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

Gamma Correction

This example performs gamma correction on a thumbnail image.

Load the source image from a file. Select a portion of the image matching the desired test size.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Input Image
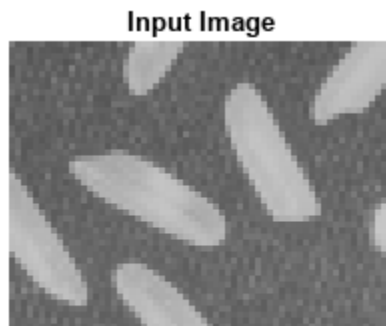
Create a serializer object and specify the size of inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+10,...
      'TotalVideoLines',frmActiveLines+10,...
      'StartingActiveLine',6,...
      'FrontPorch',5);
```

Create a gamma corrector object.

```
 gammacorr = visionhdl.GammaCorrector(...
      'Gamma', 1.75);
```

Serialize the test image by calling the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the stream, compute the gamma corrected pixel value.

```
for p = 1:numPixelsPerFrame
    [pixOut(p),ctrlOut(p)] = gammacorr(pixIn(p),ctrlIn(p));
end
```

Create a deserializer object with a format matching that of the serializer. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'TotalPixelsPerLine',frmActivePixels+10,...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image

Algorithms

For the equations used for gamma correction, see Gamma Correction.

To save hardware resources, the object implements the gamma correction equation as a lookup table. The lookup table maps each input pixel value to a corrected output value.

Latency

The latency of the visionhdl.GammaCorrector object is 2 cycles.

Version History
Introduced in R2015a

R2022b: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 2, 4, or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
Gamma Corrector

Objects
vision.GammaCorrector | visionhdl.FrameToPixels

Functions
imadjust

Topics
"Pixel-Streaming Design in MATLAB"

visionhdl.Histogram

Package: visionhdl

Frequency distribution

Description

The `visionhdl.Histogram` System object computes the frequency distribution of pixel values in a video stream. You can configure the number and size of the bins. The object keeps a running histogram until you clear the bin values and provides a read interface for accessing each bin.

To compute the frequency distribution of pixel values in a video stream:

1   Create the `visionhdl.Histogram` 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

```
histo = visionhdl.Histogram
histo = visionhdl.Histogram(Name,Value)
```

Description

`histo = visionhdl.Histogram` returns a System object, `histo`, that computes image histograms over 256 bins. Each bin value is 16 bits wide.

`histo = visionhdl.Histogram(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.

NumBins — Number of histogram bins
`'256'` (default) | `'32'` | `'64'` | `'128'` | `'512'` | `'1024'` | `'2048'` | `'4096'`

Number of histogram bins, specified as a string or character vector representing a power of two from `'32'` to `'4096'`. Choose the number of bins depending on the input word length (*WL*). If the number

of bins is less than 2^{WL}, the object truncates the least-significant bits of each pixel. If the number of bins is greater than 2^{WL}, some bins are impossible to hit, and when you synthesize your design it will use more hardware resources than necessary.

Data Types: char | string

OutputDataType — Data type of histogram values
'Unsigned fixed point' (default) | 'double' | 'single'

Data type of histogram values, specified as 'Unsigned fixed point', 'double', or 'single'.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: char | string

OutputWordLength — Histogram bin word length
16 (default) | positive integer

Histogram bin word length, specified as a positive integer. If a bin overflows, the count saturates and the object shows a warning.

Dependencies

To enable this property, set the OutputDataType property to 'Unsigned fixed point'.

Usage

Syntax

```
histo(~,~,~,~)
[dataout,readrdy,validout] = histo(pixelin,ctrlin,~,0)
[dataout,readrdy,validout] = histo(~,~,binaddr,0)
[dataout,readrdy,validout] = histo(~,~,binaddr,binreset)
```

Description

histo(~,~,~,~) performs an initial reset phase before processing input data. After object creation or reset, call the object with dummy arguments for NumBins cycles before applying data. You do not have to set the binreset argument to 1 (true) during this phase.

[dataout,readrdy,validout] = histo(pixelin,ctrlin,~,0) adds the input pixel, pixelin, to the internal histogram when the input control signals, ctrl indicate that the pixel is valid. Call the object with this syntax for each pixel in a frame. The object returns readrdy set to 1 (true) when the histogram for the frame is complete. For this syntax, the object returns dataout set to 0 and validout set to 0 (false).

This object uses a streaming pixel interface with a structure for frame control signals. The interface enables the object to operate independently of image size and format and connect with other Vision HDL Toolbox objects. The object accepts pixel data as integer, fixed-point, or floating-point data types. The object accepts control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the visionhdl.FrameToPixels object. For a full description of the interface, see "Streaming Pixel Interface".

[dataout,readrdy,validout] = histo(~,~,binaddr,0) reads the histogram bin specified by binaddr. Use this syntax after the object returns readrdy set to 1 (true). Call the object with this syntax for each histogram bin. After two further calls to the object (input arguments can change), the object returns dataout set to the bin value at binaddr and validout set to 1 (true).

[dataout,readrdy,validout] = histo(~,~,binaddr,binreset) resets the histogram values when binreset is 1(true). You can initiate the reset while also specifying a binaddr to read. Before applying more video data, complete the reset sequence by calling the object with dummy arguments for NumBins cycles.

To visualize the sequence of operations, see the timing diagrams in the "Algorithms" on page 1-115 section of the Histogram block page.

Input Arguments

pixelin — Input pixel stream
unsigned scalar

Single image pixel, specified as an unsigned scalar.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: uint | fi(0,W,0) | logical | double | single

ctrlin — Control signals accompanying input pixel stream
pixelcontrol structure

Control signals accompanying the input pixel stream, specified as a pixelcontrol structure containing five logical data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: struct

binaddr — Bin number for reading histogram values
nonnegative integer

Bin number for reading histogram values, specified as a nonnegative integer. The object expects this input after it has returned readrdy set to 1 (true). The data type must be fixdt(0,$\log_2$(NumBins),0).

Data Types: fixdt(0,N,0)

binreset — Reset histogram bin values
true or 1 | false or 0

Reset histogram bin values, specified as 1 (true) or 0 (false). A binreset value of 1 (true) triggers a RAM initialization sequence that resets the histogram bin values. It takes NumBins calls to the object to clear all locations and ignores input arguments during this interval.

Data Types: logical

Output Arguments

`readrdy` — Indicates histogram bins available for read
true or 1 | false or 0

Indication that histogram bins are available for read, returned as 1 (true) or 0 (false). When the object sets readrdy to 1 (true), the histogram bins are ready to read. The object returns readrdy as 1 (true) two cycles after the final pixel of a frame.

Data Types: logical

`dataout` — Bin value for requested address
nonnegative integer

Histogram bin value that corresponds to the requested address, binaddr, returned as a nonnegative integer. The OutputDataType property specifies the data type for this value.

`validout` — Indicates that histogram value is valid
true or 1 | false or 0

Indication that histogram value is valid, returned as 1 (true) or 0 (false). When the object returns validout as 1 (true), the histogram bin value, dataout, is 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 Histogram of Image

Set the dimensions of the test image, and load a source image. Select a portion of the image matching the desired test size.

```
frmActivePixels = 64;
frmActiveLines = 48;
frmOrig = imread('rice.png');
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```

Input Image

Create a serializer System object™ and define inactive pixel regions. Then, create a histogram System object. The default setting is 256 bins.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+10,...
      'TotalVideoLines',frmActiveLines+10,...
      'StartingActiveLine',6,...
      'FrontPorch',5);

histo = visionhdl.Histogram();
bins = str2double(histo.NumBins);
```

Serialize the test image. `pixelIn` is a vector of intensity values and `ctrlIn` is a vector of control signal structures. Initialize output signals for the histogram results.

```
[pixelIn,ctrlIn] = frm2pix(frmInput);

[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
readRdy = false(numPixelsPerFrame,1);
dataOut = zeros(bins-1,1,'uint8');
validOut  = false(bins-1,1);
noOpCtrl = pixelcontrolstruct(0,0,0,0,0);
noAddr = uint8(0);
noReset = false;
```

Call the object with dummy input to initialize the bin memory.

```
for p = 1:bins
    histo(uint8(0),noOpCtrl,noAddr,noReset);
end
```

For each pixel in the padded frame, sort the pixel into a bin. The object returns `readRdy` as 1 (`true`) two cycles after the active frame is complete.

```
for p = 1:numPixelsPerFrame
    [~,readRdy(p),~] = histo(pixelIn(p),ctrlIn(p),noAddr,noReset);
end
```

Once the frame is complete, as indicated by readRdy, read the bin values. The bin addresses are 0:bins-1. The object returns each bin value after two cycles of latency, so call the object bins+2 times.

```
if readRdy(numPixelsPerFrame)
  for p = 1:bins+2
    if (p < bins)
       % Read a normal bin
       [dataOut(p),~,validOut(p)] = histo(uint8(0),noOpCtrl,uint8(p-1),noReset);
    elseif (p == bins)
       % Read the final bin value and initiate binReset
       [dataOut(p),~,validOut(p)] = histo(uint8(0),noOpCtrl,uint8(p-1),true);
    elseif (p > bins)
       % Flush final bin values with two more calls
       [dataOut(p),~,validOut(p)] = histo(uint8(0),noOpCtrl,noAddr,noReset);
    end
  end
end
```
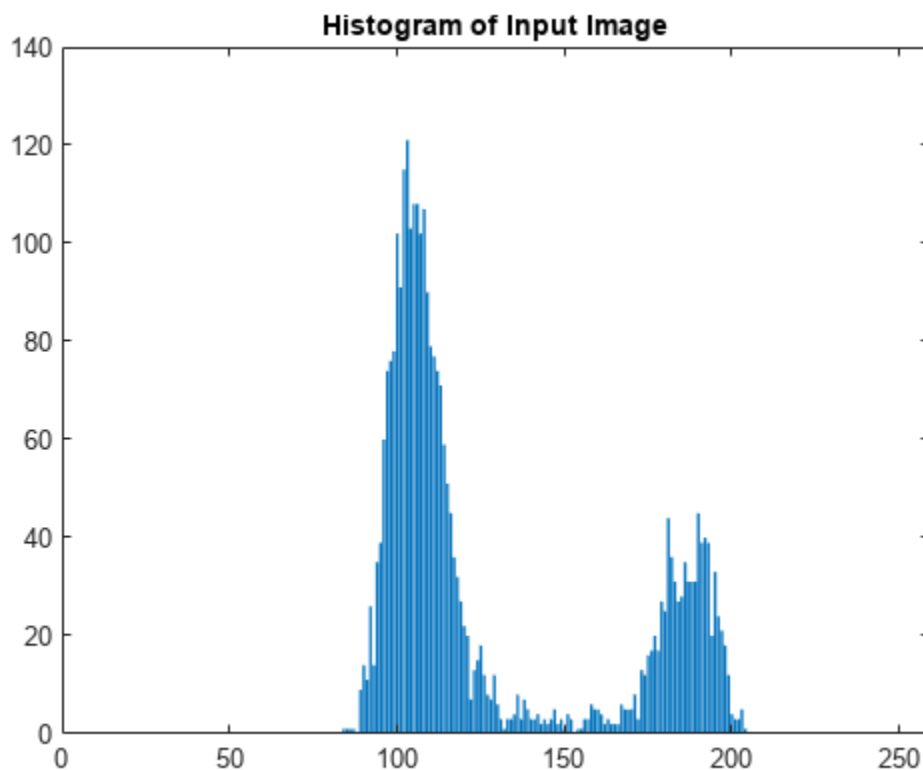
Graph the bin values.

```
dataOut = dataOut(validOut==1);
figure
bar(dataOut)
title('Histogram of Input Image')
```


Call the object with dummy input to clear the bin memory.

```
for p = 1:bins
    histo(uint8(0),noOpCtrl,noAddr,noReset);
end
```

Algorithms

This object implements the algorithms described in the Algorithms section of the Histogram block page.

Version History
Introduced in R2015a

R2022b: Two pixels-per-clock streaming

The object now supports multipixel streams that have 2 pixels per clock cycle.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2022a: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input column vectors of 4, or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector. The output interface is the same whether the input is scalar or vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2019b: Increased histogram sizes

You can now configure the object to have 2048 or 4096 histogram bins.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also
Histogram | `imhist` | `visionhdl.FrameToPixels`
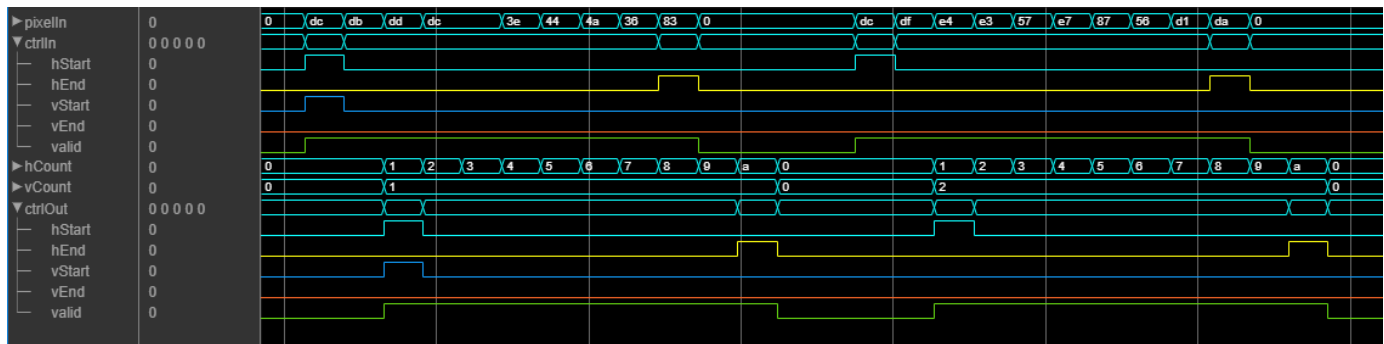
visionhdl.HVCounter

Package: visionhdl

Counts active pixel dimensions of streaming video

Description

The `visionhdl.HVCounter` System object analyzes a video stream and returns the current count of lines per frame and pixels per line. The object also delays control signals of the pixel stream to correspond with the count result. Use this object for algorithms that use the location of a pixel in a frame or region of interest.

This waveform shows the pixel-stream control signals and the resulting counter outputs for the first two lines of a video frame that has 10 pixels per line. The `visionhdl.HVCounter` System object has a latency of two cycles to return the current counter values.

To count active lines-per-frame and active pixels-per-line of a pixel stream:

1 Create the `visionhdl.HVCounter` 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

```
HVcntr = visionhdl.HVCounter
HVcntr = visionhdl.HVCounter(Name,Value)
```

Description

`HVcntr = visionhdl.HVCounter` returns a System object that counts the line and pixel location within a frame or region-of-interest. An object that uses the default values for the properties can count frames up to (511 pixels)-by-(355 lines).

`HVcntr = visionhdl.HVCounter(Name,Value)` returns a System object sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example, `HVcnt =`

visionhdl.HVCounter('ActivePixelsPerLine',1920,'ActiveVideoLines',1080) creates an object that can count frames up to (2047 pixels)-by-(2047 lines).

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.

ActivePixelsPerLine — Maximum pixels per line
320 (default) | positive integer

Maximum pixels per line, specified as a positive integer. The object implements a pixel counter that has `ceil(log`$_2$`(ActivePixelsPerLine))` bits. For example, for a frame with 320 pixels per line, the counter has 9 bits. For a frame with 1024 pixels per line, the counter has 11 bits.

ActiveVideoLines — Maximum lines per frame
240 (default) | positive integer

Maximum lines per frame, specified as a positive integer. The object implements a line counter that has `ceil(log`$_2$`(ActiveVideoLines))` bits. For example, for a frame with 240 pixels per line, the counter has 8 bits. For a frame with 1080 pixels per line, the counter has 11 bits.

Usage

Syntax

[hcount,vcount,ctrlout] = HVcntr(ctrlin)

Description

[hcount,vcount,ctrlout] = HVcntr(ctrlin) updates horizontal and vertical location counters hcount and vcount based on pixel-stream control signals, ctrlin. The ctrlout output is a delayed version of ctrlin that matches the timing of the counter values.

This System object uses a structure for frame control signals associated with each pixel of a pixel stream. This interface enables the object to operate independently of image size and format. All Vision HDL Toolbox System objects use the same streaming interface. The object accepts and returns a structure containing five control signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the visionhdl.FrameToPixels System object. For a full description of the interface, see "Streaming Pixel Interface".

Input Arguments

ctrlin — Control signals accompanying input pixel stream
pixelcontrol structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`hcount` — Pixel location in line
positive integer

Pixel location within a line, returned as a positive integer. The object clears the pixel count at the start of each line. If the input pixels per line exceeds the size of the counter, the object returns a saturated value until the start of the next line.

Data Types: `fi(0,ceil(log2(ActiveVideoLines)),0)`

`vcount` — Line location in frame
positive integer

Line location within the frame or region of interest, returned as a positive integer. The object clears the line count at the start of each frame. If the input lines per frame exceeds the size of the counter, the object returns a saturated value until the start of the next frame.

Data Types: `fi(0,ceil(log2(ActiveVideoLines)),0)`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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
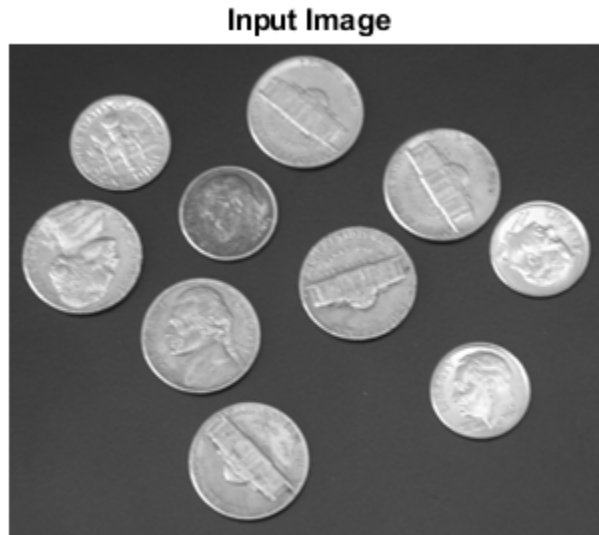reset   Reset internal states of System object

Examples

Count Pixel Locations in Streaming Video

Obtain the current pixel location within a region of interest (ROI).

Load an input frame.

```
frmOrig = imread('coins.png');
[frmActiveLines,frmActivePixels] = size(frmOrig);
imshow(frmOrig)
title 'Input Image'
```

Input Image

Create a serializer object and define inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels(...
    'NumComponents',1,...
    'VideoFormat','custom',...
    'ActivePixelsPerLine',frmActivePixels,...
    'ActiveVideoLines',frmActiveLines,...
    'TotalPixelsPerLine',frmActivePixels+20,...
    'TotalVideoLines',frmActiveLines+20,...
    'StartingActiveLine',3,...
    'FrontPorch',10);
```

Create an object to select a small region of interest. Define a rectangular region by the coordinates of the top-left corner and the dimensions.

```
hPos = 80;
vPos = 60;
hSize = 15;
vSize = 20;
roicoin = visionhdl.ROISelector('Regions',[hPos vPos hSize vSize]);
```

Serialize the test image by calling the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmOrig);
```

Prepare to process pixels by preallocating output vectors. The output frame from the ROI object is the same size as the input frame, but the control signals indicate a different active region. The counter block returns a delayed version of the control signals that aligns with the counter values.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixROIOut = uint8(zeros(numPixelsPerFrame,1));
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
ctrlOut2 = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

Write a function that creates and calls a visionhdl.HVCounter System object™. This object returns horizontal and vertical count values that represent the current pixel location within the region of interest. You can generate HDL code from this function.

```
function  [hCount,vCount,ctrlOut] = pixelCount(ctrlIn)
%pixelCount
% Returns the current active pixel count per line, hCount, and the active
% line count, vCount, based on the pixel-stream control signals.
% ctrlIn and ctrlOut are structures that contain control signals associated
% with the pixel.
% The object implements counters that accommodate the next power-of-two above
% each property value. This configuration can count frames up to (512
% pixels)-by-(256 lines).
% You can generate HDL code from this function.

  persistent hvcount;
  if isempty(hvcount)
    hvcount = visionhdl.HVCounter(...
      'ActivePixelsPerLine',320,...
      'ActiveVideoLines',240);
  end

 [hCount,vCount,ctrlOut] = hvcount(ctrlIn);
end
```

For each pixel in the padded frame, apply the region mask, and then call the counter to indicate the active pixel locations. If the input control signals indicate an inactive pixel at that location, the object returns hCount and vCount set to zero.

