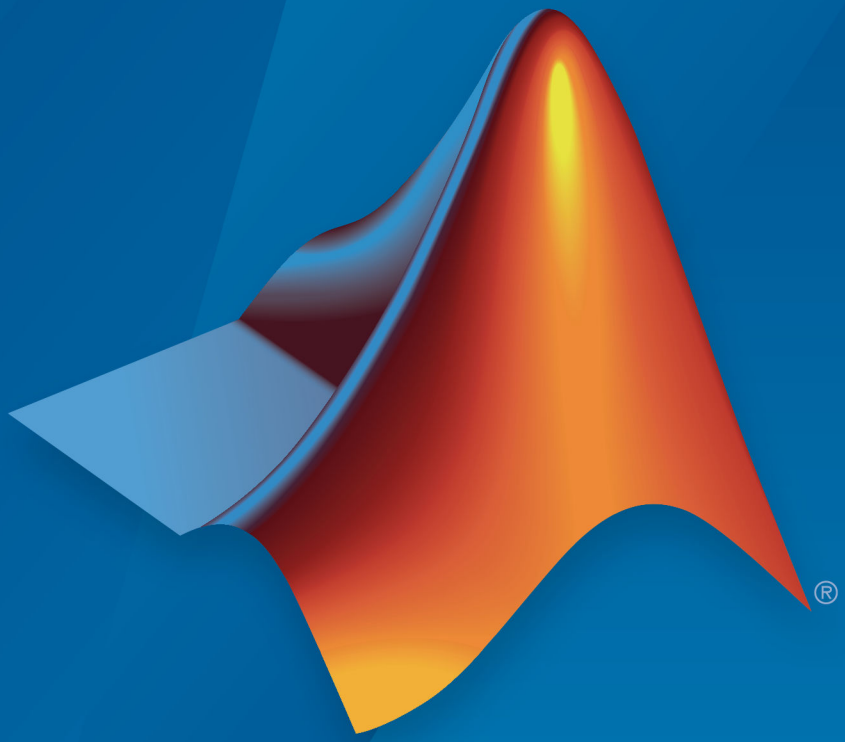
Vision HDL Toolbox™

Reference



MATLAB®

R2019b

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

Vision HDL Toolbox™ Reference

Revision History

March 2015	Online only	New for Version 1.0 (Release R2015a)
September 2015	Online only	Revised for Version 1.1 (Release R2015b)
March 2016	Online only	Revised for Version 1.2 (Release R2016a)
September 2016	Online only	Revised for Version 1.3 (Release R2016b)
March 2017	Online only	Revised for Version 1.4 (Release R2017a)
September 2017	Online only	Revised for Version 1.5 (Release R2017b)
March 2018	Online only	Revised for Version 1.6 (Release 2018a)
September 2018	Online only	Revised for Version 1.7 (Release 2018b)
March 2019	Online only	Revised for Version 1.8 (Release 2019a)
September 2019	Online only	Revised for Version 2.0 (Release 2019b)

Contents

Blocks — Alphabetical List

Bilateral Filter

2-D bilateral filtering
Library: 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` — Single image pixel
scalar

Single image pixel of a pixel stream, specified as a scalar value representing intensity. This value is interpreted on the range [0,1], assuming the entire range of the input data type. Integer and fixed-point data types larger than 16 bits are not supported.

`double` and `single` data types are supported 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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 image pixel

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.

`double` and `single` data types are supported 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

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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

`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`

Method for padding the boundary of the input image.

- `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` — Pad the input matrix with its mirror image.

For more information on these methods, see “Edge 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 single horizontal line.

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 mode for internal fixed-point calculations
`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 option does not apply when the input data type is `single` or `double`.

`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 Data Type` — 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.

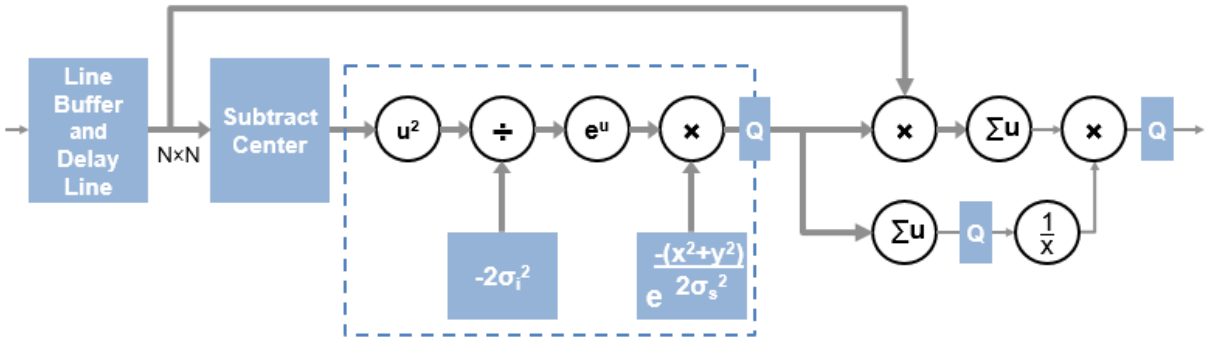
`Output Data Type` — Method to determine data type of output pixels
`Inherit: Same as first input` (default) | `fixdt(1,16,0)` | data type expression

The filtered pixel values are cast to this data type.