```
oldvCount = 0;
for p = 1:numPixelsPerFrame
    [pixROIOut(p),ctrlOut(p)] = roicoin(pixIn(p),ctrlIn(p));
    [hCount,vCount,ctrlOut2(p)] = pixelCount(ctrlOut(p));
    if (vCount ~=0 && vCount ~= oldvCount)
        fprintf('\n Valid pixels in line #%d : ',vCount);
        oldvCount = vCount;
    end
    if hCount ~= 0
        fprintf('#%d ',hCount);
    end
end
```

```
 Valid pixels in line #1 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
 Valid pixels in line #2 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
 Valid pixels in line #3 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
 Valid pixels in line #4 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
```

```
Valid pixels in line #5 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #6 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #7 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #8 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #9 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #10 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #11 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #12 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #13 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #14 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #15 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #16 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #17 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #18 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #19 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
Valid pixels in line #20 : #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15
```

Version History

Introduced in R2019a

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
HV Counter

visionhdl.ImageFilter

Package: visionhdl

2-D FIR filtering

Description

The `visionhdl.ImageFilter` System object performs two-dimensional finite-impulse-response (FIR) filtering on a pixel stream. It supports the use of programmable filter coefficients.

To filter a pixel stream:

1   Create the `visionhdl.ImageFilter` 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

```
F = visionhdl.ImageFilter(Name,Value)
F = visionhdl.ImageFilter(coeff,lineSize,Name,Value)
```

Description

`F = visionhdl.ImageFilter(Name,Value)` returns a 2-D FIR filter System object that filters a pixel stream. Set properties using one or more name-value pairs. Enclose each property name in single quotes.

`F = visionhdl.ImageFilter(coeff,lineSize,Name,Value)` returns a 2-D FIR filter System object, with the `Coefficients` property set to `coeff`, the `LineBufferSize` property set to `lineSize`, and additional options specified by one or more `Name,Value` pair arguments.

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.

CoefficientsSource — Source to provide filter coefficients
`'Property'` (default) | `'Input port'`

Source of the filter coefficients, specified as one of these values.

- 'Property' — The object uses the filter coefficients specified in the `Coefficients` property.
- 'Input port' — The object uses the filter coefficients from the `coeff` argument.

Coefficients — Coefficients of filter
[1,0;0,-1] (default) | matrix

Coefficients of the filter, specified as a matrix. Each dimension of the matrix must have at least 2 elements and no more than 64 elements.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Dependencies

This property applies when you set `CoefficientsSource` to `'Property'`.

PaddingMethod — Method for padding
'Constant' (default) | 'Replicate' | 'Symmetric' | 'Reflection'

Methods for padding the boundary of the input image, specified as one of these values.

- 'Constant' — Interpret pixels outside the image frame as having a constant value.
- 'Replicate' — Repeat the value of pixels at the edge of the image.
- 'Symmetric' — Set the value of the padding pixels to mirror the edge of the image.
- 'Reflection' — Set the value of the padding pixels to reflect around the pixel at the edge of the image.
- 'None' — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the *n*/2 pixels around the edge of the frame for downstream operations. *n* is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

For more information about these methods, see "Edge Padding".

PaddingValue — Value used to pad boundary of input image
0 (default) | integer

Value used to pad the boundary of the input image, specified as an integer. The object casts this value to the same data type as the input pixel.

Dependencies

This parameter applies when you set `PaddingMethod` to `'Constant'`.

LineBufferSize — Size of line buffer
2048 (default) | integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (*coefficient rows – 1*)-by-`LineBufferSize` memory locations to store the pixels.

CoefficientsDataType — Method to determine data type of filter coefficients
'Same as first input' (default) | 'Custom'

Method for determining the data type of the filter coefficients, specified as one of these values.

- 'Same as first input'' — Sets the data type of the coefficients to match the data type of the pixelin argument.
- 'Custom' — Sets the data type of the coefficients to match the data type defined in the CustomCoefficientsDataType property.

When converting the coefficients to the specified data type, the object rounds to the nearest representable value and saturates on overflow.

Dependencies

This property applies when you set CoefficientsSource to 'Property'.

CustomCoefficientsDataType — Data type for the filter coefficients
numerictype(true,16,15) (default) | numerictype(S,WL,FL)

Data type for the filter coefficients, specified as numerictype(S,WL,FL), where *S* is 1 (true) for signed and 0 (false) for unsigned, *WL* is the word length, and *FL* is the fraction length in bits.

Dependencies

This property applies when you set CoefficientsDataType to 'Custom'.

OutputDataType — Method to determine data type of output pixels
'Same as first input' (default) | 'Full precision' | 'Custom'

Method for determining the data type of the output pixels, specified as one of these values.

- 'Same as first input' — Sets the data type of the output pixels to match the data type of the pixelin argument.
- 'Full precision' — Computes internal and output data types using full precision rules. These rules provide accurate fixed-point numerics and prevent quantization within the object. The object adds bits as needed to prevent rounding and overflow.
- 'Custom' — Sets the data type of the output pixels to match the data type in the CustomOutputDataType property.

CustomOutputDataType — Data type for the filter coefficients
numerictype(true,8,0) (default) | numerictype(S,WL,FL)

Data type for the output pixels, specified as numerictype(S,WL,FL), where *S* is 1 (true) for signed and 0 (false) for unsigned, *WL* is the word length, and *FL* is the fraction length in bits.

Dependencies

This property applies only when you set OutputDataType to 'Custom'.

RoundingMethod — Rounding mode for fixed-point operations
'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Zero'

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is single or double.

OverflowAction — Overflow mode used for fixed-point operations
'Wrap' (default) | 'Saturate'

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is `single` or `double`.

Usage

Syntax

```
[pixelout,ctrlout] = F(pixelin,ctrlin)
[pixelout,ctrlout] = F(pixelin,ctrlin,coeff)
```

Description

[pixelout,ctrlout] = F(pixelin,ctrlin) returns the next pixel, `pixelout`, of the filtered image resulting from applying the coefficients in the `Coefficients` property to the image described by the input pixel stream, `pixelin`.

[pixelout,ctrlout] = F(pixelin,ctrlin,coeff) returns the next pixel, `pixelout`, of the filtered image resulting from applying the coefficients in the `coeff` argument to the image described by the input pixel stream, `pixelin`. The object samples the values from the `coeff` argument only at the start of a frame and ignores any changes within a frame. To enable this syntax, set the `CoefficientsSource` property to `'Input port'`.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel stream
scalar

Single pixel, specified as a scalar value.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Data Types: uint | int | fi | double | single

ctrlin — Control signals accompanying input pixel stream
pixelcontrol structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

coeff — Filter coefficients
matrix

Filter coefficients, specified as a matrix. Each dimension of the matrix must have at least 2 elements and no more than 16 elements. The object samples the values from the `coeff` argument only at the start of a frame and ignores any changes within a frame.

Dependencies

To enable this argument, set the `CoefficientsSource` property to `'Input port'`.

Data Types: `uint` | `int` | `fi` | `double` | `single`

Output Arguments

pixelout — Output pixel
scalar

Single filtered pixel, returned as a scalar value.

Configure the data type of the output pixel by using the `OutputDataType` and `CustomOutputDataType` properties.

Data Types: `uint` | `int` | `fi` | `double` | `single`

ctrlout — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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
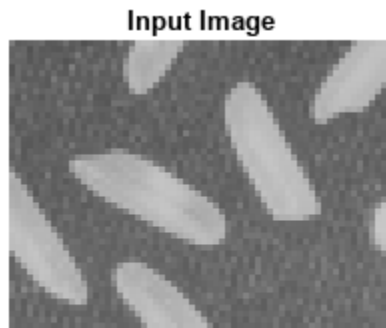reset   Reset internal states of System object

Examples

Filter Pixel Stream

This example implements a 2-D blur filter on a thumbnail image.

Load the source image from a file. Select a portion of the image matching the desired test size.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Input Image

Create a serializer object and specify the size of the inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',6, ...
      'FrontPorch',5);
```

Create a filter object.

```
 filt2d = visionhdl.ImageFilter( ...
         'Coefficients',ones(2,2)/4, ...
         'CoefficientsDataType','Custom', ...
         'CustomCoefficientsDataType',numerictype(0,1,2), ...
         'PaddingMethod','Symmetric');
```

Serialize the test image by calling the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

2-127

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the filtered value. Monitor the control signals to determine the latency of the object. The latency of a filter configuration depends on:

- The number of active pixels in a line.
- The size of the filter kernel.
- Optimization of symmetric or duplicate coefficients.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = filt2d(pixIn(p),ctrlIn(p));
    if (ctrlOut(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
sprintf('object latency is %d cycles',foundValOut-foundValIn)

ans =
'object latency is 101 cycles'
```

Create a deserializer object with a format that matches the format of the serializer. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```

Output Image

Algorithms

This System object implements the algorithms described on the Image Filter block reference page.

Version History
Introduced in R2015a

R2022a: Two pixels-per-clock streaming

The object now supports multipixel streams that have 2 pixels per clock cycle.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

R2019b: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2019a: Increased kernel size limits

The object now allows for a coefficient kernel with up to 64-by-64 elements.

R2018b: Programmable coefficients
Behavior changed in R2018b

The object now accepts coefficients from an input argument. Each dimension of the matrix must have at least 2 and no more than 16 elements. The object samples the values from the `coeff` argument at the start of a frame only and ignores any changes within a frame.

In previous releases, you could specify a row vector of coefficients, that is, a matrix of 1-by-N elements. Now, the coefficient matrix must have at least 2 elements in each dimension.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Image Filter | imfilter | visionhdl.FrameToPixels

visionhdl.ImageResizer

Package: visionhdl

Change dimensions of image in a pixel stream

Description

The visionhdl.ImageResizer System object downscales greyscale input images. The horizontal and vertical scale factors are independent. The object provides an optional antialiasing filter, and you can choose nearest-neighbor or bilinear interpolation.

You can use this System object for image resize operations such as:

- Converting images to a uniform size for input to deep learning algorithms
- Modifying disparate source videos for processing, resource, or display requirements

To change the dimensions of an image in a pixel stream:

1   Create the visionhdl.ImageResizer 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

resizer = visionhdl.ImageResizer(Name=Value)

Description

resizer = visionhdl.ImageResizer(Name=Value) returns a System object that resizes the active image in a pixel stream. Set properties 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.

InputSize — Dimensions of active frame in input stream
[1080,1920] (default) | [*ActiveVideoLines*,*ActivePixelsPerLine*]

Dimensions of the input frame, specified as a row vector of the form [*ActiveVideoLines,ActivePixelsPerLine*]. The dimensions must be integers greater than or equal to [16,16] and less than or equal to [4320,8192].

OutputSize — Dimensions of active frame in output stream
[720,1280] (default) | [*ActiveVideoLines,ActivePixelsPerLine*]

Dimensions of the output frame, specified as a matrix of the form [*ActiveVideoLines,ActivePixelsPerLine*]. The dimensions must be integers greater than or equal to [16,16] and less than or equal to [4320,8192].

The output dimensions must be less than or equal to the input dimensions.

InsertHorizontalBlanking — Add invalid cycles between active lines
false (default) | true

The horizontal blanking interval is the period of inactive cycles between the end of one line and the beginning of the next line. When you set this property to true, the object includes a buffer to store the output pixels while inserting extra horizontal blanking cycles between the active lines of the output video stream. Set the number of blanking cycles in the MinOutputHorizontalBlanking property. For information on when to use horizontal blanking, see "Configure Blanking Intervals".

MinOutputHorizontalBlanking — Number of invalid cycles to insert between output lines
370 (default)

The horizontal blanking interval is the period of inactive cycles between the end of one line and the beginning of the next line. If your downstream processing requires a certain blanking interval, specify that interval in this property. This value must be an integer between 1 and 1195. The object includes a buffer that stores the output pixels until the minimum number of blanking cycles is reached.

The recommended minimum horizontal blanking interval for Vision HDL Toolbox System objects that use a line buffer is 2×*Kw* when using padding or 12 cycles when you set the PaddingMethod property to 'None'. For more information on blanking requirements for image processing systems, see "Configure Blanking Intervals".

Dependencies

To enable this property, set the InsertHorizontalBlanking property to true.

InterpolationMethod — Method of interpolating pixel values
'Bilinear' (default) | 'Nearest Neighbor'

Method of interpolating pixel values in horizontal and vertical directions, specified as one of these types:

- 'Bilinear' — The object implements a two-tap filter where the taps are the current and previous pixel values.
- 'Nearest Neighbor' — The object implements muxes to select the closest of the current pixel value or the previous pixel value. This interpolation method uses the fewest hardware resources, at the tradeoff of coarser interpolation accuracy.

For more information about interpolation methods, see the "Change Image Size" example.

Nearest Neighbour Interpolation   Bilinear Interpolation
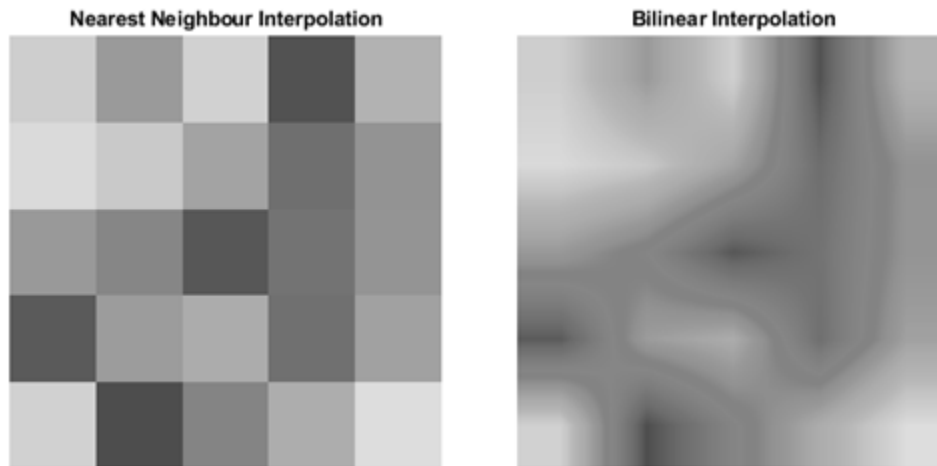
HasAntialiasingFilter — Apply an antialiasing filter before resizing
true (default) | false

When you set this property to true, the object applies an antialiasing filter to the input image before the resize operation. The antialiasing filter is a separable 5-by-5 filter. The coefficients are [1 4 6 4 1]/16.

RoundingMethod — Rounding mode for fixed-point operations
'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Zero'

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is single or double.

OverflowAction — Overflow mode used for fixed-point operations
'Wrap' (default) | 'Saturate'

When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This property does not apply when the input data type is single or double.

CoefficientsDataType — Data type of filter coefficients
numerictype(0,16) (default) | numerictype(0,WL)

Data type for the filter coefficients, specified as numerictype(0,WL), where the type is unsigned and *WL* is a word length in the range 2 to 17. The object computes the binary point scaling to accommodate the interpolation coefficients. When converting the coefficients to the specified data type, the object uses the settings of the RoundingMethod and OverflowAction properties.

The interpolation filter also limits the internal pixel data types to 24 bits wide so that the filter maps to DSP blocks on your FPGA.

OutputDataType — Method to determine data type of output pixels
'Same as first input' (default) | numerictype(0,16,0) | numerictype(0,WL,0)

Specify the data type for the output pixels as 'Same as first input' or as a data type expression in the form numerictype(0,WL,0), where the type is unsigned and *WL* is in the range 8 to 16.

- 'Same as first input' — Sets the data type of the output pixels to match the data type of the input pixel argument.

- <data type expression> — Sets the data type of the output pixels to this data type.

When converting the pixel values to the specified data type, the object uses the settings of the RoundingMethod and OverflowAction properties.

Usage

Syntax

[pixelout,ctrlout] = resizer(pixelin,ctrlin)

Description

[pixelout,ctrlout] = resizer(pixelin,ctrlin) returns the next pixel, pixelout, of the resized image resulting from resampling the image described by the input pixel stream, pixelin, according to the properties of the object.

Input Arguments

pixelin — Input pixel stream
scalar

Single pixel, specified as a scalar value.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: uint8 | uint16 | fi(0,WL,0), WL between 8 and 16 | double | single

ctrlin — Control signals accompanying input pixel stream
pixelcontrol structure

Control signals accompanying the input pixel stream, specified as a pixelcontrol structure containing five logical data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: struct

Output Arguments

pixelout — Output pixel stream
scalar

Single pixel of a resized pixel stream, returned as a scalar value.

Configure the data type of the output pixel by using the OutputDataType property.

Data Types: uint8 | uint16 | fi(0,WL,0), WL between 8 and 16 | double | single

ctrlout — Control signals accompanying output pixel stream
pixelcontrol structure

Control signals accompanying the output pixel stream, returned as a pixelcontrol structure containing five logical data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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

Version History
Introduced in R2023a

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also
Image Resizer

visionhdl.ImageStatistics

Package: `visionhdl`

Mean, variance, and standard deviation

Description

The `visionhdl.ImageStatistics` System object calculates the mean, variance, and standard deviation of streaming video data. The object performs each calculation over all pixels in the input region of interest (ROI) and implements the calculations by using hardware-efficient algorithms.

- To change the size and dimensions of the ROI, you can manipulate the input video stream control signals. See "Regions of Interest" on page 1-137.
- The number of valid pixels in the input image affect the accuracy of the mean approximation. To avoid approximation errors, use an image that contains fewer than 64 pixels, a multiple of 64 pixels up to 64^2 pixels, a multiple of 4096 pixels up to 64^3 pixels, or a multiple of 64^3 pixels up to 64^4 pixels. For details of the mean approximation, see "Algorithm" on page 1-134.
- The object calculates statistics over frames up to 64^4 (16,777,216) pixels in size.

To calculate mean, variance, and standard deviation:

1 Create the `visionhdl.ImageStatistics` 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

```
stats = visionhdl.ImageStatistics(Name,Value)
```

Description

`stats = visionhdl.ImageStatistics(Name,Value)` returns a System object that calculates statistics from an input image region. Set properties using one or more name-value pairs. Enclose each property name in single quotes.

For example, when you configure the object like this, the object returns mean and standard deviation, but not variance:

```
stats = visionhdl.ImageStatistics('mean',true, ...
            'variance',false, ...
            'stdDev',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.

`mean` — Calculate mean of each input frame
`true` (default) | `false`

If you set this property to `false`, the object does not return the mean.

`variance` — Calculate variance of each input frame
`true` (default) | `false`

If you set this property to `false`, the object does not return the variance.

`stdDev` — Calculate standard deviation of each input frame
`true` (default) | `false`

If you set this property to `false`, the object does not return the standard deviation.

Usage

Syntax

`[mean,variance,stddeviation,validout] = stats(pixelin,ctrlin)`

Description

`[mean,variance,stddeviation,validout] = stats(pixelin,ctrlin)` incorporates one pixel value, `pixelIn`, into ongoing calculations of video frame statistics. The control signals associated with each pixel, `ctrlin`, indicate the frame boundaries. When `validout` is `true`, the output values of `mean`, `variance`, and `stddeviation` represent the statistics for the most recent input frame completed. The number of statistics returned depends on the object property settings.

This object uses a streaming pixel interface with a structure for frame control signals. The interface enables the object to operate independently of image size and format and connect with other Vision HDL Toolbox objects. The object accepts pixel data as integer, fixed-point, or floating-point data types. The object accepts control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a full description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel stream
scalar

Single pixel, specified as an unsigned scalar value.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `single` | `double` | `fixdt(0,W,0)`, `W = 8,9,...,16`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`mean` — Mean
scalar

Mean of the most recent frame of video input, returned as a scalar value.

The data type is the same as `pixelin`.

`variance` — Variance
scalar

Variance of the most recent frame of video input, returned as a scalar value.

The data type is the same as `pixelin`. The fixed-point output word length is double the input word length.

`stdDeviation` — Standard deviation
scalar

Standard deviation of the most recent frame of video input, returned as a scalar value.

The data type is the same as `pixelin`. Fixed-point output word length is double the input word length.

`validout` — Indicates valid output data
scalar `logical`

Validity of output statistics. When the object completes the calculations, it returns `true`. When this output is `true`, the other output arguments are valid. Data type is `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 Statistics of Image

This example computes the mean, variance, and standard deviation of a thumbnail image.

Load the source image from a file. Select a portion of the image matching the desired test size.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```
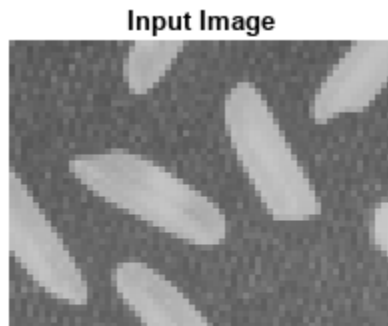

Create a serializer object and define inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+10,...
      'TotalVideoLines',frmActiveLines+10,...
      'StartingActiveLine',6,...
      'FrontPorch',5);
```

Create an object that returns mean, variance, and standard deviation.

```
 stats = visionhdl.ImageStatistics();
```

Serialize the test image by calling the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
validOut  = false(numPixelsPerFrame,1);
mean  = zeros(numPixelsPerFrame,1,'uint8');
variance  = zeros(numPixelsPerFrame,1,'uint8');
stddev  = zeros(numPixelsPerFrame,1,'uint8');
```

For each pixel in the stream, increment the internal statistics.

```
for p = 1:numPixelsPerFrame
    [mean(p),variance(p),stddev(p),validOut(p)] = stats(pixIn(p),ctrlIn(p));
end
```

The results are valid when `validOut` is returned `true`.

```
mean = mean(validOut==1)

mean = uint8
    125

variance = variance(validOut==1)

variance = uint8
    255

stddev = stddev(validOut==1)

stddev = uint8
    36
```

Algorithms

This object implements the algorithms described on the Image Statistics block reference page.