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.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

See Also

Blocks
Frame To Pixels

Objects
`visionhdl.BilateralFilter`

Introduced in R2017b

Birds-Eye View

Transform front-facing camera image into top-down view
Library: 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` — Single image pixel

scalar

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

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 image pixel

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.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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 “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

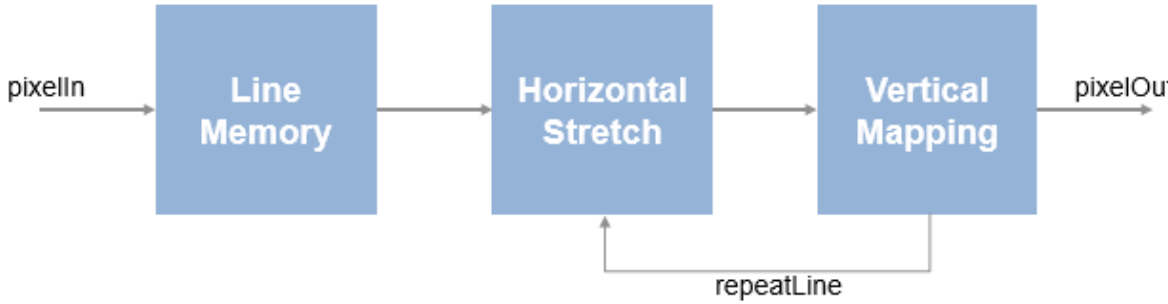
Vertical size of output frame, specified as an integer. This parameter is the number of active lines in each output frame.

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.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

See Also

Blocks
Frame To Pixels | Warp

Objects
`visionhdl.BirdsEyeView`

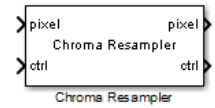
Functions
`estimateGeometricTransform` | `imwarp`

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

Introduced in R2017b

Chroma Resampler

Downsample or upsample chrominance component
Library: 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. The values can be `uint8`, `uint16`, or `fixdt(0,N,0)`, where *N* is from 8 through 16.

`double` and `single` data types are supported 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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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.

`double` and `single` data types are supported 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

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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`

Rounding mode applies when you select linear interpolation, or include an antialiasing filter.

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

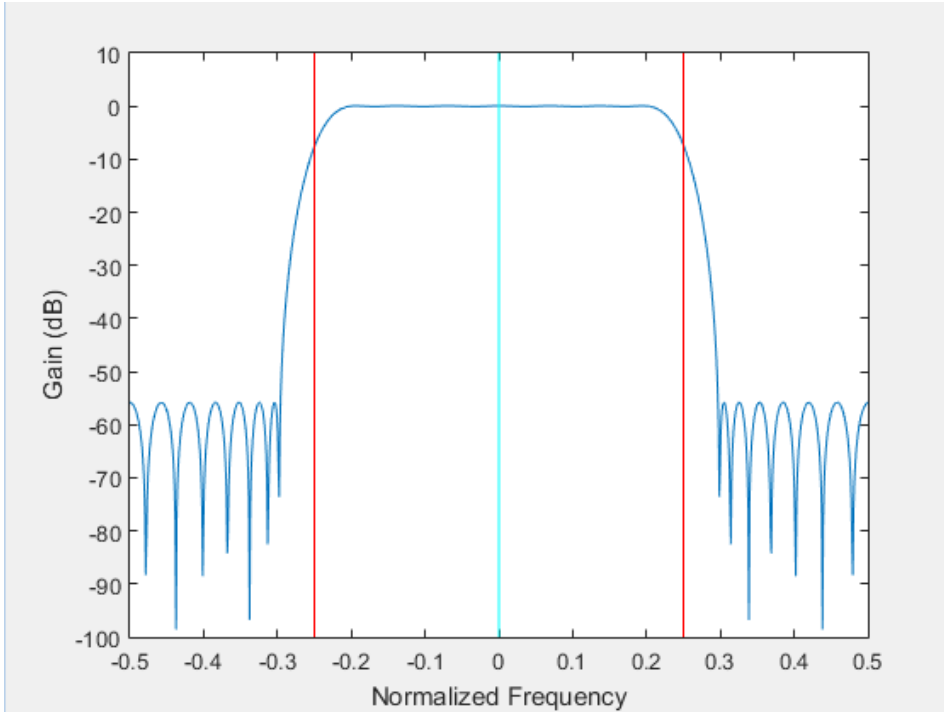
Overflow can occur when you include an antialiasing filter. By default, fixed-point values wrap around on overflow.

`Filter coefficients` — Data type for antialiasing filter coefficients
`fixdt(1,16,0)` (default) | data type expression

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

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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

See Also

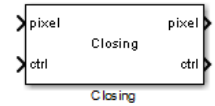
Blocks
Chroma Resampling | Frame To Pixels

Objects
`visionhdl.ChromaResampler`

Introduced in R2015a

Closing

Morphological closing of binary pixel data
Library: 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 `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”.

Ports

Input

`pixel` — Single input pixel

Boolean scalar

Single input pixel, specified as a Boolean scalar.

Data Types: `Boolean`

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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

Boolean scalar

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

Data Types: `Boolean`

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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
`[0,1,0; 1,1,1; 0,1,0]` (default) | vector or matrix of `1`s and `0`s

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The block supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, specify **Neighborhood** as `getnhood(strel(`*shape*`))`.

`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 single horizontal line.

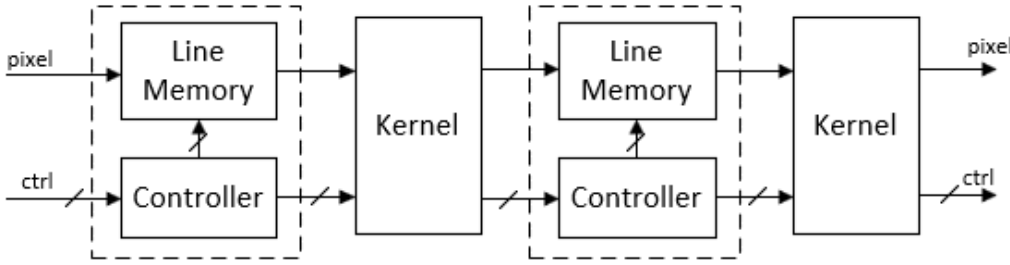
If you specify a value that is not a power of two, the block uses the next largest power of two. The block allocates (*neighborhood lines* – 1)-by-**Line buffer size** memory locations to store the pixels.

Algorithms

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. 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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

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

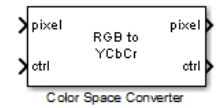
Topics

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

Introduced in R2015a

Color Space Converter

Convert color information between color spaces
Library: 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. 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”.

Input

`pixel` — Single image pixel

vector of three values

Single image pixel in Y'CbCr or R'G'B' color space, specified as a vector of three values. The values can be `uint8`, `uint16`, or `fixdt(0,N,0)`, where *N* is from 8 through 16.

`double` and `single` data types are supported 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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 image pixel in new colorspace
scalar | vector of three values

Single image pixel in intensity, Y'CbCr, or R'G'B' color space, returned as a scalar or a vector of three values. The data type of the output pixels is the same as the data type of the input pixels.

`double` and `single` data types are supported 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

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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

`Conversion` — Type of color space conversion
`RGB to YCbCr` (default) | `YCbCr to RGB` | `RGB to intensity`

The block 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.

`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

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	$\begin{bmatrix} 0.25678824 & 0.50412941 & 0.09790588 \\ -0.1482229 & -0.29099279 & 0.43921569 \\ 0.43921569 & -0.36778831 & -0.0714273 \end{bmatrix}$	$\begin{bmatrix} 0.18258588 & 0.61423059 & 0.06256706 \\ -0.1006437 & -0.33857195 & 0.43921569 \\ 0.43921569 & -0.39894216 & -0.0402735 \end{bmatrix}$	$\begin{bmatrix} 0.25678824 & 0.50412941 & 0.09790588 \\ -0.1482229 & -0.29099279 & 0.43921569 \\ 0.43921569 & -0.36778831 & -0.0714273 \end{bmatrix}$
B	$\begin{bmatrix} 1.1643836 & 0 & 1.5960268 \\ 1.1643836 & -0.39176229 & -0.81296765 \\ 1.1643836 & 2.0172321 & 0 \end{bmatrix}$	$\begin{bmatrix} 1.1643836 & 0 & 1.7927411 \\ 1.1643836 & -0.21324861 & -0.5329093 \\ 1.1643836 & 2.1124018 & 0 \end{bmatrix}$	$\begin{bmatrix} 1.1643836 & 0 & 1.5960268 \\ 1.1643836 & -0.39176229 & -0.81296765 \\ 1.1643836 & 2.0172321 & 0 \end{bmatrix}$

Conversion from R'G'B' to Intensity

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

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

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

Blocks with R'G'B' input have a latency of 9 cycles and blocks with Y'CbCr input have a latency of 10 cycles. The extra cycle is required to check for and correct headroom and footroom violations.

Note The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

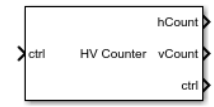
See Also

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

Introduced in R2015a

HV Counter

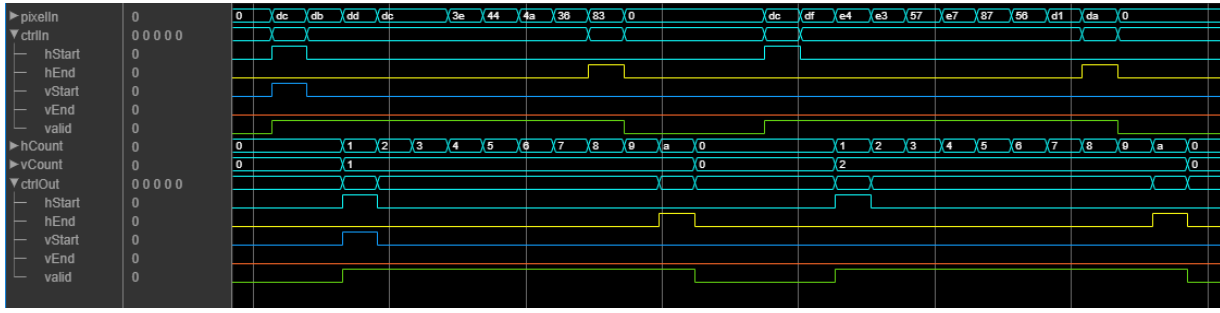
Count active dimensions of pixel stream
Library: Vision HDL Toolbox / Utilities



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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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(log2(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(log2(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.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

See Also

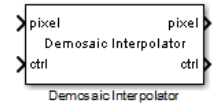
Blocks
Frame To Pixels

Objects
`visionhdl.HVCounter`

Introduced in R2019a

Demosaic Interpolator

Construct RGB pixel data from Bayer pattern pixels
Library: 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 3×3 pixel window using only additions and bit shifts.
- When you select gradient correction, the block operates on a 5×5 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”.

Input

`pixel` — Single image pixel

scalar

Single image pixel, specified as a scalar. Images in the Bayer format have one color component for each pixel location. Select the sequence of R, G, and B pixels with the **Sensor alignment** parameter.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 image pixel

three-element vector of RGB values

Single image pixel, returned as a three-element vector of 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.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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

`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 3×3 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 single horizontal line.

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 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

Algorithms

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

Interpolation

Bilinear Interpolation

The block interpolates the missing color values by using a 3×3 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.