Version History
Introduced in R2015a

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Image Statistics | `vision.Mean` | `vision.Variance` | `vision.StandardDeviation` | `mean2` | `std2` | `visionhdl.FrameToPixels`
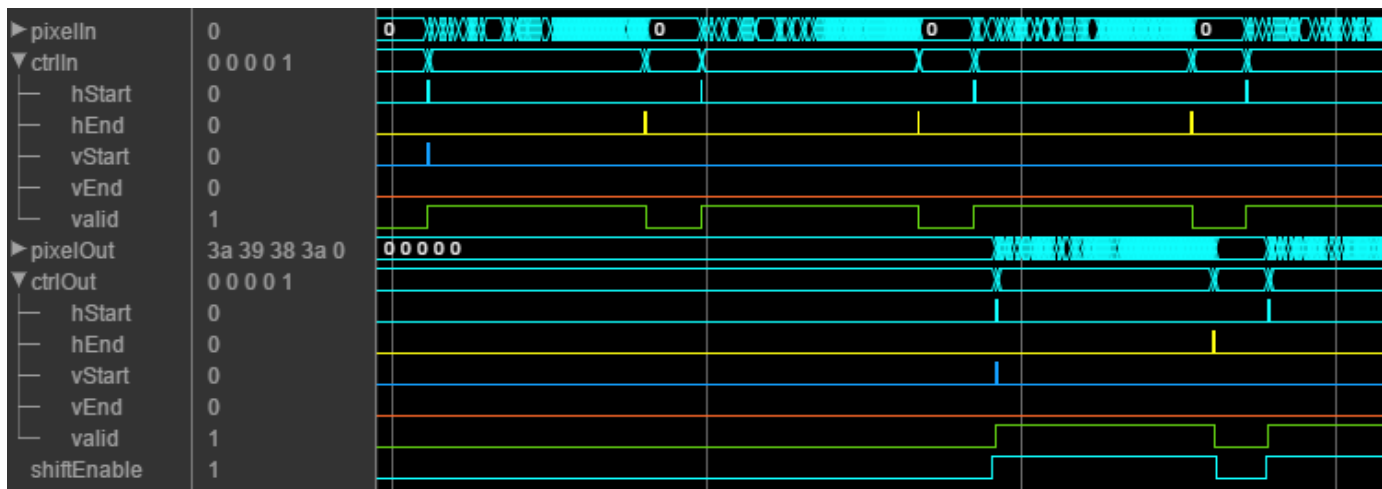
visionhdl.LineBuffer

Package: visionhdl

Store video lines and return neighborhood pixels

Description

The `visionhdl.LineBuffer` System object selects neighborhood pixels from streaming image data. It handles video control signals and edge padding, and is pipelined for high-speed video designs. The object outputs one column of the neighborhood at a time. To compose a neighborhood for further processing, use the `shiftEnable` signal to store the output columns, including padding, in a shift register. This object allows you to share the line buffer resources when your design performs multiple operations on the same neighborhood.

The following waveform shows the `visionhdl.LineBuffer` object returning 5-by-1 pixel columns that make up a 5-by-5 neighborhood. The time frame shown is at the beginning (top-left corner) of an input frame. The output starts after the object has stored two (`floor(M/2)`) lines and is receiving the start of the third line. The `shiftEnable` signal is asserted two cycles earlier than the output `ctrl.valid` signal, which indicates that the first two (`floor(M/2)`) columns are exclusively padding pixels. Similarly, `shiftEnable` stays high for two extra cycles at the end of the line.

To extract sliding pixel neighborhoods from a video stream:

1 Create the `visionhdl.LineBuffer` 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

`lineMem = visionhdl.LineBuffer(Name,Value)`

Description

`lineMem = visionhdl.LineBuffer(Name,Value)` returns a line buffer System object. Set properties using name-value pairs. Enclose each property name in single quotes.

Example: `lineMem = visionhdl.LineBuffer('NeighborhoodSize',[5 5])`

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.

NeighborhoodSize — Size of output neighborhood
[3 3] (default) | 2-element row vector of integer dimensions

Size of output neighborhood to be formed, specified as a 2-element row vector of integer dimensions [*vertical horizontal*]. The object returns a column vector of *vertical* elements. The *horizontal* dimension is used to determine padding.

PaddingMethod — Method for padding the boundary of input image
'Symmetric' (default) | 'Constant' | 'Replicate' | 'Reflection' | 'None'

Methods for padding the boundary of the input image, specified as one of these values.

- `'Constant'` — Interpret pixels outside the image frame as having a constant value.
- `'Replicate'` — Repeat the value of pixels at the edge of the image.
- `'Symmetric'` — Set the value of the padding pixels to mirror the edge of the image.
- `'Reflection'` — Set the value of the padding pixels to reflect around the pixel at the edge of the image.
- `'None'` — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the $n/2$ pixels around the edge of the frame for downstream operations. n is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

For more information about these methods, see "Edge Padding".

PaddingValue — Value used to pad boundary of input image
0 (default) | integer

Value used to pad the boundary of the input image, specified as an integer. The object casts this value to the same data type as the input pixel.

Dependencies

This parameter applies when you set `PaddingMethod` to `'Constant'`.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

Usage

Syntax

`[pixelout,ctrlout,shiftEnable] = lineMem(pixelin,ctrlin)`

Description

`[pixelout,ctrlout,shiftEnable] = lineMem(pixelin,ctrlin)` returns a column vector of pixels in the kernel, and accompanying control signals.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel stream
scalar

Single image pixel in a pixel stream, specified as a scalar value representing intensity.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `int` | `uint` | `fi` | `logical` | `double` | `single`

ctrlin — Control signals accompanying input pixel stream
pixelcontrol structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

2-145

Data Types: `struct`

Output Arguments

`pixelout` — Single column of neighborhood
column vector

Single column neighborhood pixel values, returned as a of 1-by-*M* vector, where *M* is the vertical neighborhood dimension. The output pixel data type is the same as the data type of the input pixels.

To compose a neighborhood for further processing, use the `shiftEnable` signal to store the output columns, including padding, in a shift register.

Note HDL code generation is supported for small matrices, but matrix operations can impact hardware performance and resource usage. Therefore, minimize how much your design operates on an *N*-by-*M* neighborhood directly. You can :

- Separate a filter into vertical and horizontal components.
- Concatenate the neighborhood pixels into a *N*M*-by-1 vector.

These design suggestions also provide opportunities to add pipelining around each adder or multiplier to increase synthesized clock speed and fit the design to DSP blocks on an FPGA.

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

`ctrlout` — Control signals accompanying output column
`pixelcontrol` structure

Control signals accompanying pixel stream, returned as a structure containing five `logical` signals. The signals describe the validity of the center pixel of the column, and the location of that pixel within the frame. See "Pixel Control Structure".

Columns that contain only padding pixels do not assert `ctrl.valid`. The `shiftEnable` signal is asserted for both padding and active columns.

Note For most of the frame, the object returns the input control signals that arrived with the bottom pixel of the column. However, for the final `floor(M/2)` lines of each frame, the bottom pixel of the column is a padding pixel, so the object generates output control signals. The object generates a contiguously asserted `ctrl.valid` signal for the valid pixels in each line.

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

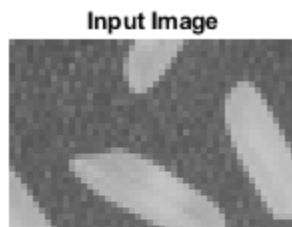### Construct an Averaging Filter Using a Line Buffer

The `visionhdl.LineBuffer` System object™ stores video lines and returns sliding neighborhoods
for kernel-based image operations. This example calculates the average of each neighborhood.

Import image source and set up serializer and deserializer objects.

```
frmOrig = imread('rice.png');
frmActivePixels = 48;
frmActiveLines = 32;
frmIn = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmIn,'InitialMagnification',300)
title 'Input Image'

frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+10,...
      'TotalVideoLines',frmActiveLines+10,...
      'StartingActiveLine',6,...
      'FrontPorch',5);
[~,~,numPixPerFrm] = getparamfromfrm2pix(frm2pix);

pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+10);
```


Input Image

Write a function that creates and calls the System object™. The object returns one column of the
neighborhood at a time. Use a shift register to save the columns. Then, calculate the average of the
pixel neighborhood. You can generate HDL from this function.

```matlab
function  [pixOut,ctrlOut] = AvgFilter(pixIn,ctrlIn)
%AvgFilter
% Calculates the average pixel value for each 3x3 kernel
% pixIn and pixOut are scalar uint8 pixel values.
% ctrlIn and ctrlOut are structures that contain control signals associated
% with the pixel.
% You can generate HDL code from this function.

  persistent linemem;
  if isempty(linemem)
      linemem = visionhdl.LineBuffer;
  end
  persistent dataShiftReg;
  if isempty(dataShiftReg)
       % use typeof(pixIn)?
      dataShiftReg = fi(zeros(3,3),0,8,0);
  end
  % HDL code gen does not support arrays of structs
  persistent controlShiftReg1;
  persistent controlShiftReg2;
  persistent controlShiftReg3;
  if isempty(controlShiftReg1)
      controlShiftReg1 = pixelcontrolstruct();
      controlShiftReg2 = pixelcontrolstruct();
      controlShiftReg3 = pixelcontrolstruct();
  end

  persistent div9;
  if isempty(div9)
      div9 = fi(1/9,0,12,15);
  end

 % Advance shift registers
 dataShiftReg(:,2:end) = dataShiftReg(:,1:end-1);
 controlShiftReg3 = controlShiftReg2;
 controlShiftReg2 = controlShiftReg1;

 % Fetch next column
 [dataShiftReg(:,1),controlShiftReg1] = linemem(fi(pixIn,0,8,0),ctrlIn);

 % Calculate the average over the neighborhood
 pixOut = uint8((sum(dataShiftReg(:),'native')).*div9);
 ctrlOut = controlShiftReg3;

end
```

Process the image by calling the function for each pixel.

```matlab
pixOutVec = ones(numPixPerFrm,1,'uint8');
ctrlOutVec = repmat(pixelcontrolstruct(false,false,false,false,false),numPixPerFrm,1);

[pixInVec,ctrlInVec] = frm2pix(frmIn);
for p = 1:numPixPerFrm
    [pixOutVec(p),ctrlOutVec(p)] = AvgFilter(pixInVec(p),ctrlInVec(p));
end
```
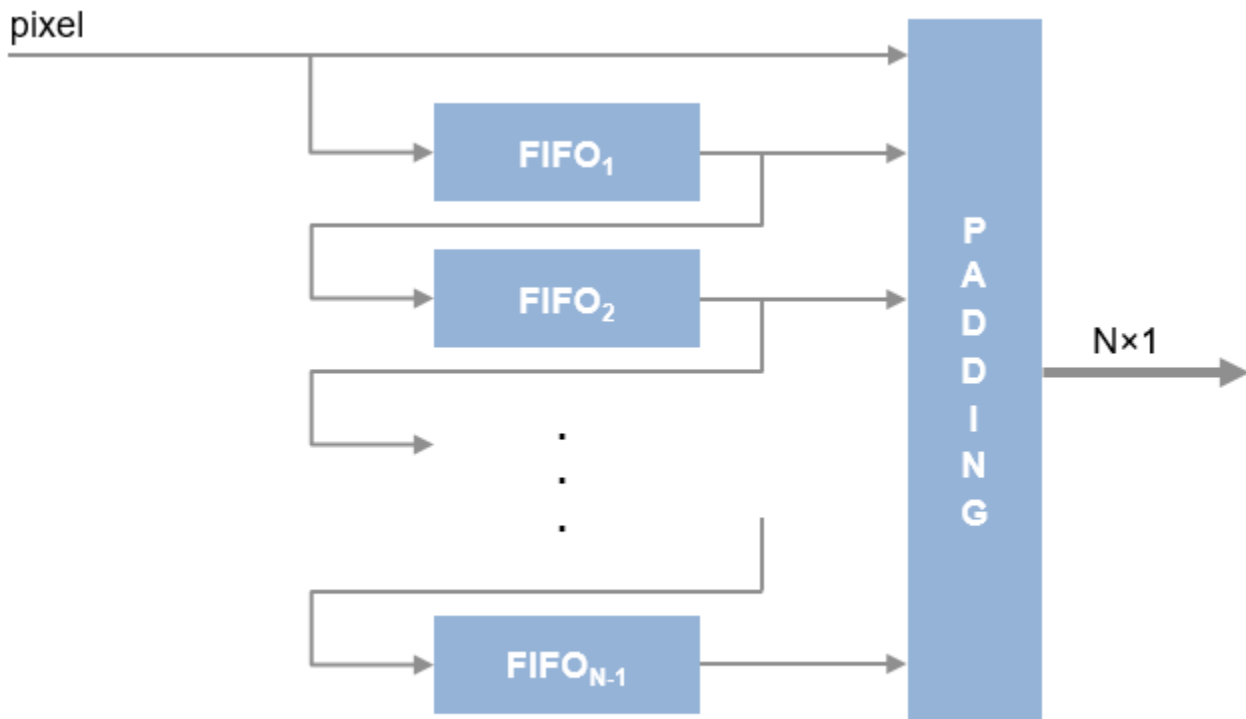
Recreate the filtered frame.

```
[frmOut,frmValid] = step(pix2frm,pixOutVec,ctrlOutVec);

if frmValid
    figure;
    imshow(frmOut,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image
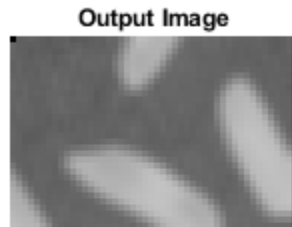
Algorithms

The object stores $M – 1$ lines of valid pixels, as specified by the neighborhood size. It adds padding bits at the edge of the frame. The object returns the first output column once it can form a complete neighborhood column, which occurs at the start of input line `floor(M/2)`.

Version History
Introduced in R2017b

R2022a: Two pixels-per-clock streaming

The object now supports multipixel streams that have 2 pixels per clock cycle.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

R2019b: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2018b: Improved line buffer
Behavior changed in R2018b

The line buffer now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
Line Buffer

Objects
visionhdl.FrameToPixels

visionhdl.LookupTable

Package: visionhdl

Map input pixel to output pixel using custom rule

Description

The visionhdl.LookupTable System object uses a custom one-to-one map to convert between an input pixel value and an output pixel value.

To map an input pixel value to an output pixel value:

1   Create the visionhdl.LookupTable 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

LUT = visionhdl.LookupTable(tabledata)

Description

LUT = visionhdl.LookupTable(tabledata) returns a lookup table System object that performs a one-to-one mapping between an input pixel and an output pixel. The mapping is defined by the Table property, which is set to the value of tabledata.

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.

Table — Map between input pixel and output pixel
uint8(0:1:255) (default) | vector

Map between input pixel and output pixel, specified as a row or column vector of any data type. The data type of the table data determines that of pixelout. This mapping determines the one-to-one correspondence between the input pixelin value and the output pixelout value.

- The length of the table data must equal $2^{WordLength}$, where *WordLength* is the size, in bits, of pixelin. This object does not perform interpolation. Every input value must have a corresponding output value in the table.

- The smallest representable value of the input data type maps to the first element of the table, the second smallest value maps to the second element, and so on. For example, if `pixelin` has a data type of `fixdt(0,3,1)`, the input value 0 maps to the first element of the table, 0.5 maps to the second element, 1 maps to the third element, and so on.

Example: `uint8(linspace(255,0,256))`

Usage

Syntax

`[pixelout,ctrlout] = LUT(pixelin,ctrlin)`

Description

`[pixelout,ctrlout] = LUT(pixelin,ctrlin)` returns the pixel value, `pixelout`, located in the table at the address specified by the input pixel value, `pixelin`. The object delays the input control signals, `ctrlin`, to align the output control signals, `ctrlout`, with the output data.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel stream
scalar | row vector

Input pixel, specified as a scalar intensity value or a row vector of three values representing one pixel in R'G'B' or Y'CbCr color space. Integer and fixed-point data types larger than 16 bits are not supported. Signed data types are not supported.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Data Types: `uint8` | `uint16` | `fi(0,N,F)`, `N <= 16` | `logical`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel stream
scalar | vector

Output pixel, returned as a scalar intensity value or a vector of three values representing R'G'B' or Y'CbCr color space. The data type of the output is the same as the data type of the entries you specify in the `Table` property. The dimensions of the output pixel is the same as the dimensions of the input pixel.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint8` | `uint16` | `int8` | `int16` | `fi` | `logical` | `double` | `single`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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 Negative Image

Create the negative of an image by looking up the opposite pixel values in a table.

Set dimensions of the test image, and load an image source. Select a portion of the image matching the desired test size.

```
frmActivePixels = 64;
frmActiveLines = 48;
frmOrig = imread('rice.png');
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```

Input Image

Create a serializer object and define inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',6, ...
      'FrontPorch',5);
```

Create a lookup table object. The input pixel data type is `uint8`, so the negative value is `255-pixel`. The output pixel data type is the same as the data type of the table contents.

```
tabledata = uint8(linspace(255,0,256));
inverter = visionhdl.LookupTable(tabledata);
```

Serialize the test image. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Initialize the output variables. Then, for each pixel in the padded frame, look up its negative value.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
for p = 1:numPixelsPerFrame
    [pixOut(p),ctrlOut(p)] = inverter(pixIn(p),ctrlIn(p));
end
```

Create deserializer object with a video format matching that of the serializer. Convert the output pixel stream to an image frame, and display the result.

```
pix2frm = visionhdl.PixelsToFrame( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10);
```

```
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image
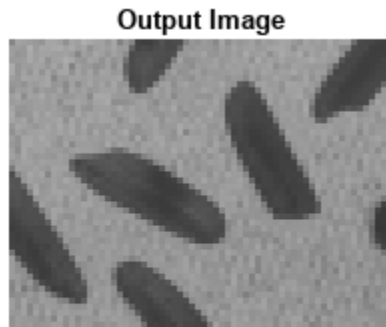
Algorithms

Latency

The latency of this System object is 2 cycles.

Version History
Introduced in R2015a

R2020b: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

HDL Code Generation

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Lookup Table | `visionhdl.FrameToPixels`

visionhdl.MeasureTiming

Package: visionhdl

Measure timing of pixel control structure input

Description

The visionhdl.MeasureTiming System object measures the timing parameters of a video stream. The Vision HDL Toolbox streaming pixel protocol implements the timing of a video system, including inactive intervals between frames. These inactive intervals are called *blanking intervals*. Many Vision HDL Toolbox objects require minimum blanking intervals. You can use the timing parameter measurements from this object to check that your video stream meets these requirements. If you manipulate the control signals of your video stream, you can use this object to verify the resulting control signals. To determine the parameters of each frame, the object measures time steps between the control signals in the input structure.

For details on the pixel control bus and the dimensions of a video frame, see "Streaming Pixel Interface". For more information on blanking intervals, see "Configure Blanking Intervals".

To measure the timing of the pixel control structure input:

1   Create the visionhdl.MeasureTiming 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

```
MT = visionhdl.MeasureTiming
```

Description

MT = visionhdl.MeasureTiming returns a System object that measures the average frame timing of a video stream.

Usage

Syntax

```
[activePixels,activeLines,totalPixels,totalLines,horizBlank,vertBlank] = MT(
ctrlin)
```

Description

[activePixels,activeLines,totalPixels,totalLines,horizBlank,vertBlank] = MT(
ctrlin) incorporates the current pixel location into calculations of video frame timing. The control

signals, `ctrlin`, associated with each pixel indicate the location of this pixel relative to the active frame boundaries. The input object measures the time steps between the control signals to determine the parameters of each frame.

For details about the pixel stream format, see "Streaming Pixel Interface".

Input Arguments

**`ctrlin` — Control signals accompanying input pixel stream**
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

The diagram shows the output measurements, as determined from the pixel stream control signals.
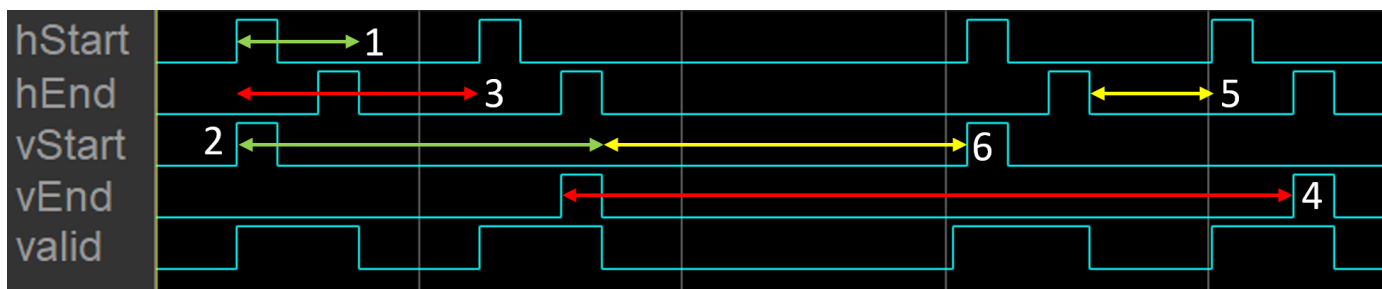
- 1 — Active pixels per line
- 2 — Active lines per frame (count `hStart` pulses)
- 3 — Total pixels per line
- 4 — Total lines per frame (cycles divided by total pixels per line)
- 5 — Horizontal blanking
- 6 — Vertical blanking (cycles, minus horizontal blanking, divided by total pixels per line)

For details on the pixel control bus and the dimensions of a video frame, see "Streaming Pixel Interface".

Note Measurements from the first simulated frame are incorrect because some parameters require measurements between frames. Before using the results, simulate at least two frames.

`activePixels` — Number of active pixels per line
integer

This value is measured between `hStart` and `hEnd`. See marker 1 in the diagram.

Data Types: `double`

`activeLines` — Number of active lines in the frame
integer

This value is measured as the number of `hStart` pulses between `vStart` and `vEnd`. See marker 2 in the diagram.

Data Types: `double`

totalPixels — Number of pixels in the line
integer

This value is measured from `hStart` to the next `hStart`, including the horizontal blanking interval. See marker 3 in the diagram.

Data Types: `double`

totalLines — Number of lines in the frame
integer

This value is measured by the interval from `vEnd` to the next `vEnd`, divided by `totalPixels`. It includes the vertical blanking interval. See marker 4 in the diagram.

Data Types: `double`

horizBlank — Number of pixels in the horizontal blanking interval
integer

The horizontal blanking interval is the number of inactive pixels between lines of a frame. This value is measured between `hEnd` and the next `hStart`. See marker 5 in the diagram.

Data Types: `double`

vertBlank — Number of lines in the vertical blanking interval
integer

The vertical blanking interval is the number of inactive lines between frames. This value is measured from `vEnd` to the next `vStart`, adjusted to remove `horizBlank`, then divided by `totalPixels`. See marker 6 in the diagram.

Data Types: `double`

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

Measure Timing Parameters of Custom Video Stream

This example shows how to use the `MeasureTiming` object to observe the frame parameters in a custom video stream. The example creates customized padding around an image frame and converts the frame to streaming video. It uses the `MeasureTiming` object to confirm that the streaming video parameters match the custom settings.

Use a `FrameToPixels` object to specify a small custom-size frame with customized blanking intervals. To obtain a frame of this size, select a small section of the input image.