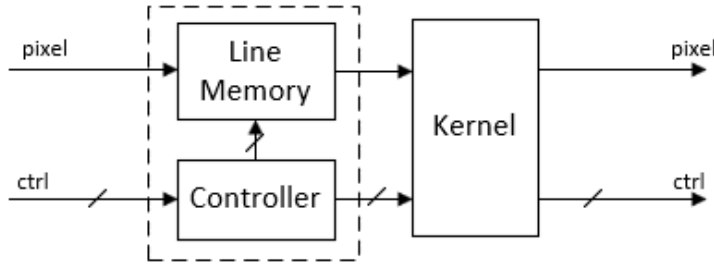
Gradient-Corrected Linear Interpolation

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 5×5 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. 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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

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 a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

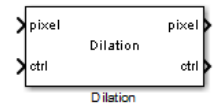
Blocks
Demosaic | Frame To Pixels

Objects
`visionhdl.DemosaicInterpolator`

Introduced in R2015a

Dilation

Morphological dilation of binary pixel data
Library: 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 `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”.

Ports

Input

`pixel` — Single input pixel
Boolean scalar

Single input pixel, specified as a Boolean scalar.

Data Types: `Boolean`

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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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

Boolean scalar

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

Data Types: `Boolean`

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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 `1`s and `0`s

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The block supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, specify **Neighborhood** as `getnhood(strel(`*shape*`))`.

`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 single horizontal line.

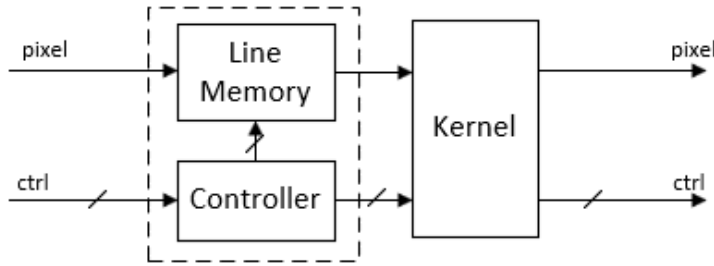
If you specify a value that is not a power of two, the block uses the next largest power of two. The block allocates (*neighborhood lines* – 1)-by-**Line buffer size** memory locations to store the pixels.

Algorithms

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

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. 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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL code for this block if it is inside a Resettable Synchronous Subsystem.

See Also

Erosion | Frame To Pixels | `visionhdl.Dilation`

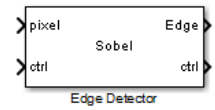
Topics

“Types of Morphological Operations” (Image Processing Toolbox)
“Structuring Elements” (Image Processing Toolbox)

Introduced in R2015a

Edge Detector

Find edges of objects in grayscale pixel stream
Library: 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 4 or 8 pixels per clock cycle to support high-frame-rate or high-resolution formats. 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 four or eight 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 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 intensity value. For multipixel streaming, specify a vector of four or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see “Filter Multipixel Video Streams”.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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.

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

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

Output

Edge — Pixel value, indicating whether pixel is an edge

Boolean scalar | Boolean vector

For single pixel streaming, **Edge** is a Boolean scalar. For multipixel streaming, **Edge** is a vector of *NumberOfPixels*-by-1 Boolean values. *NumberOfPixels* can be four or eight. 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 *NumberOfPixels*-by-1 values. *NumberOfPixels* can be four or eight.

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

Dependencies

These ports are visible when you set **Method** to `Sobel` or `Prewitt`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `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 *NumberOfPixels*-by-1 values. *NumberOfPixels* can be four or eight.

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

Dependencies

These ports are visible when you set **Method** to `Roberts`.

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 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`

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 1-55.

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 single 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 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.

Data Types

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

Rounding mode for internal fixed-point calculations.

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

When this parameter is `off`, fixed-point and integer values wrap around to 0 when the value overflows what is representable with that data type. When this parameter is `on`, 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**.

Algorithms

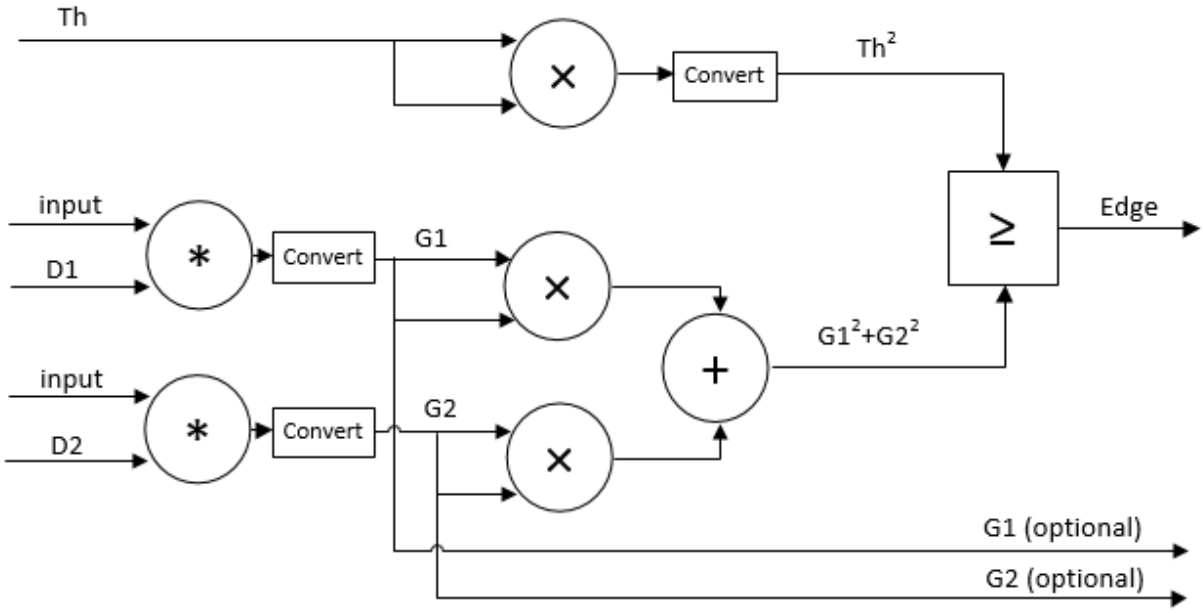
The Edge Detector block provides three methods for detecting edges in an input image. The methods use different derivative approximation matrices to find two orthogonal gradients. The Sobel and Prewitt methods calculate the gradient in horizontal and vertical directions. The Roberts method calculates 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 *NumberOfPixels* filters 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	$\frac{1}{8}\begin{bmatrix}1 & 0 & -1\\2 & 0 & -2\\1 & 0 & -1\end{bmatrix}$	$\frac{1}{8}\begin{bmatrix}1 & 2 & 1\\0 & 0 & 0\\-1 & -2 & -1\end{bmatrix}$
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**.