```
frm2pix = visionhdl.FrameToPixels( ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',32, ...
      'ActiveVideoLines',18, ...
      'TotalPixelsPerLine',42, ...
      'TotalVideoLines',26, ...
      'StartingActiveLine',6, ...
      'FrontPorch',5);
[actPixPerLine,actLine,numPixPerFrm] = getparamfromfrm2pix(frm2pix);

frmFull = imread('rice.png');
frmIn = frmFull(74:73+actLine,104:103+actPixPerLine);
imshow(frmIn)
```


Create a `MeasureTiming` object to observe the parameters of the serial pixel output from the `FrameToPixels` object.

```
measure = visionhdl.MeasureTiming;
```

Serialize the input frame.

```
[pixInVec,ctrlInVec] = frm2pix(frmIn);
```

Some parameters require measurements between frames, so you must simulate at least two frames before using the results. Because you serialized only one input frame, process that frame twice to measure all parameters correctly.

```
for f = 1:2
    for p = 1:numPixPerFrm
        [activePixels,activeLines,totalPixels,totalLines, ...
            horizBlank,vertBlank] = measure(ctrlInVec(p));
    end
    fprintf('\nFrame %d:\n',f)
    fprintf('activePixels: %f\n',activePixels)
    fprintf('activeLines: %f\n',activeLines)
    fprintf('totalPixels: %f\n',totalPixels)
    fprintf('totalLines: %f\n',totalLines)
```

```
        fprintf('horizBlank: %f\n',horizBlank)
        fprintf('vertBlank: %f\n',vertBlank)
end
Frame 1:
activePixels: 32.000000
activeLines: 18.000000
totalPixels: 42.000000
totalLines: 22.880952
horizBlank: 10.000000
vertBlank: 4.880952
Frame 2:
activePixels: 32.000000
activeLines: 18.000000
totalPixels: 42.000000
totalLines: 26.000000
horizBlank: 10.000000
vertBlank: 8.000000
```

The measurements after the first frame are not accurate. However, after the second frame, the measurements match the parameters chosen in the `FrameToPixels` object.

Version History
Introduced in R2016b

See Also
Measure Timing | `visionhdl.FrameToPixels`

visionhdl.MedianFilter

Package: visionhdl

2-D median filtering

Description

The `visionhdl.MedianFilter` System object performs 2-D median filtering on a pixel stream. The object replaces each pixel value with the median value of the adjacent pixels.

To perform 2-D median filtering on a pixel stream:

1 Create the `visionhdl.MedianFilter` 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

```
MF = visionhdl.MedianFilter(Name,Value)
MF = visionhdl.MedianFilter(size,Name,Value)
```

Description

`MF = visionhdl.MedianFilter(Name,Value)` returns a median filter System object that performs 2-D median filtering of a pixel stream. Set properties using one or more name-value pairs. Enclose each property name in single quotes.

`MF = visionhdl.MedianFilter(size,Name,Value)` returns a median filter System object, with the `NeighborhoodSize` property set to `size` and additional options specified by one or more `Name,Value` pair arguments.

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.

NeighborhoodSize — Size of image region used to compute median
'3×3' (default) | '5×5' | '7×7'

Neighborhood size, specified as the dimensions in pixels.

LineBufferSize — Size of line buffer
2048 (default) | integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates *N – 1*-by-`LineBufferSize` memory locations to store the pixels used to compute the median value. *N* is the number of lines in the square region specified in `NeighborhoodSize`.

PaddingMethod — Method for padding boundary of input image
'Symmetric' (default) | 'Constant' | 'Replicate' | 'Reflection' | 'None'

Methods for padding the boundary of the input image, specified as one of these values.

- 'Constant' — Interpret pixels outside the image frame as having a constant value.
- 'Replicate' — Repeat the value of pixels at the edge of the image.
- 'Symmetric' — Set the value of the padding pixels to mirror the edge of the image.
- 'Reflection' — Set the value of the padding pixels to reflect around the pixel at the edge of the image.
- 'None' — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the *n*/2 pixels around the edge of the frame for downstream operations. *n* is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

For more information about these methods, see "Edge Padding".

You cannot use a neighborhood size of '7×7' when you set `PaddingMethod` to 'None'.

PaddingValue — Value used to pad boundary of input image
0 (default) | integer

Value used to pad the boundary of the input image, specified as an integer. The object casts this value to the same data type as the input pixel.

Dependencies

This parameter applies when you set `PaddingMethod` to 'Constant'.

Usage

Syntax

[pixelout,ctrlout] = MF(pixelin,ctrlin)

Description

[pixelout,ctrlout] = MF(pixelin,ctrlin) returns the filtered pixel value, `pixelout`, and accompanying control signals, resulting from calculating the median of the neighborhood around

each input pixel, `pixelin`. Before filtering, the object pads the image edges according to the `PaddingMethod` property.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel stream
scalar

Single image pixel in a pixel stream, specified as a scalar value. Integer and fixed-point data types larger than 16 bits are not supported.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fi` | `logical` | `double` | `single`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel stream
scalar

Single image pixel in a pixel stream, returned as a scalar value. The data type of `pixelout` is the same as the data type of `pixelin`.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fi` | `logical` | `double` | `single`

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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

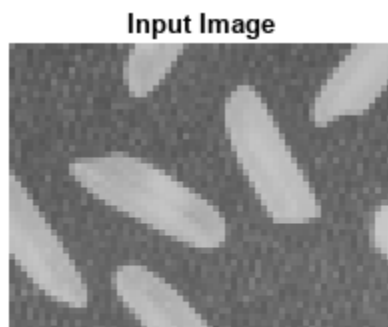| step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Median Filter Pixel Stream

This example implements a 5-by-5 median filter on a thumbnail image.

Load the source image from a file. Select a portion of the image matching the desired test size.

```
frmOrig = imread('rice.png');
frmActiveLines = 48;
frmActivePixels = 64;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Create a serializer object and specify the size of the inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',6, ...
      'FrontPorch',5);
```

Create a filter object and specify a neighbourhood.

```
 medianfilt = visionhdl.MedianFilter( ...
          'NeighborhoodSize','5x5');
```

Serialize the test image by calling the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the local median. Monitor control signals to determine latency of the object. The latency of a filter configuration depends on:

• The number of active pixels in a line.

• The size of the neighbourhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = medianfilt(pixIn(p),ctrlIn(p));
    if (ctrlOut(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
sprintf('object latency is %d cycles',foundValOut-foundValIn)

ans =
'object latency is 177 cycles'
```

Create a deserializer object with a format that matches the format of the serializer. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines,...
      'TotalPixelsPerLine',frmActivePixels+10);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
```

```
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image
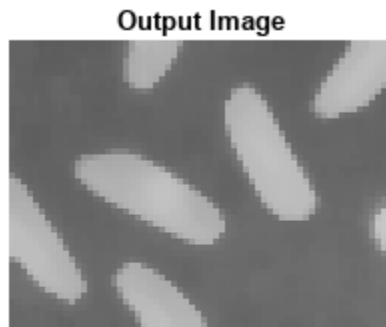
Algorithms

This object implements the algorithms described on the Median Filter block reference page.

Version History

Introduced in R2015a

R2021b: Reflection padding

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

R2019b: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also
Median Filter | `medfilt2` | `visionhdl.FrameToPixels`

visionhdl.Opening

Package: visionhdl

Morphologically open binary pixel stream

Description

The visionhdl.Opening System object morphologically opens a binary pixel stream. This operation morphologically erodes and then morphologically dilates each pixel by using the same neighborhood for both calculations. The object operates on a stream of binary intensity values.

To morphologically open a binary pixel stream:

1   Create the visionhdl.Opening 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

```
opener = visionhdl.Opening
opener = visionhdl.Opening(Name,Value)
```

Description

opener = visionhdl.Opening creates a System object that morphologically opens a binary pixel stream.

opener = visionhdl.Opening(Name,Value) sets properties using one or more name-value arguments. For example, 'Neighborhood',getnhood(strel('disk',4)) specifies a 4-by-4 disk-pattern neighborhood.

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.

Neighborhood — Pixel neighborhood
ones(3,3) (default) | vector or matrix of binary values

Pixel neighborhood, specified as a vector or matrix of binary values.

The object supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the Neighborhood property to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (*n* – 1)-by-LineBufferSize memory locations to store the pixels, where *n* is the number of lines in the Neighborhood property value.

PaddingMethod — Method for padding boundary of input image
'Constant' (default) | 'None'

Method for padding the boundary of the input image, specified as one of these values.

- 'Constant' — The object pads the image with ones for the erosion operation and with zeros for the dilation operation. These values prevent opening at the boundaries of the active frame.

- 'None' — Exclude padding logic. The object does not set the pixels outside the image frame to any particular value. This option reduces the hardware resources that are used by the object and reduces the blanking that is required between frames. However, this option affects the accuracy of the output pixels at the edges of the frame. To maintain pixel stream timing, the output frame is the same size as the input frame. To avoid using pixels calculated from undefined padding values, mask off the *n*/2 pixels around the edge of the frame for downstream operations. *n* is the size of the operation kernel. For more details, see "Increase Throughput by Omiting Padding".

For more information about these methods, see "Edge Padding".

Usage

Syntax

[pixelout,ctrlout] = opener(pixelin,ctrlin)

Description

[pixelout,ctrlout] = opener(pixelin,ctrlin) returns the next binary pixel value, pixelout, resulting from morphologically opening the neighborhood around each input binary pixel, pixelin.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the visionhdl.FrameToPixels object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel
`0` or `false` | `1` or `true`

Input pixel, specified as a `logical` value of `0`(`false`) or `1` (`true`).

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Data Types: `logical`

ctrlin — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

pixelout — Output pixel
`0` | `1`

Output pixel that is transformed by a morphological operation, returned as a `logical` value.

Data Types: `logical`

ctrlout — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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)
```

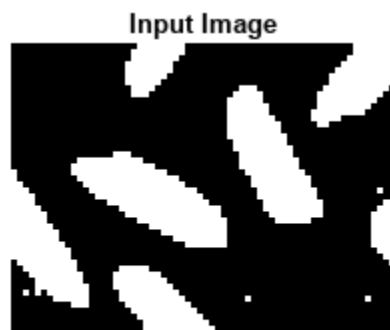## Common to All System Objects

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

Morphologically Open an Image

Load a source image from a file. Select a portion of the image that matches the desired test size. This source image contains pixel intensity values of `uint8` data type. Apply a threshold to convert the pixel data to binary values.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
frmInput = frmInput>128;
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Input Image

Create a serializer System object™ and define inactive pixel regions. Set the number of inactive pixels following each active line to at least double the horizontal size of the neighborhood. Set the number of lines following each frame to at least double the vertical size of the neighborhood.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',3, ...
      'FrontPorch',10);
```

Create a morphological open System object.

```
 opener = visionhdl.Opening(...
          'Neighborhood',ones(5,5));
```

Serialize the test image by calling the serializer object. `pixin` is a vector of intensity values. `ctrlin` is a vector of control signal structures.

```
[pixin,ctrlin] = frm2pix(frmInput);
```

Prepare to process the pixel stream by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixout = false(numPixelsPerFrame,1);
ctrlout  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlin(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixout(p),ctrlout(p)] = opener(pixin(p),ctrlin(p));
    if (ctrlout(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
objLatency_cycles= foundValOut - foundValIn
```

```
objLatency_cycles = 368
```

Create a deserializer System object with a format that matches the serializer format. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20);
[frmOutput,frmValid] = pix2frm(pixout,ctrlout);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image
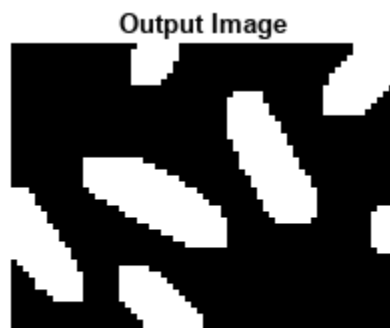
Algorithms

This object implements the algorithms described on the Opening block reference page.

Version History

Introduced in R2015a

R2020a: Option to omit padding

You can now configure the object to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the object and reduces the blanking interval required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the `PaddingMethod` property to `'None'`.

R2020a: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Objects
visionhdl.Erosion | visionhdl.Dilation | visionhdl.Closing | visionhdl.FrameToPixels

Blocks
Opening

Functions
imopen

visionhdl.GrayscaleOpening

Package: visionhdl

Morphologically open grayscale pixel stream

Description

The `visionhdl.GrayscaleOpening` System object morphologically opens a grayscale pixel stream. This operation morphologically erodes, and then morphologically dilates each pixel, by using the same neighborhood for both calculations.

The object operates on a stream of pixel intensity values. You can specify a neighborhood, or structuring element, of up to 32-by-32 pixels. For line, square, or rectangle structuring elements that are more than 8 pixels wide, the object uses the Van Herk algorithm to find the maximum and minimum. For structuring elements that are less than 8 pixels wide, or that contain zero elements, the object implements a pipelined comparison tree to find the maximum and minimum.

To morphologically open a grayscale pixel stream:

1   Create the `visionhdl.GrayscaleOpening` 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

```
opener = visionhdl.GreyOpening
opener = visionhdl.GrayOpening(Name,Value)
```

Description

`opener = visionhdl.GreyOpening` creates a System object that morphologically opens a grayscale pixel stream.

`opener = visionhdl.GrayOpening(Name,Value)` sets properties using one or more name-value arguments. For example, `'Neighborhood',getnhood(strel('disk',4))` specifies a 4-by-4 disk-pattern neighborhood.

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.

Neighborhood — Pixel neighborhood
ones(3,3) (default) | vector or matrix of binary values

Pixel neighborhood, specified as a vector or matrix of binary values.

The object supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, set the Neighborhood property to getnhood(strel(*shape*)), where *shape* is specified by the input arguments to the strel function. The minimum neighborhood size is a 2-by-2 matrix, or a 2-element column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

LineBufferSize — Size of line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a positive integer. Choose a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the buffer uses the next largest power of two.

The object allocates (n – 1)-by-LineBufferSize memory locations to store the pixels, where n is the number of lines in the Neighborhood property value.

Usage

Syntax

[pixelout,ctrlout] = opener(pixelin,ctrlin)

Description

[pixelout,ctrlout] = opener(pixelin,ctrlin) returns the next pixel intensity value, pixelout, resulting from morphologically opening the neighborhood around each input pixel intensity value, pixelin.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the visionhdl.FrameToPixels object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

pixelin — Input pixel
unsigned integer

Input pixel, specified as an unsigned integer.

The software supports double and single data types for simulation, but not for HDL code generation.

Data Types: single | double | uint8 | uint16 | uint32 | uint64 | fixdt(0,N,0)

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelout` — Output pixel
unsigned integer

Output pixel that is transformed by a morphological operation, returned as an unsigned integer.

The data type is the same as the data type of `pixelin`.

`ctrlout` — Control signals accompanying output pixel stream
`pixelcontrol` structure

Control signals accompanying the output pixel stream, returned as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

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

Morphologically Open Grayscale Image

Load a source image from a file. Select a portion of the image that matches the desired test size.

```
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```

Input Image

Create a serializer System object™ and define the inactive pixel regions. Set the number of inactive pixels following each active line to at least double the horizontal size of the neighborhood. Set the number of lines following each frame to at least double the vertical size of the neighborhood.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20, ...
      'TotalVideoLines',frmActiveLines+20, ...
      'StartingActiveLine',3, ...
      'FrontPorch',10);
```

Create a morphological open object.

```
opener = visionhdl.GrayscaleOpening(...
          'Neighborhood',ones(2,7));
```

Serialize the test image by calling the serializer object. `pixin` is a vector of intensity values. `ctrlin` is a vector of control signal structures.

```
[pixin,ctrlin] = frm2pix(frmInput);
```

Prepare to process the pixel stream by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixout = uint8(zeros(numPixelsPerFrame,1));
ctrlout = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlin(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixout(p),ctrlout(p)] = opener(pixin(p),ctrlin(p));
```

```matlab
        if (ctrlout(p).valid && foundValOut==0)
            foundValOut = p;
        end
    end
    sprintf('object latency is %d cycles',foundValOut - foundValIn)
```
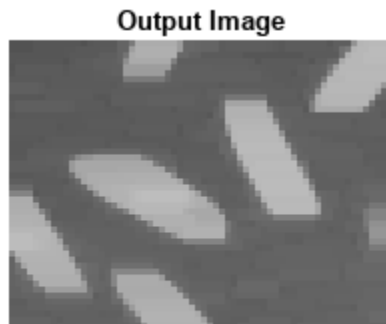
```
ans =
'object latency is 216 cycles'
```

Create a deserializer System object with a format that matches the serializer format. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```matlab
pix2frm = visionhdl.PixelsToFrame(...
        'NumComponents',1, ...
        'VideoFormat','custom', ...
        'ActivePixelsPerLine',frmActivePixels, ...
        'ActiveVideoLines',frmActiveLines, ...
        'TotalPixelsPerLine',frmActivePixels+20);
[frmOutput,frmValid] = pix2frm(pixout,ctrlout);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image

Algorithms

This object implements the algorithms described on the Grayscale Opening block reference page.

Version History
Introduced in R2016a

R2018b: Improved line buffer
Behavior changed in R2018b

The internal line buffer in this object now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding

logic and native support for kernel sizes with an even number of lines. This change affects the `visionhdl.LineBuffer` object and objects that use an internal line buffer.

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your designs. A best practice is to synchronize parallel paths in your designs by using the pixel stream control signals rather than by inserting a specific number of delays.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Objects
visionhdl.GrayscaleErosion | visionhdl.GrayscaleDilation | visionhdl.GrayscaleClosing | visionhdl.FrameToPixels

Blocks
Grayscale Opening

Functions
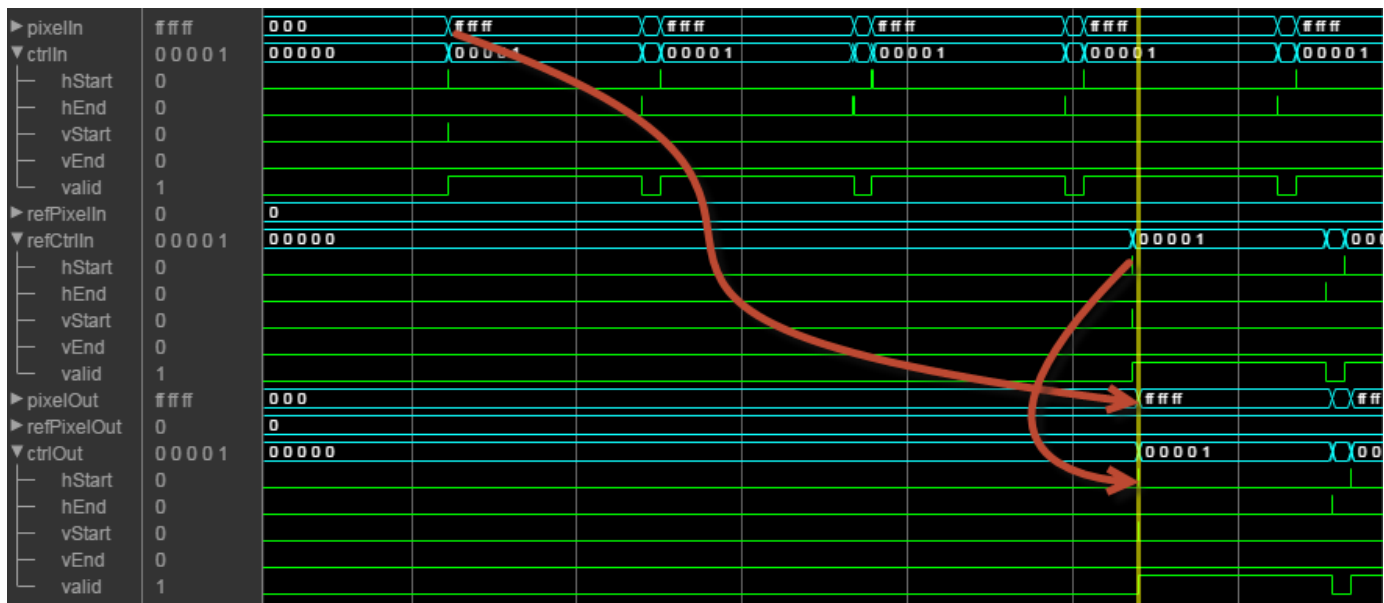imopen

visionhdl.PixelStreamAligner

Package: visionhdl

Align two streams of pixel data

Description

The `visionhdl.PixelStreamAligner` System object synchronizes two pixel streams by delaying one stream to match the timing of a reference stream. Many Vision HDL Toolbox algorithms delay the pixel stream, and the amount of delay can change as you adjust algorithm parameters. You can use this object to align streams for overlaying, comparing, or combining two streams such as in a Gaussian blur operation. Use the delayed stream as the `refpixel` and `refctrl` arguments. Use the earlier stream as the `pixelin` and `ctrlin` arguments.

This waveform diagram shows the input streams, `pixelin` and `refpixel`, and their associated control signals. The reference input frame starts later than the `pixelin` frame. The output signals show that the object delays `pixelin` to match the reference stream, and that both output streams share control signals. The waveform shows the short latency between the input `ctrl` and the output `ctrl`. In this simulation, to accommodate the delay of four lines between the input streams, the `MaxNumberofLines` property must be set to at least 4.

For details on the pixel control bus and the dimensions of a video frame, see "Streaming Pixel Interface".

To align two streams of pixel data:

1   Create the `visionhdl.PixelStreamAligner` 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

```
aligner = visionhdl.PixelStreamAligner(Name,Value)
```

Description

`aligner = visionhdl.PixelStreamAligner(Name,Value)` creates a System object that synchronizes a pixel stream with a reference pixel stream. Set properties using one or more name-value pairs. Enclose each property name in single quotes. For example, `'MaxNumberOfLines',16` sets the buffer depth that accommodates the timing offset between the two input streams.

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.

LineBufferSize — Size of the line memory buffer
2048 (default) | positive integer

Size of the line memory buffer, specified as a power of two that accommodates the number of active pixels in a horizontal line. If you specify a value that is not a power of two, the object uses the next largest power of two. The object implements a circular buffer of 2^M locations, where M is `MaxNumberofLines` + $\log_2$(`LineBufferSize`).

MaxNumberOfLines — Buffer depth that accommodates the timing offset between input streams
10 (default) | positive integer

Buffer depth that accommodates the timing offset between input streams, specified as a positive integer. The object implements a circular buffer of 2^M locations, where M is `MaxNumberofLines` + $\log_2$(`LineBufferSize`), and a line address buffer of `MaxNumberofLines` locations. The circular memory stores the earlier input lines until the reference control signals arrive. The line address buffer stores the address of the start of each line. When the reference control signals arrive, the object uses the stored address to read and send the delayed line. This property must accommodate the difference in timing between the two input streams, including the internal latency before the object reads the first line. During simulation, the object warns when an overflow occurs. To avoid the overflow condition, increase `MaxNumberofLines`. The delay between streams cannot exceed an entire frame.

Usage

Syntax

```
[pixelout,refout,ctrlout] = aligner(pixelin,ctrlin,refpixel,refctrl)
```

Description

`[pixelout,refout,ctrlout] = aligner(pixelin,ctrlin,refpixel,refctrl)` synchronizes a pixel stream to a reference stream, `refpixel` and `refctrl`, by delaying the first input, `pixelin`, to align with the reference input. The resulting aligned pixel streams, `pixelout` and `refout`, share the control signals, `ctrlout`. You can use this object to align streams for overlay or comparison.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel stream
scalar | row vector

Input pixel stream, specified as a scalar intensity value or a row vector of three values representing one pixel in R'G'B' or Y'CbCr color space. Because the object delays this pixel stream to match the control signals of the reference stream, `refpixel pixelin` must be the earlier of the two streams.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `fi` | `uint` | `int` | `logical` | `double` | `single`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

`refpixel` — Input reference pixel stream
scalar | vector

Input pixel, specified as a scalar intensity value or a row vector of three values representing one pixel in R'G'B' or Y'CbCr colorspace. Because the object delays the `pixelin` input stream to match the reference control signals, `refpixel` must be the later of the two streams. The reference data and its control signals pass through the object with a small delay.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `fi` | `uint` | `int` | `logical` | `double` | `single`

`refctrl` — Reference pixel stream control signals
structure

Reference pixel stream control signals, specified as a structure containing five `logical` signals. The object uses these control signals for the aligned output stream. For more details, see "Pixel Control Structure".

Output Arguments

`pixelout` — Output pixel stream
scalar | row vector

Output pixel, returned as a scalar intensity value or a row vector of three values representing one pixel in R'G'B' or Y'CbCr color space.

The data type is the same as the data type of `pixelin`.

Data Types: `fi` | `uint` | `int` | `logical` | `double` | `single`

`refout` — Output reference pixel stream
scalar | row vector

Output reference pixel, returned as a scalar intensity value or a row vector of three values representing one pixel in R'G'B' or Y'CbCr color space.

The data type is the same as the data type of `refpixel`.

Data Types: `fi` | `uint` | `int` | `logical` | `double` | `single`

`ctrlout` — Pixel stream control signals
structure

Pixel stream control signals for both output streams, returned as a structure containing five `logical` signals. For more details, see "Pixel Control Structure". These signals are a delayed version of the `refctrl` input.

Data Types: `struct`

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

Align Pixel Streams for HDL Generation

Overlay a processed video stream on the input stream.

Prepare a test image by selecting a portion of an image file.

```
frmActivePixels = 64;
frmActiveLines = 48;
frmOrig = imread('rice.png');
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```


Input Image
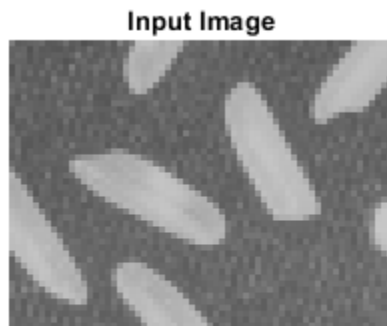
Create a serializer and specify the size of the inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',6, ...
      'FrontPorch',5);
```

Serialize the test image. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures. Preallocate vectors for the output signals.

```
[pixIn,ctrlIn] = frm2pix(frmInput);

[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
overlayOut = zeros(numPixelsPerFrame,1,'uint8');
```

Write a function that creates and calls the System objects to detect edges and align the edge data with the original pixel data. The edge results are delayed by the latency of the

`visionhdl.EdgeDetector` object. The associated control signals become the reference for the aligned stream. You can generate HDL from this function.

```
function [pixelOut,ctrlOut] = EdgeDetectandOverlay(pixelIn,ctrlIn)
% EdgeDetectandOverlay
% Detect edges in an input stream, and overlay the edge data onto the
% original stream.
% pixelIn and ctrlIn are a scalar pixel and its associated pixelcontrol
% structure, respectively.
% You can generate HDL code from this function.

  persistent align
  if isempty(align)
    align = visionhdl.PixelStreamAligner;
  end

  persistent find_edges
  if isempty(find_edges)
    find_edges = visionhdl.EdgeDetector;
  end

  [edgeOut,edgeCtrl] = find_edges(pixelIn,ctrlIn);
  [origOut,alignedEdgeOut,ctrlOut] = align(pixelIn,ctrlIn,edgeOut,edgeCtrl);
  if (alignedEdgeOut)
      pixelOut = uint8(0); % Set edge pixels to black
  else
      pixelOut = origOut;
  end
end
```
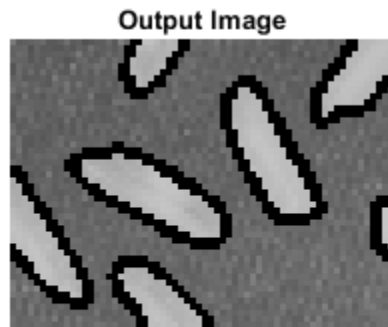
For each pixel in the frame, call the function to search for edges and align the edge data with the input stream.

```
for p = 1:numPixelsPerFrame
    [overlayOut(p),ctrlOut(p)] = EdgeDetectandOverlay(pixIn(p),ctrlIn(p));
end
```

Create a deserializer object with a format matching that of the serializer. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10);
[frmOutput,frmValid] = pix2frm(overlayOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```

Output Image

Version History

Introduced in R2017a

R2022b: Two pixels-per-clock streaming

The object now supports multipixel streams that have 2 pixels per clock cycle.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2021a: Multipixel-multicomponent streaming

The object now supports streams that are both multipixel and multicomponent. The object supports input and output matrices of *NumPixels*-by-*NumComponents* pixels, where *NumPixels* is 4 or 8. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the matrix.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2020b: Multipixel streaming

The object now supports multipixel streams. For multipixel streaming, the object supports input and output column vectors of 4 or 8 pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
Pixel Stream Aligner

Objects
visionhdl.FrameToPixels

visionhdl.PixelsToFrame

Package: visionhdl

Convert pixel stream to frame-based video

Description

The `visionhdl.PixelsToFrame` System object converts a color or grayscale pixel stream and control structures into frame-based video. The control structure indicates the validity of each pixel and its location in the frame. The pixel stream format can include padding pixels around the active frame. You can configure the frame and padding dimensions by selecting a common video format or by specifying custom dimensions. For details about the pixel stream format, see "Streaming Pixel Interface".

Use this object to convert the output of a function targeted for HDL code generation back to frames. The object itself does not support HDL code generation.

If your design converts frames to a pixel stream and later converts the stream back to frames, specify the same video format for the `FrameToPixels` object and the `PixelsToFrame` object.

To convert pixel stream to frame-based video:

1   Create the `visionhdl.PixelsToFrame` 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

```
P2F = visionhdl.PixelsToFrame(Name,Value)
```

Description

`P2F = visionhdl.PixelsToFrame(Name,Value)` returns a System object that converts a pixel stream to image frames. Set 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.

NumComponents — Number of values used to represent each pixel
1 (default) | 2 | 3 | 4

Number of values used to represent each pixel, specified as 1, 2, 3, or 4 components.

- For grayscale video, set this property to 1.
- For color video, for example, {R,G,B} or {Y,Cb,Cr}, set this property to 3.
- For color video with an alpha channel for transparency, set this property to 4.

The visionhdl.PixelsToFrame object expects a matrix of *P*-by-NumComponents values, where *P* is the total number of pixels in the padded frame.

Dependencies

When NumComponents is greater than 1, you must set the NumPixels property to 1.

NumPixels — Number of pixels streamed per time step
1 (default) | 2 | 4 | 8

Number of pixels transferred on the streaming interface for each cycle, specified as 1, 2, 4, or 8. To enable multipixel streaming and increase throughput for high-resolution or high-frame-rate video, set this property to 2, 4, or 8. The visionhdl.PixelsToFrame object expects a *P*-by-NumPixels matrix, where *P* is the total number of pixels in the padded frame.

Note You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Dependencies

When NumPixels is greater than 1, you must set the NumComponents property to 1.

VideoFormat — Dimensions of active and inactive regions in video frame
'1080p' (default) | '240p' | '480p' | '480pH' | '576p' | '720p' | '768p' | '1024p' | '1200p' | '2KCinema' | '4KUHDTV' | '8KUHDTV' | 'Custom'

Dimensions of the active region of a video frame. To select a predefined format, specify the VideoFormat property as one of the options in the first column of the table. For a custom format, set VideoFormat to 'Custom', and specify the dimensional properties as integers.

Video Format	Active Pixels Per Line	Active Video Lines
240p	320	240
480p	640	480
480pH	720	480
576p	720	576
720p	1280	720
768p	1024	768
1024p	1280	1024

Video Format	Active Pixels Per Line	Active Video Lines
1080p (default)	1920	1080
1200p	1600	1200
2KCinema	2048	1080
4KUHDTV	3840	2160
8KUHDTV	7680	4320
Custom	User-defined	User-defined

Usage

Syntax

`[frm,validOut] = P2F(pixels,ctrlIn)`

Description

`[frm,validOut] = P2F(pixels,ctrlIn)` converts a vector of pixel values representing a padded image, `pixels`, and an associated vector of control structures, `ctrlIn`, to an image matrix, `frm`. The control structure indicates the validity of each pixel and its location in the frame. The output image, `frm`, is valid if `validOut` is `true`.

For details about the pixel stream format, see "Streaming Pixel Interface".

Input Arguments

pixels — Input pixels
matrix

Pixel values, specified as a *P*-by-`NumComponents` matrix, or *P*-by-`NumPixels` matrix, where:

- *P* is the total number of pixels in the padded image, which is `TotalPixelsPerLine` × `TotalVideoLines`.
- `NumComponents` is the number of components used to express a single pixel.
- `NumPixels` is the number of pixels transferred on the streaming interface per cycle. When `NumPixels` is greater than 1, you must set `NumComponents` to 1.

Note You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Set the size of the padded image using the `VideoFormat` property. If the number of elements in `pixels` does not match the dimensions specified by `VideoFormat`, the object returns a warning.

Data Types: `uint` | `int` | `logical` | `fi` | `double` | `single`

ctrlIn — Pixel stream control signals
vector of structures

Control structures associated with the input pixels, specified as a *P*-by-1 vector. *P* is the total number of pixels in the padded image, which is `TotalPixelsPerLine` × `TotalVideoLines`. Each structure contains five control signals indicating the validity of the pixel and its location in the frame. For multipixel streaming, the control signals apply to each set of `NumPixels` values. See "Pixel Control Structure".

Output Arguments

`frm` — Output image
matrix

Output image, returned as an `ActiveVideoLines`-by-`ActivePixelsPerLine`-by-`NumComponents` matrix, where:

- `ActiveVideoLines` is the height of the active image.
- `ActivePixelsPerLine` is the width of the active image.
- `NumComponents` is the number of components used to express a single pixel.

The data type of the pixels is the same as the input `pixels`.

Data Types: uint | int | logical | fi | double | single

`validOut` — Indication of valid output
scalar logical

Frame status, returned as a logical value. When `validOut` is true, the frame is reassembled and ready for use.

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

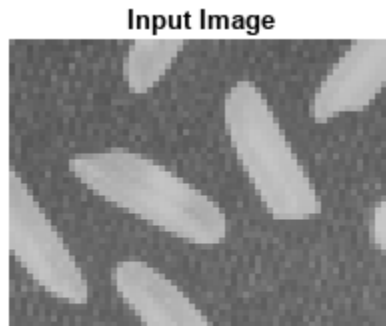### Convert Between Full-Frame and Pixel-Streaming Data

This example converts a custom-size grayscale image to a pixel stream. It uses the `visionhdl.LookupTable` object to obtain the negative image. Then it converts the pixel stream back to a full-frame image.

Load the source image from a file. Select a portion of the image matching the desired test size.

```matlab
frmOrig = imread('rice.png');
frmActivePixels = 64;
frmActiveLines = 48;
frmInput = frmOrig(1:frmActiveLines,1:frmActivePixels);
figure
imshow(frmInput,'InitialMagnification',300)
title 'Input Image'
```

Input Image

Create a serializer object and specify the size of the inactive pixel regions.

```matlab
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10, ...
      'TotalVideoLines',frmActiveLines+10, ...
      'StartingActiveLine',6, ...
      'FrontPorch',5);
```

Create a lookup table (LUT) object to generate the negative of the input image.

```matlab
tabledata = linspace(255,0,256);
inverter = visionhdl.LookupTable(tabledata);
```

Serialize the test image by calling the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```matlab
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

```matlab
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

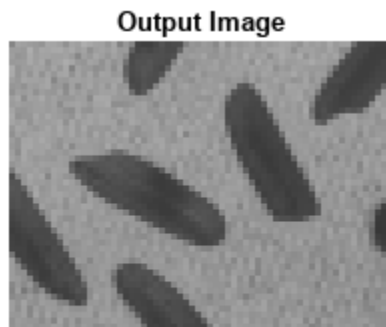For each pixel in the stream, look up the negative of the pixel value.

```matlab
for p = 1:numPixelsPerFrame
    [pixOut(p),ctrlOut(p)] = inverter(pixIn(p),ctrlIn(p));
end
```

Create a deserializer object with a format that matches the format of the serializer. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+10);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```


Output Image

Version History

Introduced in R2015a

R2022a: Change in default frame dimensions
Behavior changed in R2022a

The default values of the custom format have changed. In previous releases, the default dimensions matched the 1080p format. Starting in R2022a, the default values match the default values of the Frame To Pixels and Pixels To Frame blocks, which define a small custom frame as shown in this code.

```
a = visionhdl.FrameToPixels('VideoFormat','Custom')

a =

  visionhdl.FrameToPixels with properties:

          NumComponents: 1
              NumPixels: 1
            VideoFormat: 'Custom'
    ActivePixelsPerLine: 32
       ActiveVideoLines: 18
```

```
      TotalPixelsPerLine: 42
         TotalVideoLines: 28
      StartingActiveLine: 6
               FrontPorch: 5
```

This change means you might need to specify additional dimensions for custom formats where previously the default value resulted in a valid format.

The object now updates frame dimension properties when you select a standard format. This change means you can refer to the object properties to determine the dimensions being used by the format.

```
a = visionhdl.FrameToPixels('VideoFormat','480p')

a =

  visionhdl.FrameToPixels with properties:

    NumComponents: 1
        NumPixels: 1
      VideoFormat: '480p'

TPPL480 = a.TotalPixelsPerLine

ans =

   800
```

R2022a: Two pixels-per-clock streaming

The object now accepts multipixel streams that have 2 pixels per clock cycle.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2020a: Multipixel-multicomponent streaming

The object can now accept multipixel-multicomponent streams. The object accepts an input matrix of *NumPixels*-by-*NumComponents* pixels. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the matrix.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2019b: Multipixel streaming

The object can now accept pixel streams that have 4 or 8 pixels per clock cycle. You can set the `NumPixels` property to 4 or 8. With this setting, the object expects an input vector of 4 or 8 pixel values on each time step.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

See Also
`visionhdl.FrameToPixels` | Pixels To Frame

Topics
"Pixel-Streaming Design in MATLAB"
"Streaming Pixel Interface"

visionhdl.ROISelector

Package: `visionhdl`

Select region of interest (ROI) from pixel stream

Description

The `visionhdl.ROISelector` System object selects a portion of the active frame from a video stream. The total size of the frame remains the same. The output control signals indicate a new active region of the frame. This diagram shows the inactive pixel regions in blue and the requested output region outlined in orange.


You can specify a fixed size and location for the ROI, or you can select the frame location dynamically by using an input argument. You can select more than one region. Define each region by specifying its upper-left corner coordinates and dimensions. The object returns one set of pixels and control signals for each region you specify. The object sets the inactive pixels in the output frame to zero.

Regions are independent from each other, so they can overlap. If you specify a region that includes the edge of the active frame, the object returns only the active portion of the region. This diagram shows the output frames for three requested regions. The second output region does not include the inactive region above the frame.

The object also provides a mode for vertical reuse. In this mode, you must specify regions that have no vertical overlap and are aligned in columns. Each column of regions shares one output pixel

stream. The control signals define each region in the stream. This arrangement enables parallel processing of each column, and the reuse of downstream processing logic for each region in a column. To use this mode, set the `VerticalReuse` property to `true`. Use the `visionhdlframetoregions` function to divide a frame into tiled regions for vertical reuse.
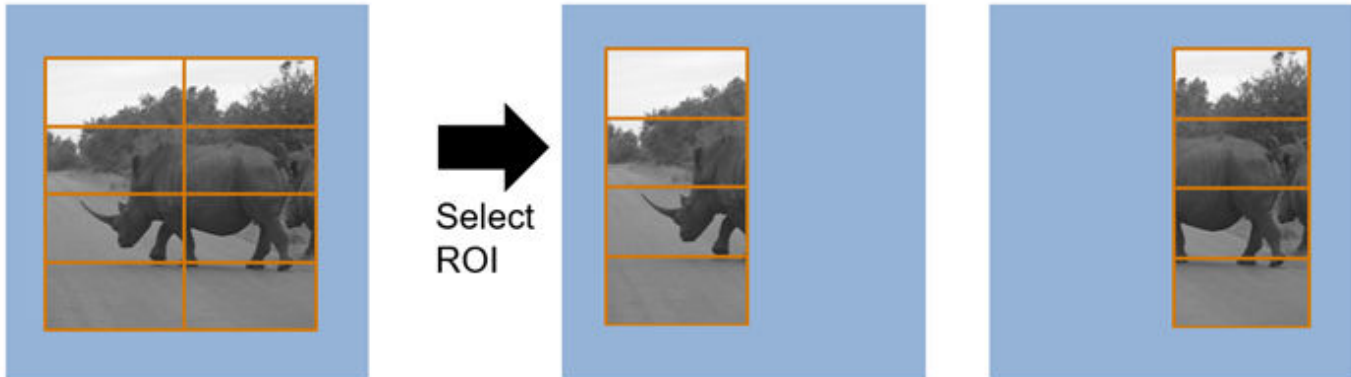
To select regions of interest (ROIs) from a pixel stream:

1   Create the `visionhdl.ROISelector` 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

`ROI = visionhdl.ROISelector(Name,Value)`

Description

`ROI = visionhdl.ROISelector(Name,Value)` creates a System object that selects regions of the active frame from an input stream. Set properties using one or more name-value pairs. Enclose each property name in single quotes. For example,.`'Regions',[1 1 32 24]` configures the object to select a 32-by-24 pixel region in the top-left corner of the frame.

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.

VerticalReuse — Option to return one pixel stream for each column of regions
`false` (default) | `true`

Option to return one pixel stream for each column of regions, specified as `false` or `true`. When you set this property to `true`, you must specify regions that have no vertical overlap and that are aligned in columns. The object has one output pixel stream for each column of regions. This arrangement enables parallel processing of each column and the reuse of downstream processing logic for each region in a column. Use the `visionhdlframetoregions` function to divide a frame into tiled regions for vertical reuse.

Dependencies

To enable this property, set the `RegionsSource` property to `'Property'`.

RegionsSource — Location of output region definitions
`'Property'` (default) | `'Input port'`

Location of the output region definitions, specified as one of these values:

`'Property'` — Specify the regions by using the `Regions` property.

`'Input port'` — Specify the regions by using input arguments to the object. Each input argument is a four-element row vector corresponding to one region. The object captures the value of the `region` input arguments when it receives `vStart` set to `true` in the input control structure.

Dependencies

To enable this property, set the `VerticalReuse` property to `false`.

### Regions — Rectangular ROIs to select from input frame
[100 100 50 50] (default) | *R*-by-4 matrix

Rectangular regions of interest to select from the input frame, specified as an *R*-by-4 matrix.

R is the number of regions. The four elements that define each region are the top-left starting coordinates and dimensions of the region and must be of the form [*hPos vPos hSize vSize*]. The coordinates count from the upper-left corner of the active frame, defined as [1,1]. *hSize* must be greater than 1.

When you set the `VerticalReuse` property to `false`, the regions are independent of each other, so they can overlap. *R* must not be greater than 16.

When you set the `VerticalReuse` property to `true`, the regions in this matrix must have no vertical overlap and be aligned in columns. This diagram shows two examples of invalid regions outlined with a blue dashed line. The region on the left overlaps another region vertically. The two regions at the bottom do not align with the other regions in their columns. The regions do not have to cover the entire frame. Pixels outside of the tiled regions are marked as inactive pixels. The number of columns must not be greater than 16.
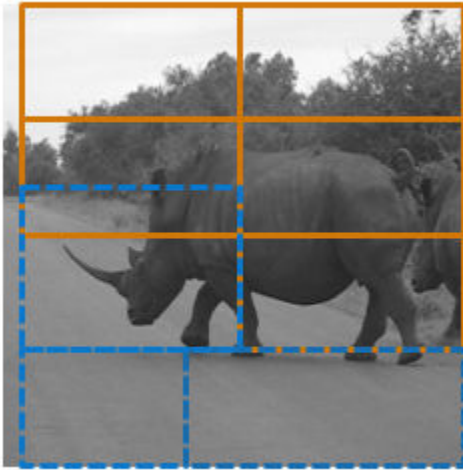
Dependencies

To enable this parameter, set the `RegionsSource` property to `'Property'`.

NumberOfRegions — Number of region input arguments
1 (default) | integer in the range [1, 16]

Number of `region` input arguments to the object, specified as an integer in the range [1, 16].

Dependencies

To enable this parameter, set the `RegionsSource` property to `'Input port'`.

Usage

Syntax