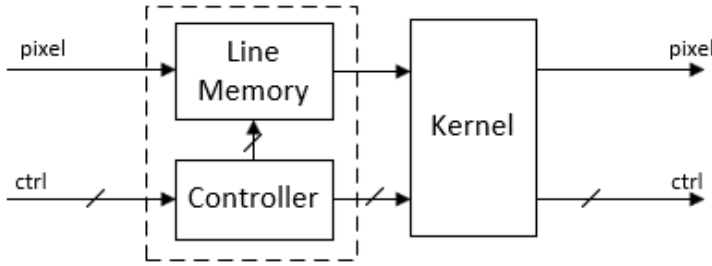
The block convolves the neighborhood of the input pixel with the derivative matrices, D1 and D2. It then compares the sum of the squares of the gradients to the square of the threshold. Computing the square of the threshold avoids constructing a square root circuit. The block casts the gradients to the 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.



The block pads the edge of the image with symmetric pixel values. See “Edge Padding”.

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. 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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

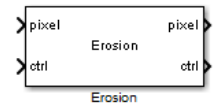
Blocks
Edge Detection | Frame To Pixels

Objects
`visionhdl.EdgeDetector`

Introduced in R2015a

Erosion

Morphological erosion of binary pixel data
Library: 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 `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”.

Ports

Input

`pixel` — Single input pixel
Boolean scalar

Single input pixel, specified as a Boolean scalar.

Data Types: `Boolean`

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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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

Boolean scalar

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

Data Types: `Boolean`

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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 `1`s and `0`s

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The block supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, specify **Neighborhood** as `getnhood(strel(`*shape*`))`.

`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 single horizontal line.

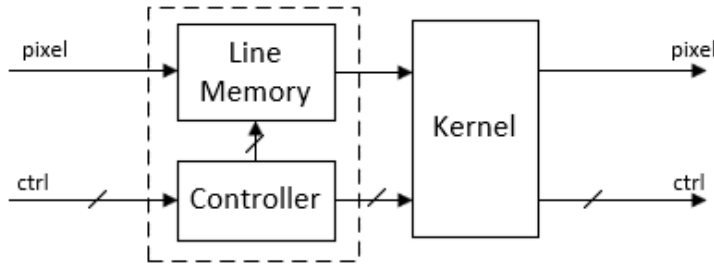
If you specify a value that is not a power of two, the block uses the next largest power of two. The block allocates (*neighborhood lines* – 1)-by-**Line buffer size** memory locations to store the pixels.

Algorithms

The block pads the edge of the image with ones for the erosion operation. For more information, see “Edge Padding”.

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. 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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL code for this block if it is inside a Resettable Synchronous Subsystem.

See Also

Dilation | Frame To Pixels | `visionhdl.Erosion`

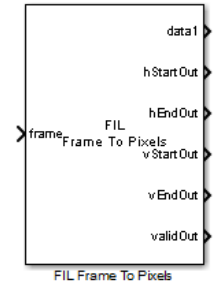
Topics

“Types of Morphological Operations” (Image Processing Toolbox)
“Structuring Elements” (Image Processing Toolbox)

Introduced in R2015a

FIL Frame To Pixels

Convert frame-based video to pixel stream for FPGA-in-the-loop
Library: 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.

When you use a multipixel stream (**Number of pixels** > 1), you must represent each pixel with a single value (*N* = 1).

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

Output

`data1,...,dataN` — Image pixels

vector | matrix

Image pixels, returned as a vector of *M* 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 a multipixel stream (**Number of pixels** > 1), you must represent each pixel with a single value (*N* = 1). The output data for multipixel streaming is a **Number of pixels**-by-*M* 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”.

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) | 3 | 4

Number of values used to represent each pixel. Each pixel can have 1, 3, or 4 components. Set this parameter to 1 for grayscale video. Set this parameter to 3 for color video, for example, {R,G,B} or {Y,Cb,Cr}. Set this parameter to 4 to use color with an alpha channel for transparency.

Dependencies

When **Number of pixels** is greater than 1, you must set **Number of components** to 1.

`Number of pixels` — Number of pixels in each column of output matrix
1 (default) | 4 | 8

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

Dependencies

When you use a multipixel stream (**Number of pixels** > 1), you must represent each pixel with a single value ($N = 1$). The output data for multipixel streaming is a **Number of pixels**-by-M 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.