```
[pixel1,ctrl1] = ROI(pixelin,ctrlin)
[pixel1,ctrl1,...,pixelR,ctrlR] = ROI(pixelin,ctrlin)
[pixel1,ctrl1,...,pixelR,ctrlR] = ROI(pixelin,ctrlin,region1,...,regionR)
```

Description

`[pixel1,ctrl1] = ROI(pixelin,ctrlin)` returns the next pixel value, `pixel1`, and control signals, `ctrl1`, resulting from masking the active image frame into a single new region. Define the region by setting the `Regions` property to a four-element row vector, [*hPos vPos hSize vSize*].

`[pixel1,ctrl1,...,pixelR,ctrlR] = ROI(pixelin,ctrlin)` returns the next pixel values, `pixel1,...,pixelR`, and control signals, `ctrl1,...,ctrlR`, of each stream resulting from masking the active image frame. When you set the `VerticalReuse` property to `false`, each output stream contains one active region, and the number of streams is the `NumberOfRegions` property value or the number of rows in the `Regions` matrix. When you set the `VerticalReuse` property to `true`, each output stream contains a set of vertically-aligned regions. In both cases, define the regions by setting the `Regions` property to an *R*-by-4 matrix of ROI coordinates.

`[pixel1,ctrl1,...,pixelR,ctrlR] = ROI(pixelin,ctrlin,region1,...,regionR)` returns the next pixel values of each stream, `pixel1,...,pixelR`, resulting from masking the

active image frame into 1 to *R* new regions, as directed by the `region1,...,regionR` arguments. Each `region` input is a four-element row vector of ROI coordinates. Use this syntax when you set the `RegionsSource` property to `'Input Port'`, and the `NumberOfRegions` property to *R*.

This object uses a streaming pixel interface with a structure for frame control signals. This interface enables the object to operate independently of image size and format and to connect with other Vision HDL Toolbox objects. The object accepts and returns a scalar pixel value and control signals as a structure containing five signals. The control signals indicate the validity of each pixel and its location in the frame. To convert a pixel matrix into a pixel stream and control signals, use the `visionhdl.FrameToPixels` object. For a description of the interface, see "Streaming Pixel Interface".

Input Arguments

`pixelin` — Input pixel stream
scalar

Single image pixel in a pixel stream, specified as a scalar value representing intensity.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

The software supports `double` and `single` data types for simulation, but not for HDL code generation.

Data Types: `uint` | `int` | `fi` | `logical` | `double` | `single`

`ctrlin` — Control signals accompanying input pixel stream
`pixelcontrol` structure

Control signals accompanying the input pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and its location in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

`region1,...,regionR` — Region of interest
four-element row vector

Region of interest, specified as a row vector of four positive integers that define the coordinates of the top-left corner and dimensions of each desired output frame. The vector must have the form [*hPos vPos hSize vSize*]. Specify *R* region arguments, where *R* is the `NumberOfRegions` property value.

Dependencies

To enable this argument, set the `VerticalReuse` property to `false` and set the `RegionsSource` property to `'Input Port'`.

Data Types: `int` | `uint`

Output Arguments

`pixel1,...,pixelR` — Output pixel stream
scalar

Output pixel, returned as a scalar value. The output data type is the same data type as the input `pixelin` argument. By default, the number of output pixel arguments is the `NumberOfRegions` property value, or the number of rows in the `Regions` property matrix. When you set `VerticalReuse` to `true`, the object has one output pixel argument for each column of vertically-aligned regions.

Data Types: `uint` | `int` | `fi` | `logical` | `double` | `single`

`ctrl1,...,ctrlR` — Pixel stream control signals
structure

Pixel stream control signals indicating the validity of each output pixel and its location within the frame, returned as a structure of five `logical` signals. For more details, see "Pixel Control Structure".

By default, the number of output control arguments is the `NumberOfRegions` property value, or the number of rows in the `Regions` property matrix. When you set `VerticalReuse` to `true`, the object has one output control argument for each column of vertically-aligned regions.

Data Types: `struct`

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

Select Fixed ROI from Input Frame

Load a source image from a file, and then display the image.

```
frmOrig = imread('coins.png');
[frmActiveLines,frmActivePixels] = size(frmOrig);
imshow(frmOrig)
title 'Input Image'
```

Input Image

Create a serializer object and define inactive pixel regions.

```
frm2pix = visionhdl.FrameToPixels( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',frmActivePixels, ...
      'ActiveVideoLines',frmActiveLines, ...
      'TotalPixelsPerLine',frmActivePixels+20, ...
      'TotalVideoLines',frmActiveLines+20, ...
      'StartingActiveLine',3, ...
      'FrontPorch',10);
```

Create an object to select a region of interest. Define a rectangular region by the coordinates of its top-left corner and dimensions.

```
hPos = 80;
vPos = 60;
hSize = 65;
vSize = 50;
roicoin = visionhdl.ROISelector('Regions',[hPos vPos hSize vSize])

roicoin =
  visionhdl.ROISelector with properties:

    VerticalReuse: false
    RegionsSource: 'Property'
          Regions: [80 60 65 50]
```

Serialize the test image by calling the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

```
[pixIn,ctrlIn] = frm2pix(frmOrig);
```

Prepare to process pixels by preallocating output vectors. The output frame is the same size as the input frame, but the control signals indicate a different active region.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = uint8(zeros(numPixelsPerFrame,1));
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

For each pixel in the padded frame, apply the region mask.

```
for p = 1:numPixelsPerFrame
    [pixOut(p),ctrlOut(p)] = roicoin(pixIn(p),ctrlIn(p));
end
```

Create a deserializer object with a format that matches the new region. Convert the pixel stream to an image frame by calling the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame( ...
      'NumComponents',1, ...
      'VideoFormat','custom', ...
      'ActivePixelsPerLine',hSize, ...
      'ActiveVideoLines',vSize, ...
      'TotalPixelsPerLine',hSize+20);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput)
    title 'Output Image'
end
```


Output Image
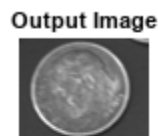
Algorithms

The generated HDL code for the `visionhdl.ROISelector` System object uses two 32-bit counters. The object does not use additional counters for additional regions.

Latency

The object has a latency of three cycles.

Version History
Introduced in R2016a

R2021a: Multipixel-multicomponent streaming

The block now supports streams that are both multipixel and multicomponent. The object supports input and output matrices of *NumPixels*-by-*NumComponents* pixels, where *NumPixels* is 2, 4, or 8. The `ctrl` argument remains scalar, and the control signals in the `pixelcontrol` structure apply to all pixels in the matrix.

R2021a: Resource sharing

The object now provides an option to share hardware resources when you select vertically aligned regions. Regions in the same column share the same `pixelcontrol` structure output.

Set the `VerticalReuse` property to `true`, and provide a set of regions that are aligned in columns and do not overlap vertically within each column. You can specify up to 1024 regions per column. To divide a frame into tiled regions that are compatible with vertical reuse, use the `visionhdlframetoregions` function.

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. For more information about acceleration, see "Accelerate Pixel-Streaming Designs Using MATLAB Coder". For more information about DPI component generation, see "Considerations for DPI Component Generation with MATLAB" (HDL Verifier).

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

To generate HDL code from Vision HDL Toolbox System objects, see "Design Hardware-Targeted Image Filters in MATLAB".

See Also

Blocks
ROI Selector

Objects
visionhdl.FrameToPixels

Functions
visionhdlframetoregions

Functions

getparamfromfrm2pix

Get frame format parameters

Syntax

```
[activePixelsPerLine,activeLines,numPixelsPerFrame] = getparamfromfrm2pix(
frm2pix)
```

Description

[activePixelsPerLine,activeLines,numPixelsPerFrame] = getparamfromfrm2pix(
frm2pix) returns video format parameters from a `visionhdl.FrameToPixels` System object.

Examples

Configure Pixel Stream Format

When you choose a standard video format for `visionhdl.FrameToPixels`, the object computes the frame dimensions. To access these values, call the `getparamfromfrm2pix` function.

```
frm2pix = visionhdl.FrameToPixels(...
      'NumComponents',1,...
      'VideoFormat','240p');
[activePixels,activeLines,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix)

activePixels = 320

activeLines = 240

numPixelsPerFrame = 130248
```

Note that `numPixelsPerFrame` includes both active and inactive regions of the frame.

Input Arguments

`frm2pix` — Video serializer
`visionhdl.FrameToPixels` System object

The `visionhdl.FrameToPixels` object converts framed video to a stream of pixel values and control signals. It contains useful parameters regarding the video format.

Output Arguments

`activePixelsPerLine` — Number of pixels in a horizontal line of the active video frame
positive integer

Number of pixels in a horizontal line of the active video frame, returned as a positive integer.

For custom video formats, this value corresponds to the `ActivePixelsPerLine` property of the `frm2pix` object.

activeLines — Number of horizontal lines in the active video frame
positive integer

Number of horizontal lines in the active video frame, returned as a positive integer.

For custom video formats, this value corresponds to the `ActiveVideoLines` property of the `frm2pix` object.

numPixelsPerFrame — Number of active and inactive pixels in the video frame
positive integer

Number of active and inactive pixels in the video frame, returned as a positive integer.

For custom video formats, this value corresponds to the product of the `TotalVideoLines` and `TotalPixelsPerLine` properties of the `frm2pix` object.

Version History
Introduced in R2015a

See Also
Frame To Pixels | Pixels To Frame

Topics
"Streaming Pixel Interface"

pixelcontrolbus

Create pixel-streaming control bus instance

Syntax

```
pixelcontrolbus
```

Description

`pixelcontrolbus` declares a bus instance in the workspace. This instance is required for HDL code generation. Call this function before you generate HDL code from Vision HDL Toolbox blocks.

Examples

Declare Bus in Base Workspace

• In the `InitFcn` callback function of your Simulink model, include this line to declare a bus instance in the base workspace. If you create your model with the Vision HDL Toolbox model template, this is done for you.

```
evalin('base','pixelcontrolbus')
```

If you do not declare an instance of `pixelcontrolbus` in the base workspace, you might encounter this error when you generate HDL code in Simulink.

```
Cannot resolve variable 'pixelcontrol'
```

Version History
Introduced in R2015a

See Also
"Pixel Control Bus" | Frame To Pixels | Pixels To Frame

Topics
"Configure the Simulink Environment for HDL Video Processing"
"Streaming Pixel Interface"

pixelcontrolsignals

Extract signals from pixel-streaming control signal structure

Syntax

```
[hStart,hEnd,vStart,vEnd,valid] = pixelcontrolsignals(ctrl)
```
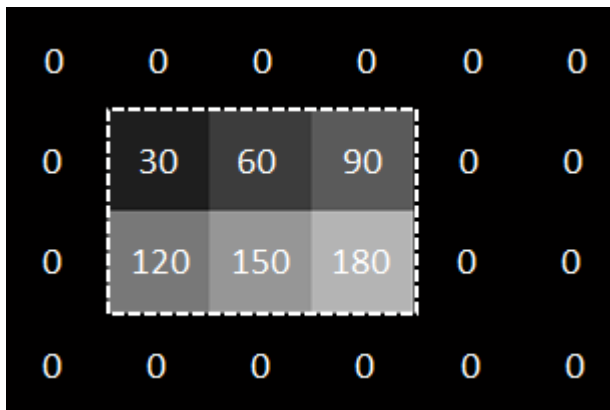
Description

`[hStart,hEnd,vStart,vEnd,valid] = pixelcontrolsignals(ctrl)` extracts five scalar `logical` control signals from a structure.

Examples

Create and Decompose pixelcontrol structures

If you integrate Vision HDL Toolbox designs with algorithms that use a different interface, you may need to create the structure manually, or manipulate the control signals outside of the structure.

Create a `pixelcontrol` structure by passing five control signal values to the `pixelcontrolstruct` function. The function arguments must be scalar values. These control signals may come from a camera or other video input source. The control signal vectors in this example describe a simple 2-by-3 pixel test image, surrounded by padding pixels.


```
hStart = [0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0];
vStart = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
hEnd   = [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0];
vEnd   = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0];
valid  = [0 0 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0];
pixel = uint8([0 0 0 0 0 0 0 30 60 90 0 0 0 120 150 180 0 0 0 0 0 0 0 0]);
[~,numPix] = size(pixel);
ctrlIn = repmat(pixelcontrolstruct,numPix,1);
for i = 1:numPix
    ctrlIn(i) = pixelcontrolstruct(hStart(i),vStart(i),...
                                   hEnd(i),vEnd(i),valid(i));
end
```

Each element of `ctrlIn` is a structure containing the five control signals.

```
ctrlIn(8)
```

```
ans = struct with fields:
    hStart: 1
      hEnd: 1
    vStart: 0
      vEnd: 0
     valid: 1
```

You can then pass this structure to a Vision HDL Toolbox System object™. This example uses the LookupTable object to invert each pixel.

```
tabledata = uint8(linspace(255,0,256));
inverter = visionhdl.LookupTable(tabledata);
pixelOut = zeros(numPix,1,'uint8');
ctrlOut = repmat(pixelcontrolstruct,numPix,1);

for i = 1:numPix
    [pixelOut(i),ctrlOut(i)] = inverter(pixel(i),ctrlIn(i));
end
```

If you need to use the control signals directly in downstream algorithms, you can flatten each structure into five `logical` control signal values by calling the `pixelcontrolsignals` function.

```
[hStartOut,vStartOut,hEndOut,vEndOut,validOut] = deal(false(numPix,1));
for i = 1:numPix
    [hStartOut(i),vStartOut(i),hEndOut(i),vEndOut(i),validOut(i)] = ...
        pixelcontrolsignals(ctrlOut(i));
end
```

Each output control signal is a vector of logical values that correspond with the `pixelOut` vector.

```
validOut'
```

```
ans = 1x24 logical array

   0   0   0   0   0   0   0   0   0   1   1   1   0   0   0   1   1   1   0   0   0   0   0   0
```

Input Arguments

`ctrl` — Pixel control signals
structure containing five `logical` values

Pixel control signals, specified as a structure containing five `logical` values.

The pixel control structure is a specific format used by Vision HDL Toolbox objects. See "Pixel Control Structure".

Output Arguments

`hStart` — Control signal indicating the first pixel in a horizontal line
logical

Control signal indicating the first pixel in a horizontal line, specified as a `logical` scalar.

hEnd — Control signal indicating the last pixel in a horizontal line
`logical`

Control signal indicating the last pixel in a horizontal line, specified as a `logical` scalar.

vStart — Control signal indicating the first pixel in the first (top) line
`logical`

Control signal indicating the first pixel in the first (top) line, specified as a `logical` scalar.

vEnd — Control signal indicating the last pixel in the last (bottom) line
`logical`

Control signal indicating the last pixel in the last (bottom) line, specified as a `logical` scalar.

valid — Control signal indicating the validity of the pixel
`logical`

Control signal indicating the validity of the pixel, specified as a `logical` scalar.

Version History
Introduced in R2015a

See Also
`pixelcontrolstruct` | `visionhdl.FrameToPixels` | `visionhdl.PixelsToFrame`

Topics
"Streaming Pixel Interface"

pixelcontrolstruct

Create pixel-streaming control signal structure

Syntax

```
ctrl = pixelcontrolstruct(hStart,hEnd,vStart,vEnd,valid)
```
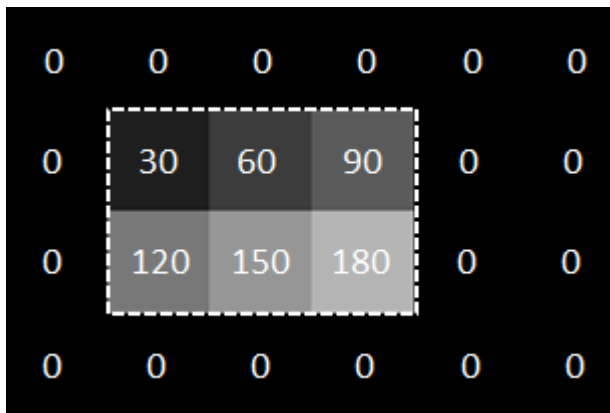
Description

`ctrl = pixelcontrolstruct(hStart,hEnd,vStart,vEnd,valid)` creates a structure containing the five control signals used by Vision HDL Toolbox objects. The input arguments must be five scalars of `logical` type. See "Pixel Control Structure".

Examples

Create and Decompose pixelcontrol structures

If you integrate Vision HDL Toolbox designs with algorithms that use a different interface, you may need to create the structure manually, or manipulate the control signals outside of the structure.

Create a `pixelcontrol` structure by passing five control signal values to the `pixelcontrolstruct` function. The function arguments must be scalar values. These control signals may come from a camera or other video input source. The control signal vectors in this example describe a simple 2-by-3 pixel test image, surrounded by padding pixels.


```
hStart = [0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0];
vStart = [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
hEnd   = [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0];
vEnd   = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0];
valid  = [0 0 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0];
pixel = uint8([0 0 0 0 0 0 0 30 60 90 0 0 0 120 150 180 0 0 0 0 0 0 0 0]);
[~,numPix] = size(pixel);
ctrlIn = repmat(pixelcontrolstruct,numPix,1);
for i = 1:numPix
    ctrlIn(i) = pixelcontrolstruct(hStart(i),vStart(i),...
```

```
                                            hEnd(i),vEnd(i),valid(i));
end
```

Each element of `ctrlIn` is a structure containing the five control signals.

```
ctrlIn(8)
```

```
ans = struct with fields:
    hStart: 1
      hEnd: 1
    vStart: 0
      vEnd: 0
     valid: 1
```

You can then pass this structure to a Vision HDL Toolbox System object™. This example uses the LookupTable object to invert each pixel.

```
tabledata = uint8(linspace(255,0,256));
inverter = visionhdl.LookupTable(tabledata);
pixelOut = zeros(numPix,1,'uint8');
ctrlOut = repmat(pixelcontrolstruct,numPix,1);

for i = 1:numPix
    [pixelOut(i),ctrlOut(i)] = inverter(pixel(i),ctrlIn(i));
end
```

If you need to use the control signals directly in downstream algorithms, you can flatten each structure into five `logical` control signal values by calling the `pixelcontrolsignals` function.

```
[hStartOut,vStartOut,hEndOut,vEndOut,validOut] = deal(false(numPix,1));
for i = 1:numPix
    [hStartOut(i),vStartOut(i),hEndOut(i),vEndOut(i),validOut(i)] = ...
        pixelcontrolsignals(ctrlOut(i));
end
```

Each output control signal is a vector of logical values that correspond with the `pixelOut` vector.

```
validOut'
```

```
ans = 1x24 logical array

   0   0   0   0   0   0   0   0   0   1   1   1   0   0   0   1   1   1   0   0   0   0   0   0
```

Input Arguments

hStart — Control signal indicating the first pixel in a horizontal line
logical

Control signal indicating the first pixel in a horizontal line, specified as a `logical` scalar.

hEnd — Control signal indicating the last pixel in a horizontal line
logical

Control signal indicating the last pixel in a horizontal line, specified as a `logical` scalar.

vStart — Control signal indicating the first pixel in the first (top) line
logical

Control signal indicating the first pixel in the first (top) line, specified as a logical scalar.

vEnd — Control signal indicating the last pixel in the last (bottom) line
logical

Control signal indicating the last pixel in the last (bottom) line, specified as a logical scalar.

valid — Control signal indicating the validity of the pixel
logical

Control signal indicating the validity of the pixel, specified as a logical scalar.