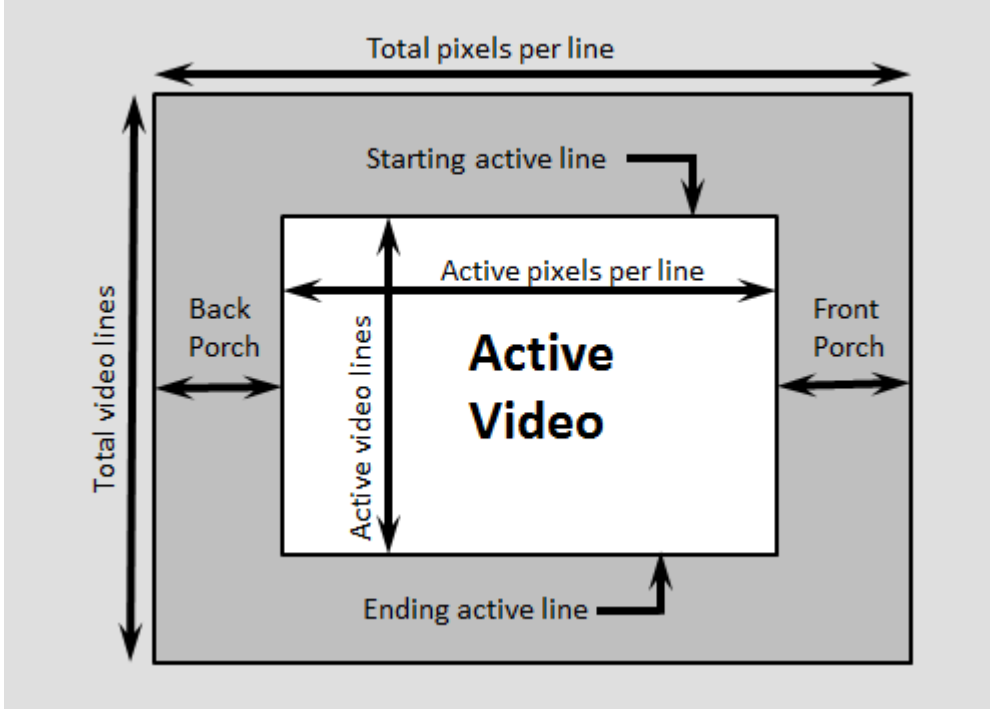
For multipixel streaming, specify one of these options for output size.

- `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**.

`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.

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.

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, 720p, 1080p, and 4k UHD support **Number of pixels** equal to 4 or 8.
- The minimum input frame size for multipixel streaming (either 4 or 8 pixels) 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.

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

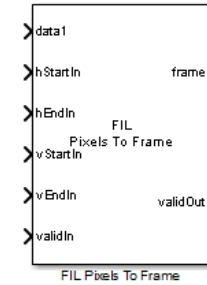
“Streaming Pixel Interface”

“FPGA Verification” (HDL Verifier)

Introduced in R2015a

FIL Pixels To Frame

Convert pixel stream from FPGA-in-the-loop to frame-based video
Library: 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 vector of *M* values, where *M* is the width of the **Input vector format**. The block has *N* data ports, where *N* is the **Number of components**.

When you use a multipixel stream (**Number of pixels** > 1), you must represent each pixel with a single value (*N* = 1). The output data for multipixel streaming is a **Number of pixels**-by-*M* 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".

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 returned 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**.

When you use a multipixel stream (**Number of pixels** > 1), you must represent each pixel with a single value (*N* = 1).

`validOut` — Indicates when output frame is ready

`Boolean` 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) | 3 | 4

Number of values used to represent each pixel. Each pixel can have 1, 3, or 4 components. Set this parameter to 1 for grayscale video. Set this parameter to 3 for color video, for example, {R,G,B} or {Y,Cb,Cr}. Set this parameter to 4 to use color with an alpha channel for transparency.

Dependencies

When **Number of pixels** is greater than 1, you must set **Number of components** to 1.

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

1 (default) | 4 | 8

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

Dependencies

When you use a multipixel stream (**Number of pixels** > 1), you must represent each pixel with a single value ($N = 1$). The output data for multipixel streaming is a **Number of pixels**-by-M 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”.

`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**.

`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

Video Format	Active Pixels Per Line	Active Video Lines
8KUHDTV	7680	4320
Custom	User-defined	User-defined

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)

Introduced in R2015a

Frame To Pixels

Convert frame-based video to pixel stream
Library: 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 scalar streaming or multipixel streaming. Multipixel streaming provides 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 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.

When you use a multipixel stream (**Number of pixels** > 1), you must represent each pixel with a single value (*N* = 1).

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

Output

`pixel` — Image pixel or multipixel vector

scalar | vector

For scalar 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 vector of **Number of pixels**-by-1 pixel intensity values. When **Number of pixels** is greater than 1, the **Number of components** must be 1.

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 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`

Parameters

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

1 (default) | 3 | 4

Number of values used to represent each pixel. Each pixel can have 1, 3, or 4 components. Set this parameter to 1 for grayscale video. Set this parameter to 3 for color

video, for example, {R,G,B} or {Y,Cb,Cr}. Set this parameter to 4 to use color with an alpha channel for transparency.

Dependencies

When **Number of pixels** is greater than 1, you must set **Number of components** to 1.

`Number of pixels` — Number of pixels streamed per time step

1 (default) | 4 | 8

Number of pixels transferred on the streaming interface for each time step, specified as 1, 4, or 8. To enable multipixel streaming and increase throughput for high-resolution or high-frame-rate video, set this parameter to 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

Dependencies

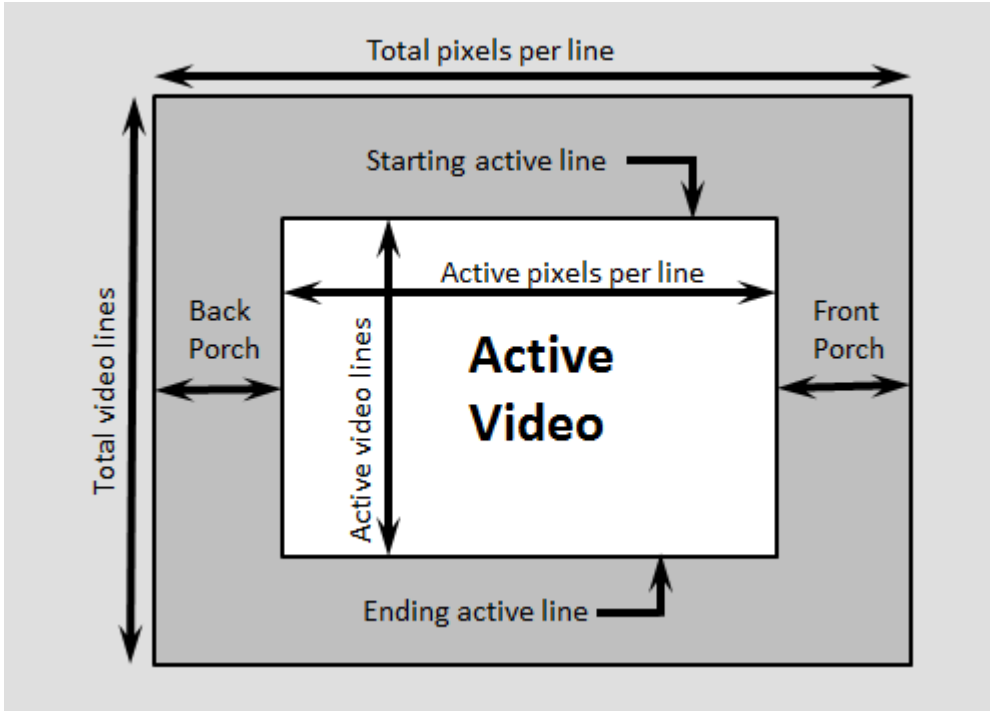
When **Number of pixels** is greater than 1, you must set **Number of components** to 1.

`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.

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.

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, 720p, 1080p, and 4k UHD support **Number of pixels** equal to 4 or 8.
- The minimum input frame size for multipixel streaming (either 4 or 8 pixels) 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.

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`

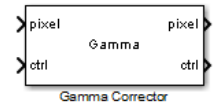
Topics

“Streaming Pixel Interface”

Introduced in R2015a

Gamma Corrector

Apply or remove gamma correction of pixel stream
Library: 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 `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 of a pixel stream, specified as a scalar value representing intensity. Integer and fixed-point data types larger than 16 bits are not supported.

`double` and `single` data types are supported 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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 image pixel

scalar

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

`double` and `single` data types are supported 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

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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

`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.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

See Also

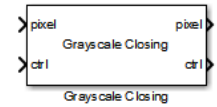
Blocks
Frame To Pixels | Gamma Correction

Objects
`visionhdl.GammaCorrector`

Introduced in R2015a

Grayscale Closing

Morphological closing of grayscale pixel data
Library: 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 a scalar that represents grayscale intensity.

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

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

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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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**.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixdt(0,N,M)`

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

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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 `1`s and `0`s

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The block supports flat neighborhoods of up to 32-by-32 pixels. To use a structuring element, specify the **Neighborhood** as `getnhood(strel(`*shape*`))`. 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 single 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 allocates (*neighborhood lines* – 1)-by-**Line buffer size** memory locations to store the pixels.

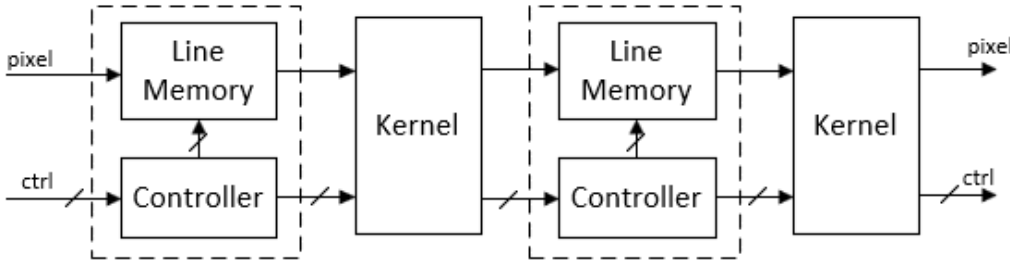
Algorithms

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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL code for this block if it is inside a Resettable Synchronous Subsystem.

See Also

Closing | Frame To Pixels | Grayscale Dilation | Grayscale Erosion | `visionhdl.GrayscaleClosing`

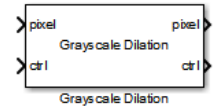
Topics

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

Introduced in R2016a

Grayscale Dilation

Morphological dilation of grayscale pixel data
Library: 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 a scalar that represents grayscale intensity.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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**.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixdt(0,N,M)`

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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 `1`s and `0`s

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The block supports flat neighborhoods of up to 32-by-32 pixels. To use a structuring element, specify the **Neighborhood** as `getnhood(strel(`*shape*`))`. 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 single horizontal line.