Output Arguments

ctrl — Pixel control signals
structure containing five logical values

Pixel control signals, specified as a structure containing five logical values.

The pixel control structure is a specific format used by Vision HDL Toolbox objects. See "Pixel Control Structure".

Version History
Introduced in R2015a

See Also
pixelcontrolsignals | visionhdl.FrameToPixels | visionhdl.PixelsToFrame

Topics
"Streaming Pixel Interface"

Simulink.sdi.compareRuns

Package: `Simulink.sdi`

Compare data in two simulation runs

Syntax

```
diffResult = Simulink.sdi.compareRuns(runID1,runID2)
diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name=Value)
```

Description

`diffResult = Simulink.sdi.compareRuns(runID1,runID2)` compares the data in the runs that correspond to `runID1` and `runID2` and returns the result in the `Simulink.sdi.DiffRunResult` object `diffResult`. For more information about the comparison algorithm, see "How the Simulation Data Inspector Compares Data" (Simulink).

`diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name=Value)` compares the simulation runs that correspond to `runID1` and `runID2` using the options specified by one or more name-value arguments. For more information about comparison options, see "How the Simulation Data Inspector Compares Data" (Simulink).

Examples

Compare Runs with Global Tolerance

You can specify global tolerance values to use when comparing two simulation runs. Global tolerance values are applied to all signals within the run. This example shows how to specify global tolerance values for a run comparison and how to analyze and save the comparison results.

First, load the session file that contains the data to compare. The session file contains data for four simulations of an aircraft longitudinal controller. This example compares data from two runs that use different input filter time constants.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

To access the run data to compare, use the `Simulink.sdi.getAllRunIDs` (Simulink) function to get the run IDs that correspond to the last two simulation runs.

```
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);
```

Use the `Simulink.sdi.compareRuns` (Simulink) function to compare the runs. Specify a global relative tolerance value of `0.2` and a global time tolerance value of `0.5`.

```
runResult = Simulink.sdi.compareRuns(runID1,runID2,'reltol',0.2,'timetol',0.5);
```

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object to see whether signals were within the tolerance values or out of tolerance.

```
runResult.Summary

ans = struct with fields:
        OutOfTolerance: 0
      WithinTolerance: 3
            Unaligned: 0
         UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 0
          TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

All three signal comparison results fell within the specified global tolerance.

You can save the comparison results to an MLDATX file using the `saveResult` (Simulink) function.

```
saveResult(runResult,'InputFilterComparison');
```

Analyze Simulation Data Using Signal Tolerances

You can programmatically specify signal tolerance values to use in comparisons performed using the Simulation Data Inspector. In this example, you compare data collected by simulating a model of an aircraft longitudinal flight control system. Each simulation uses a different value for the input filter time constant and logs the input and output signals. You analyze the effect of the time constant change by comparing results using the Simulation Data Inspector and signal tolerances.

First, load the session file that contains the simulation data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains four runs. In this example, you compare data from the first two runs in the file. Access the `Simulink.sdi.Run` objects for the first two runs loaded from the file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Now, compare the two runs without specifying any tolerances.

```
noTolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
```

Use the `getResultByIndex` function to access the comparison results for the `q` and `alpha` signals.

```
qResult = getResultByIndex(noTolDiffResult,1);
alphaResult = getResultByIndex(noTolDiffResult,2);
```

Check the `Status` of each signal result to see whether the comparison result fell within our out of tolerance.

```
qResult.Status

ans =
  ComparisonSignalStatus enumeration
```

```
    OutOfTolerance
```

alphaResult.Status

```
ans =
  ComparisonSignalStatus enumeration

    OutOfTolerance
```

The comparison used a value of `0` for all tolerances, so the `OutOfTolerance` result means the signals are not identical.

You can further analyze the effect of the time constant by specifying tolerance values for the signals. Specify the tolerances by setting the properties for the `Simulink.sdi.Signal` objects that correspond to the signals being compared. Comparisons use tolerances specified for the baseline signals. This example specifies a time tolerance and an absolute tolerance.

To specify a tolerance, first access the `Signal` objects from the baseline run.

```
runTs1 = Simulink.sdi.getRun(runIDTs1);
qSig = getSignalsByName(runTs1,'q, rad/sec');
alphaSig = getSignalsByName(runTs1,'alpha, rad');
```

Specify an absolute tolerance of `0.1` and a time tolerance of `0.6` for the `q` signal using the `AbsTol` and `TimeTol` properties.

```
qSig.AbsTol = 0.1;
qSig.TimeTol = 0.6;
```

Specify an absolute tolerance of `0.2` and a time tolerance of `0.8` for the `alpha` signal.

```
alphaSig.AbsTol = 0.2;
alphaSig.TimeTol = 0.8;
```

Compare the results again. Access the results from the comparison and check the `Status` property for each signal.

```
tolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
qResult2 = getResultByIndex(tolDiffResult,1);
alphaResult2 = getResultByIndex(tolDiffResult,2);
```

qResult2.Status

```
ans =
  ComparisonSignalStatus enumeration

    WithinTolerance
```

alphaResult2.Status

```
ans =
  ComparisonSignalStatus enumeration

    WithinTolerance
```

Configure Comparisons to Check Metadata

You can use the `Simulink.sdi.compareRuns` function to compare signal data and metadata, including data type and start and stop times. A single comparison may check for mismatches in one or more pieces of metadata. When you check for mismatches in signal metadata, the `Summary` property of the `Simulink.sdi.DiffRunResult` object may differ from a basic comparison because the `Status` property for a `Simulink.sdi.DiffSignalResult` object can indicate the metadata mismatch. You can configure comparisons using the `Simulink.sdi.compareRuns` function for imported data and for data logged from a simulation.

This example configures a comparison of runs created from workspace data three ways to show how the `Summary` of the `DiffSignalResult` object can provide specific information about signal mismatches.

Create Workspace Data

The `Simulink.sdi.compareRuns` function compares time series data. Create data for a sine wave to use as the baseline signal, using the `timeseries` format. Give the `timeseries` the name `Wave Data`.

```
time = 0:0.1:20;
sig1vals = sin(2*pi/5*time);
sig1_ts = timeseries(sig1vals,time);
sig1_ts.Name = 'Wave Data';
```

Create a second sine wave to compare against the baseline signal. Use a slightly different time vector and attenuate the signal so the two signals are not identical. Cast the signal data to the `single` data type. Also name this `timeseries` object `Wave Data`. The Simulation Data Inspector comparison algorithm will align these signals for comparison using the name.

```
time2 = 0:0.1:22;
sig2vals = single(0.98*sin(2*pi/5*time2));
sig2_ts = timeseries(sig2vals,time2);
sig2_ts.Name = 'Wave Data';
```

Create and Compare Runs in the Simulation Data Inspector

The `Simulink.sdi.compareRuns` function compares data contained in `Simulink.sdi.Run` objects. Use the `Simulink.sdi.createRun` function to create runs in the Simulation Data Inspector for the data. The `Simulink.sdi.createRun` function returns the run ID for each created run.

```
runID1 = Simulink.sdi.createRun('Baseline Run','vars',sig1_ts);
runID2 = Simulink.sdi.createRun('Compare to Run','vars',sig2_ts);
```

You can use the `Simulink.sdi.compareRuns` function to compare the runs. The comparison algorithm converts the signal data to the `double` data type and synchronizes the signal data before computing the difference signal.

```
basic_DRR = Simulink.sdi.compareRuns(runID1,runID2);
```

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object to see the result of the comparison.

```
basic_DRR.Summary

ans = struct with fields:
        OutOfTolerance: 1
       WithinTolerance: 0
             Unaligned: 0
          UnitsMismatch: 0
                 Empty: 0
              Canceled: 0
            EmptySynced: 0
       DataTypeMismatch: 0
           TimeMismatch: 0
      StartStopMismatch: 0
            Unsupported: 0
```

The difference between the signals is out of tolerance.

Compare Runs and Check for Data Type Match

Depending on your system requirements, you may want the data types for signals you compare to match. You can use the `Simulink.sdi.compareRuns` function to configure the comparison algorithm to check for and report data type mismatches.

```
dataType_DRR = Simulink.sdi.compareRuns(runID1,runID2,'DataType','MustMatch');
dataType_DRR.Summary

ans = struct with fields:
        OutOfTolerance: 0
       WithinTolerance: 0
             Unaligned: 0
          UnitsMismatch: 0
                 Empty: 0
              Canceled: 0
            EmptySynced: 0
       DataTypeMismatch: 1
           TimeMismatch: 0
      StartStopMismatch: 0
            Unsupported: 0
```

The result of the signal comparison is now `DataTypeMismatch` because the data for the baseline signal is `double` data type, while the data for the signal compared to the baseline is `single` data type.

Compare Runs and Check for Start and Stop Time Match

You can use the `Simulink.sdi.compareRuns` function to configure the comparison algorithm to check whether the aligned signals have the same start and stop times.

```
startStop_DRR = Simulink.sdi.compareRuns(runID1,runID2,'StartStop','MustMatch');
startStop_DRR.Summary

ans = struct with fields:
        OutOfTolerance: 0
       WithinTolerance: 0
             Unaligned: 0
          UnitsMismatch: 0
```

```
            Empty: 0
         Canceled: 0
      EmptySynced: 0
  DataTypeMismatch: 0
      TimeMismatch: 0
 StartStopMismatch: 1
       Unsupported: 0
```

The signal comparison result is now `StartStopMismatch` because the signals created in the workspace have different stop times.

Compare Runs with Alignment Criteria

When you compare runs using the Simulation Data Inspector, you can specify alignment criteria that determine how signals are paired with each other for comparison. This example compares data from simulations of a model of an aircraft longitudinal control system. The simulations used a square wave input. The first simulation used an input filter time constant of `0.1s` and the second simulation used an input filter time constant of `0.5s`.

First, load the simulation data from the session file that contains the data for this example.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains data for four simulations. This example compares data from the first two runs. Access the run IDs for the first two runs loaded from the session file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Before running the comparison, define how you want the Simulation Data Inspector to align the signals between the runs. This example aligns signals by their name, then by their block path, and then by their Simulink identifier.

```
alignMethods = [Simulink.sdi.AlignType.SignalName
                Simulink.sdi.AlignType.BlockPath
                Simulink.sdi.AlignType.SID];
```

Compare the simulation data in your two runs, using the alignment criteria you specified. The comparison uses a small time tolerance to account for the effect of differences in the step size used by the solver on the transition of the square wave input.

```
diffResults = Simulink.sdi.compareRuns(runIDTs1,runIDTs2,'align',alignMethods,...
    'timetol',0.005);
```

You can use the `getResultByIndex` function to access the comparison results for the aligned signals in the runs you compared. You can use the `Count` property of the `Simulink.sdi.DiffRunResult` object to set up a `for` loop to check the `Status` property for each `Simulink.sdi.DiffSignalResult` object.

```
numComparisons = diffResults.count;

for k = 1:numComparisons
    resultAtIdx = getResultByIndex(diffResults,k);
```

```
        sigID1 = resultAtIdx.signalID1;
        sigID2 = resultAtIdx.signalID2;

        sig1 = Simulink.sdi.getSignal(sigID1);
        sig2 = Simulink.sdi.getSignal(sigID2);

        displayStr = 'Signals %s and %s: %s \n';
        fprintf(displayStr,sig1.Name,sig2.Name,resultAtIdx.Status);
end

Signals q, rad/sec and q, rad/sec: OutOfTolerance
Signals alpha, rad and alpha, rad: OutOfTolerance
Signals Stick and Stick: WithinTolerance
```

Input Arguments

runID1 — Baseline run identifier
integer

Numeric identifier for the baseline run in the comparison, specified as a run ID that corresponds to a run in the Simulation Data Inspector. The Simulation Data Inspector assigns run IDs when runs are created. You can get the run ID for a run by using the `ID` property of the `Simulink.sdi.Run` object, the `Simulink.sdi.getAllRunIDs` function, or the `Simulink.sdi.getRunIDByIndex` function.

runID2 — Identifier for run to compare
integer

Numeric identifier for the run to compare, specified as a run ID that corresponds to a run in the Simulation Data Inspector. The Simulation Data Inspector assigns run IDs when runs are created. You can get the run ID for a run by using the `ID` property of the `Simulink.sdi.Run` object, the `Simulink.sdi.getAllRunIDs` function, or the `Simulink.sdi.getRunIDByIndex` function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `AbsTol=x,Align=alignOpts`

Align — Signal alignment options
Simulink.sdi.AlignType scalar | Simulink.sdi.AlignType vector

Signal alignment options, specified as a `Simulink.sdi.AlignType` scalar or vector. The `Simulink.sdi.AlignType` enumeration includes a value for each option available for pairing each signal in the baseline run with a signal in the comparison run. You can specify one or more alignment options for the comparison. To use more than one alignment option, specify an array. When you specify multiple alignment options, the Simulation Data Inspector aligns signals first by the option in the first element of the array, then by the option in the second element array, and so on. For more information, see "Signal Alignment" (Simulink).

Value	Aligns By
`Simulink.sdi.AlignType.BlockPath`	Path to the source block for the signal
`Simulink.sdi.AlignType.SID`	Automatically assigned Simulink identifier
`Simulink.sdi.AlignType.SignalName`	Signal name
`Simulink.sdi.AlignType.DataSource`	Path of the variable in the MATLAB workspace

Example: `[Simulink.sdi.AlignType.SignalName,Simulink.sdi.AlignType.BlockPath]` specifies signal alignment by signal name and then by block path.

AbsTol — Global absolute tolerance for comparison
`0` (default) | positive-valued scalar

Global absolute tolerance for comparison, specified as a positive-valued scalar.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about how tolerances are used in comparisons, see "Tolerance Specification" (Simulink).

Example: `0.5`

Data Types: `double`

RelTol — Global relative tolerance for comparison
`0` (default) | positive-valued scalar

Global relative tolerance for comparison, specified as a positive-valued scalar. The relative tolerance is expressed as a fractional multiplier. For example, `0.1` specifies a 10 percent tolerance.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about how tolerances are used in comparisons, see "Tolerance Specification" (Simulink).

Example: `0.1`

Data Types: `double`

TimeTol — Global time tolerance for comparison
`0` (default) | positive-valued scalar

Global time tolerance for comparison, specified as a positive-valued scalar, using units of seconds.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about tolerances in the Simulation Data Inspector, see "Tolerance Specification" (Simulink).

Example: `0.2`

Data Types: `double`

DataType — Comparison sensitivity to signal data types
`"MustMatch"`

Comparison sensitivity to signal data types, specified as `"MustMatch"`. Specify `DataType="MustMatch"` when you want the comparison to be sensitive to numeric data type mismatches in compared signals.

When signal data types do not match, the `Status` property of the `Simulink.sdi.DiffSignalResult` object for the result is set to `DataTypeMismatch`.

The `Simulink.sdi.compareRuns` function compares the data types for aligned signals before synchronizing and comparing the signal data. When you do not specify this name-value argument, the comparison checks data types only to detect a comparison between string and numeric data. For a comparison between string and numeric data, results are not computed, and the status for the result is `DataTypeMismatch`. For aligned signals that have different numeric data types, the comparison computes results.

When you configure the comparison to stop on the first mismatch, a data type mismatch stops the comparison. A stopped comparison may not compute results for all signals.

Time — Comparison sensitivity to signal time vectors
`"MustMatch"`

Comparison sensitivity to signal time vectors, specified as `"MustMatch"`. Specify `Time="MustMatch"` when you want the comparison to be sensitive to mismatches in the time vectors of compared signals. When you specify this name-value argument, the algorithm compares the time vectors of aligned signals before synchronizing and comparing the signal data.

When the time vectors for signals do not match, the `Status` property of the `Simulink.sdi.DiffSignalResult` object for the result is set to `TimeMismatch`.

Comparisons are not sensitive to differences in signal time vectors unless you specify this name-value argument. For comparisons that are not sensitive to differences in the time vectors, the comparison algorithm synchronizes the signals prior to the comparison. For more information about how synchronization works, see "How the Simulation Data Inspector Compares Data" (Simulink).

When you specify that time vectors must match and configure the comparison to stop on the first mismatch, a time vector mismatch stops the comparison. A stopped comparison may not compute results for all signals.

StartStop — Comparison sensitivity to signal start and stop times
`"MustMatch"`

Comparison sensitivity to signal start and stop times, specified as `"MustMatch"`. Specify `StartStop="MustMatch"` when you want the comparison to be sensitive to mismatches in signal start and stop times. When you specify this name-value argument, the algorithm compares the start and stop times for aligned signals before synchronizing and comparing the signal data.

When the start times and stop times do not match, the `Status` property of the `Simulink.sdi.DiffSignalResult` object for the result is set to `StartStopMismatch`.

When you specify that start and stop times must match and configure the comparison to stop on the first mismatch, a start or stop time mismatch stops the comparison. A stopped comparison may not compute results for all signals.

StopOnFirstMismatch — Whether comparison stops on first detected mismatch
"Metadata" | "Any"

Whether comparison stops on first detected mismatch without comparing remaining signals, specified as "Metadata" or "Any". A stopped comparison may not compute results for all signals, and can return a mismatched result more quickly.

- "Metadata" — A mismatch in metadata for aligned signals stops the comparison. Metadata comparisons happen before comparing signal data.

  The Simulation Data Inspector always aligns signals and compares signal units. When you configure the comparison to stop on the first mismatch, an unaligned signal or mismatched units always stop the comparison. You can specify additional name-value arguments to configure the comparison to check and stop on the first mismatch for additional metadata, such as signal data type, start and stop times, and time vectors.

- "Any" — A mismatch in metadata or signal data for aligned signals stops the comparison.

ExpandChannels — Whether to compute comparison results for each channel in multidimensional signals
true or 1 (default) | false or 0

Whether to compute comparison results for each channel in multidimensional signals, specified as logical true (1) or false (0).

- true or 1 — Comparison expands multidimensional signals represented as a single signal with nonscalar sample values to a set of signals with scalar sample values and computes a comparison result for each signal.

  The representation of the multidimensional signal in the Simulation Data Inspector as a single signal with nonscalar sample values does not change.

- false or 0 — Comparison does not compute results for multidimensional signals represented as a single signal with nonscalar sample values.

Output Arguments

diffResult — Comparison results
Simulink.sdi.DiffRunResult object

Comparison results, returned as a Simulink.sdi.DiffRunResult object.

Limitations

The Simulation Data Inspector does not support comparing:

- Signals of data types int64 or uint64.
- Variable-size signals.

Version History
Introduced in R2011b

See Also

Functions
Simulink.sdi.compareSignals | Simulink.sdi.getRunIDByIndex |
Simulink.sdi.getRunCount | getResultByIndex

Objects
Simulink.sdi.DiffRunResult | Simulink.sdi.DiffSignalResult

Topics
"Inspect and Compare Data Programmatically" (Simulink)
"Compare Simulation Data" (Simulink)
"How the Simulation Data Inspector Compares Data" (Simulink)

visionhdlframetoregions

Convert video frame dimensions into tiled regions of interest

Syntax

```
regions = visionhdlframetoregions(activePixelsPerLine,activeLines,
numHorTiles,numVerTiles)
regions = visionhdlframetoregions(activePixelsPerLine,activeLines,
numHorTiles,numVerTiles,'numPix',4)
regions = visionhdlframetoregions(activePixelsPerLine,activeLines,
numHorTiles,numVerTiles,'fillType','full')
```

Description

`regions = visionhdlframetoregions(activePixelsPerLine,activeLines, numHorTiles,numVerTiles)` converts an `activePixelsPerLine`-by-`activeLines` video frame into `numHorTiles×numVerTiles` nonoverlapping regions valid for vertical reuse with the ROI Selector block or the `visionhdl.ROISelector` System object. By default, the function returns a set of equally sized regions and ignores any remainder pixels at the right and bottom of the frame.

`regions = visionhdlframetoregions(activePixelsPerLine,activeLines, numHorTiles,numVerTiles,'numPix',4)` returns regions that are compatible with multipixel streaming. You can set `numPix` to 4 or 8.

`regions = visionhdlframetoregions(activePixelsPerLine,activeLines, numHorTiles,numVerTiles,'fillType','full')` returns regions that cover the entire frame size. If remainder pixels at the right and bottom of the frame exist, the function extends the right and bottom regions to include the remainder pixels.

In this diagram, the frame on the left shows the default behavior of the function. The frame is 160-by-120 pixels, divided into five horizontal and three vertical tiles. The function returns a set of fifteen 30-by-40 pixel regions and ignores the 10 remainder pixels at the right of the frame (blue).

In this diagram, the frame on the right shows the regions returned when you specify the `'filltype','full'` argument. Similar to the previous case, the frame is 160-by-120 pixels, divided into five horizontal and three vertical tiles. The function returns twelve 30-by-40 pixel regions, and three 40-by-40 regions that include the remainder pixels at the right of the frame.

'none'   'full'
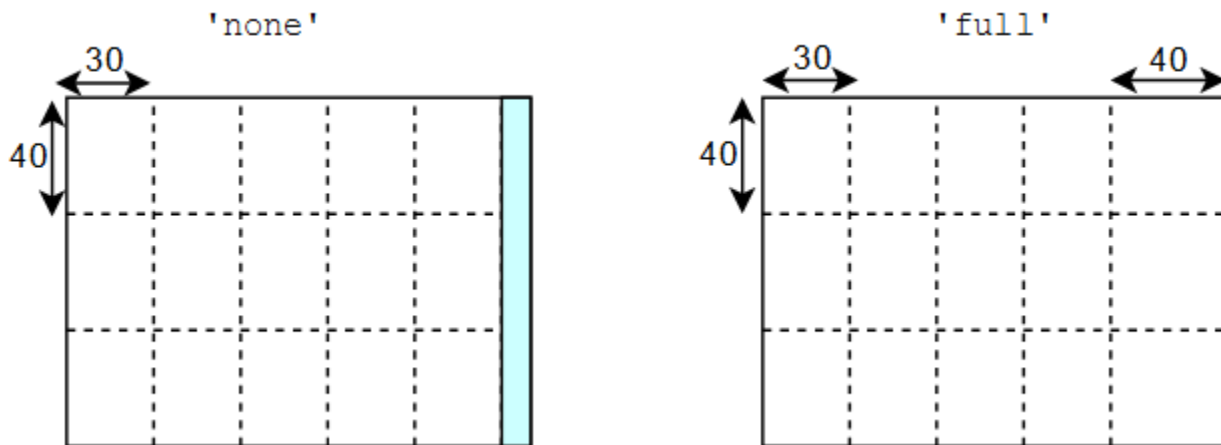
Examples

Select Regions for Vertical Reuse

This example shows how to divide a frame into tiled regions of interest (ROIs) and use those regions to configure the ROI Selector block for vertical reuse.

Vertical reuse means dividing each frame into vertically-aligned regions where each column of regions shares a pixel stream. This arrangement enables parallel processing of each column, and the reuse of downstream processing logic for each region in the column.

Set up the size of the frame.

```
frmActiveLines = 240;
frmActivePixels = 320;
```

Divide the frame into equally-sized vertically-aligned regions, or tiles. The `visionhdlframetoregions` function returns an array of such regions, where each region is defined by four coordinates, and is of the form [*hPos vPos hSize vSize*]. These tile counts divide evenly into the frame dimensions, so no remainder pixels exist. The output regions cover the entire frame.

```
numHorTiles = 2;
numVerTiles = 2;
regions = visionhdlframetoregions(frmActivePixels,frmActiveLines,numHorTiles,numVerTiles)
```

```
regions =

     1      1    160    120
   161      1    160    120
     1    121    160    120
   161    121    160    120
```

The ROI Selector block in the Simulink model has the **Reuse output ports for vertically aligned regions** parameter selected, and uses the `regions` variable to define its output streams. The block has one output pixel stream per column of regions.

```
open_system('TiledROIHDL')
```
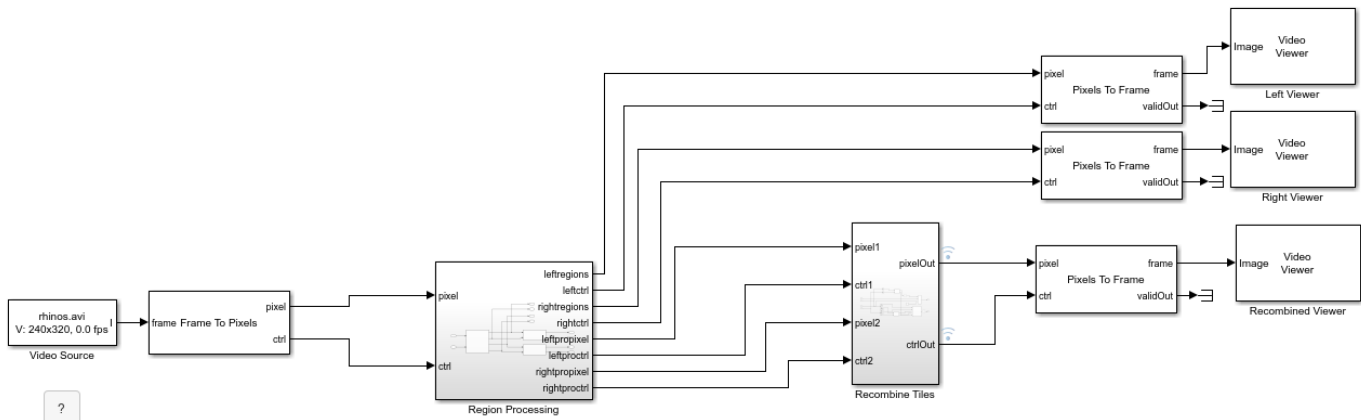

Copyright 2020 The MathWorks, Inc.

The start and end signals define each region in the pixel stream. When you run the model, you can see the output tiled regions changing in the Left Viewer and Right Viewer windows. The example performs opposite gamma correction operations on the left and right tiles, and then reassembles the four tiles into a complete frame by manipulating the `pixelcontrol` signals.

The blanking interval required by the downstream processing algorithm must be less than the interval between tiles. The blanking interval after each region is less than one line of pixels, so operations that require a vertical blanking interval, like those that use a line buffer, do not work. The gamma correction operation uses a lookup table that does not require a blanking interval.

```
sim('TiledROIHDL')
```
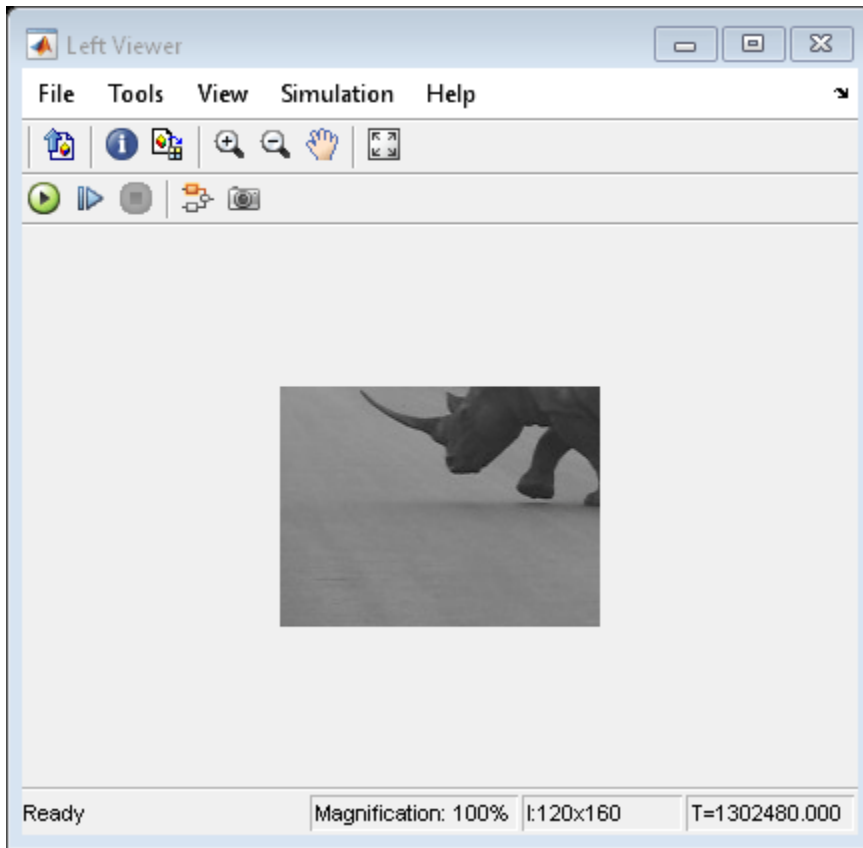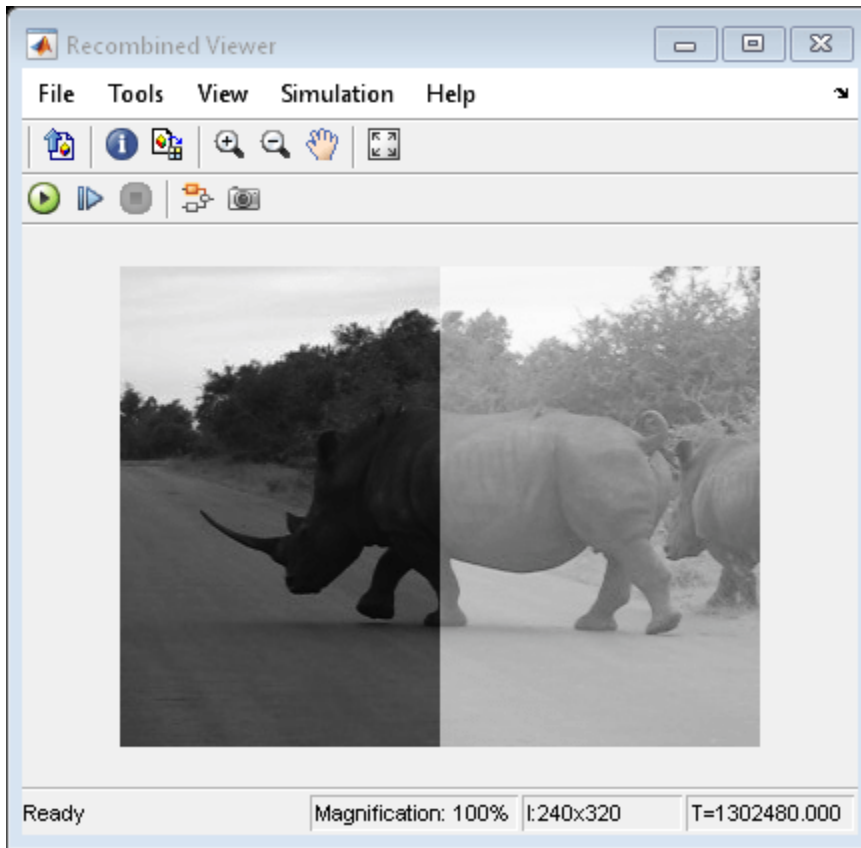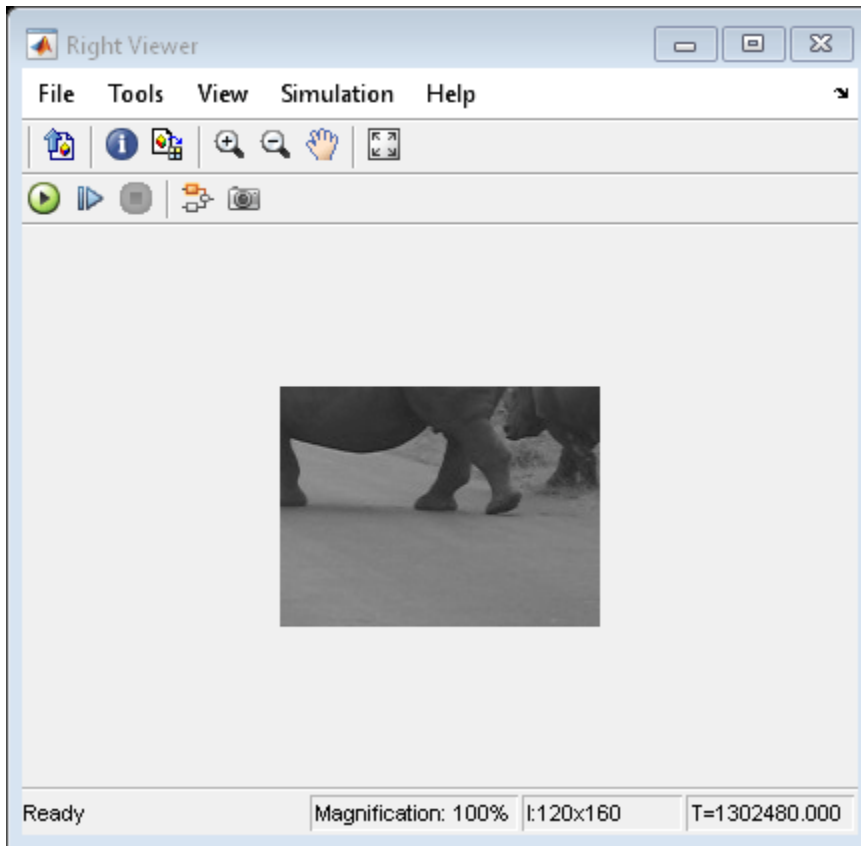
Input Arguments

activePixelsPerLine — Number of pixels in horizontal line of active input frame
positive integer

Number of pixels in a horizontal line of the active input frame, specified as a positive integer. The function divides this dimension into `numHorTiles` columns. Any remainder pixels from this division are handled according to the `'filltype'` setting.

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

activeLines — Number of horizontal lines in active input frame
positive integer

Number of horizontal lines in the active input frame, specified as a positive integer. The function divides this dimension into `numVerTiles` rows. Any remainder pixels from this division are handled according to the `'filltype'` setting.

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

numHorTiles — Number of tiles in the horizontal direction
positive integer

Number of tiles in the horizontal direction, specified as a positive integer. The function returns `numHorTiles`×`numVerTiles` regions. This value must be in the range [1, 16].

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

numVerTiles — Number of tiles in the vertical direction
positive integer

Number of tiles in the vertical direction, specified as a positive integer. The function returns `numHorTiles`×`numVerTiles` regions. This value must be in the range [1, 1024].

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

Output Arguments

regions — Tiled regions of interest
numHorTiles×numVerTiles-by-4 matrix

Tiled regions of interest, returned as a `numHorTiles`×`numVerTiles`-by-4 matrix. Each region is represented by four positive integers that define the coordinates of the top-left corner and dimensions of the region, [*hPos vPos hSize vSize*]. Use these regions to configure the ROI Selector block or `visionhdl.ROISelector` System object for vertical reuse. Regions in each column share an output pixel stream of the block or System object.

Version History
Introduced in R2020b

See Also

Blocks
ROI Selector

Objects
`visionhdl.ROISelector`

visionhdlsetup

Set up model parameters for HDL code generation for streaming video

Syntax

```
visionhdlsetup(modelname)
```

Description

`visionhdlsetup(modelname)` sets the parameters of a Simulink model specified by `modelname`, to the recommended values for HDL code generation for streaming video designs. Use this function instead of `hdlsetup`.

Open the model before you call this function. If you do not have an HDL Coder license, this function returns a warning. After you call this function, you can modify any of HDL Coder parameters using `set_param` or **Model Parameters** > **HDL Code Generation**.

When you generate HDL code for a model that you have configured using the `visionhdlsetup` function, in the HDL Code Generation Check Report there is a warning about `BalanceDelays`.

```
'BalanceDelays' is set to 'Off' for the model.
```

This warning is acceptable since Vision HDL Toolbox blocks do not participate in automatic pipelining or delay balancing during HDL code generation.

Examples

Configure a Model for HDL Code Generation with Streaming Video

Set model parameters to recommended values for HDL code generation for streaming video designs.

Open a model.

```
open_system('MyHDLModel');
```

Call `visionhdlsetup` to configure the model.

```
visionhdlsetup('MyHDLModel');
```

Input Arguments

modelname — Simulink model to configure
character vector | string

Simulink model to configure, specified as a character vector or string. The model must be loaded before you call this function.

Data Types: `char` | `string`

Version History
Introduced in R2019b

See Also
`hdlsetup`

Apps and Tools

Simulation Data Inspector

Inspect and compare data and simulation results to validate and iterate model designs

Description

The Simulation Data Inspector visualizes and compares multiple kinds of data.

Using the Simulation Data Inspector, you can inspect and compare time series data at multiple stages of your workflow. This example workflow shows how the Simulation Data Inspector supports all stages of the design cycle:

1 "View Data in the Simulation Data Inspector" (Simulink)

Run a simulation in a model configured to log data to the Simulation Data Inspector, or import data from the workspace or a MAT-file. You can view and verify model input data or inspect logged simulation data while iteratively modifying your model diagram, parameter values, or model configuration.

2 "Inspect Simulation Data" (Simulink)

Plot signals on multiple subplots, zoom in and out on specified plot axes, and use data cursors to understand and evaluate the data. "Create Plots Using the Simulation Data Inspector" (Simulink) to tell your story.
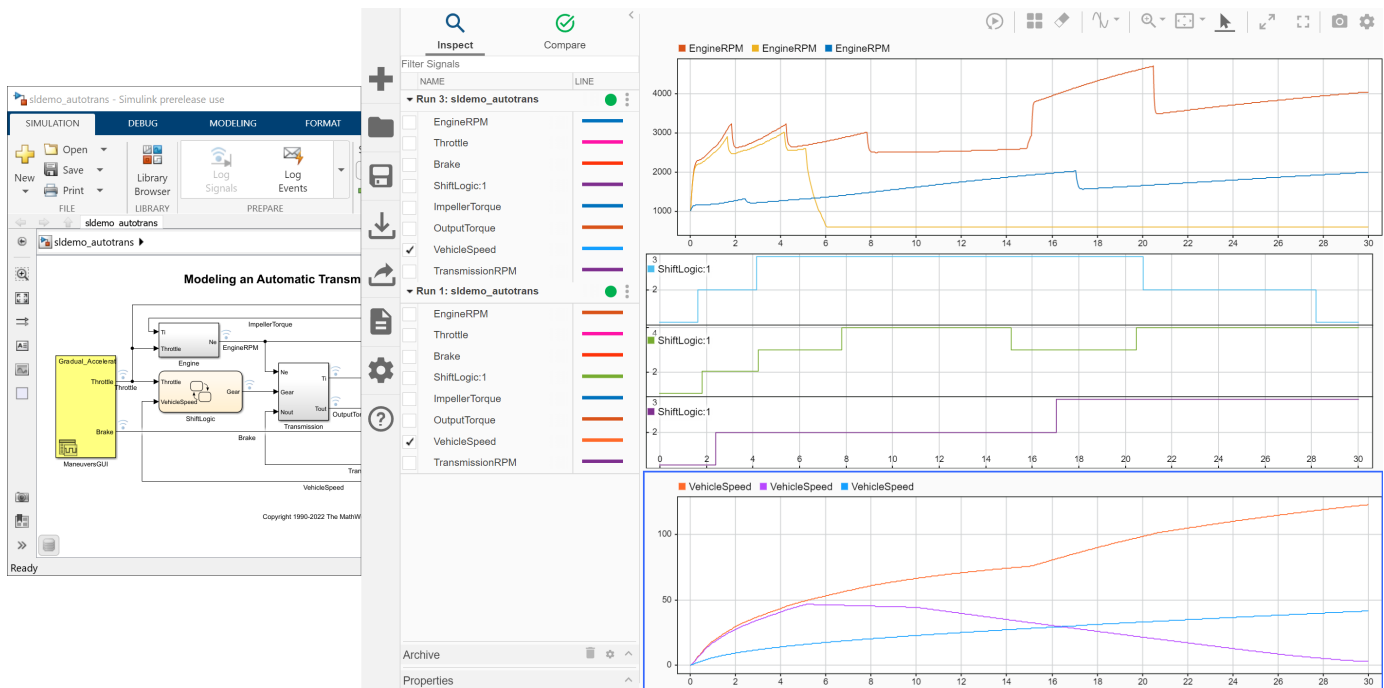
3 "Compare Simulation Data" (Simulink)

Compare individual signals or simulation runs and analyze your comparison results with relative, absolute, and time tolerances. The compare tools in the Simulation Data Inspector facilitate iterative design and allow you to highlight signals that do not meet your tolerance requirements. For more information about the comparison operation, see "How the Simulation Data Inspector Compares Data" (Simulink).

4 "Save and Share Simulation Data Inspector Data and Views" (Simulink)

Share your findings with others by saving Simulation Data Inspector data and views.

You can also harness the capabilities of the Simulation Data Inspector from the command line. For more information, see "Inspect and Compare Data Programmatically" (Simulink).

Open the Simulation Data Inspector

- Simulink Toolstrip: On the **Simulation** tab, under **Review Results**, click **Data Inspector**.
- Click the streaming badge on a signal to open the Simulation Data Inspector and plot the signal.
- MATLAB command prompt: Enter `Simulink.sdi.view`.

Examples

Apply a Tolerance to a Signal in Multiple Runs

You can use the Simulation Data Inspector programmatic interface to modify a parameter for the same signal in multiple runs. This example adds an absolute tolerance of `0.1` to a signal in all four runs of data.

First, clear the workspace and load the Simulation Data Inspector session with the data. The session includes logged data from four simulations of a Simulink® model of a longitudinal controller for an aircraft.

```
Simulink.sdi.clear
Simulink.sdi.load('AircraftExample.mldatx');
```

Use the `Simulink.sdi.getRunCount` function to get the number of runs in the Simulation Data Inspector. You can use this number as the index for a for loop that operates on each run.

```
count = Simulink.sdi.getRunCount;
```

Then, use a for loop to assign the absolute tolerance of `0.1` to the first signal in each run.

```matlab
for a = 1:count
    runID = Simulink.sdi.getRunIDByIndex(a);
    aircraftRun = Simulink.sdi.getRun(runID);
    sig = getSignalByIndex(aircraftRun,1);
    sig.AbsTol = 0.1;
end
```

- "View Data in the Simulation Data Inspector" (Simulink)
- "Inspect Simulation Data" (Simulink)
- "Compare Simulation Data" (Simulink)
- "Iterate Model Design Using the Simulation Data Inspector" (Simulink)

Programmatic Use

`Simulink.sdi.view` opens the Simulation Data Inspector from the MATLAB command line.

Version History
Introduced in R2010b

See Also

Functions
`Simulink.sdi.clear` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.snapshot`

Topics
"View Data in the Simulation Data Inspector" (Simulink)
"Inspect Simulation Data" (Simulink)
"Compare Simulation Data" (Simulink)
"Iterate Model Design Using the Simulation Data Inspector" (Simulink)