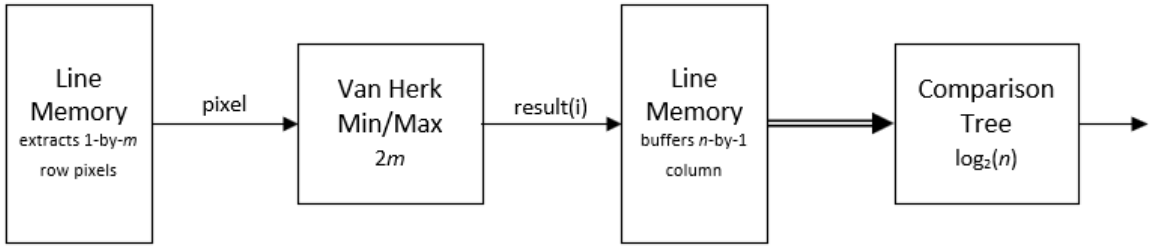
If you specify a value that is not a power of two, the block uses the next largest power of two. The block allocates (*neighborhood lines* – 1)-by-**Line buffer size** memory locations to store the pixels.

Algorithms

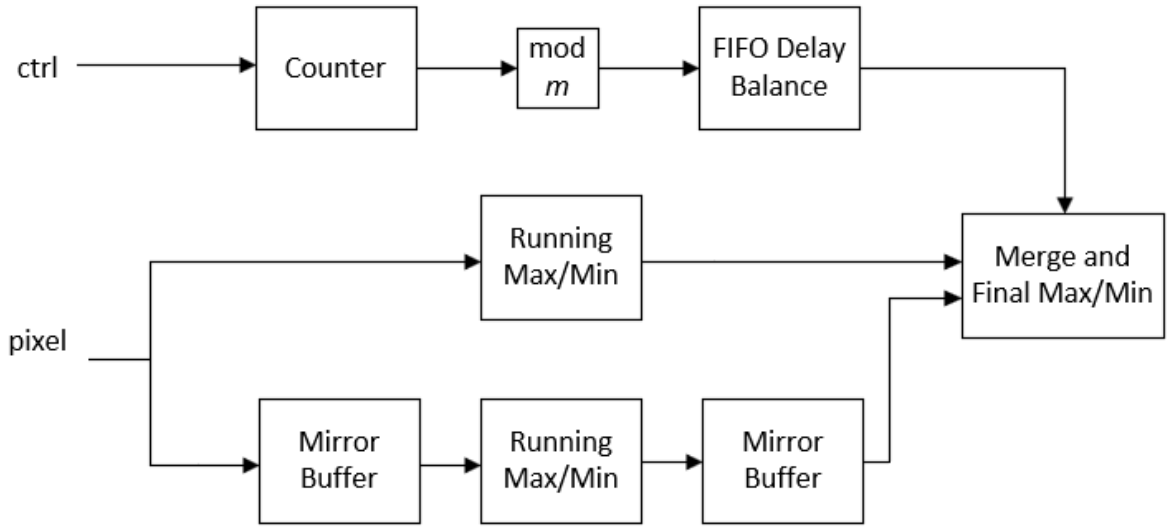
The block pads the image with zeros for the dilation operation. For more information, see “Edge Padding”.

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.



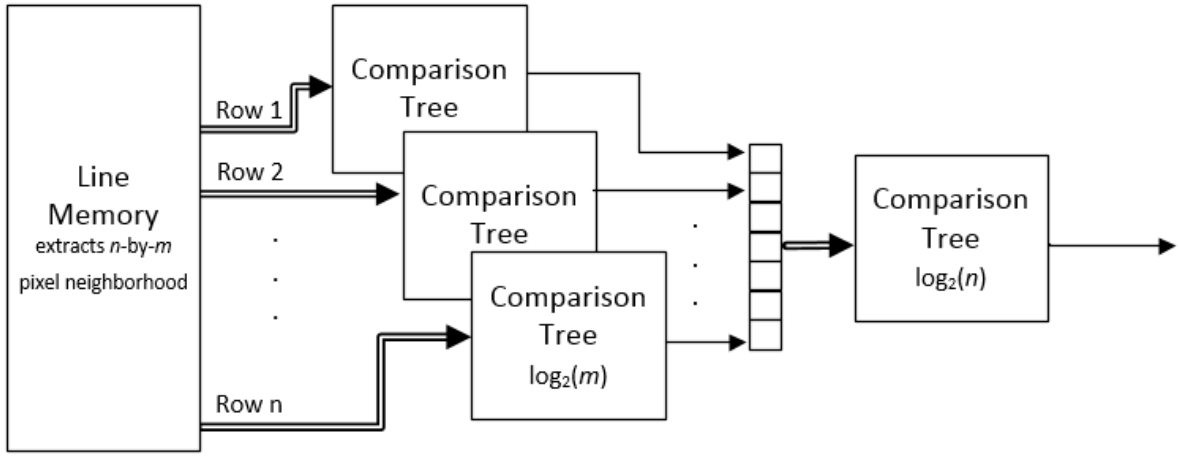
The Van Herk kernel computes a running forward maximum and a running backward maximum of the pixel values in each row of the neighborhood. For this computation, the pixels in the row must be buffered and the order reversed. The buffer adds latency relative to the comparison tree implementation. The Mirror 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.



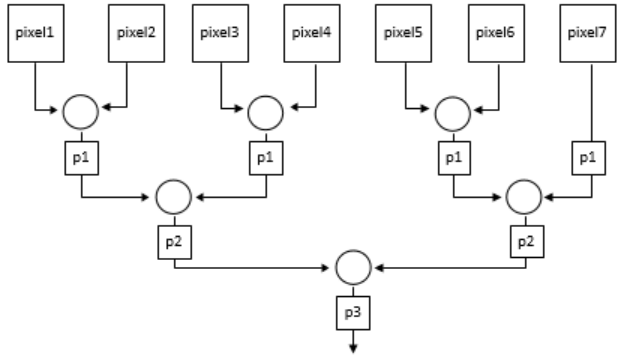
Comparison Tree Implementation

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

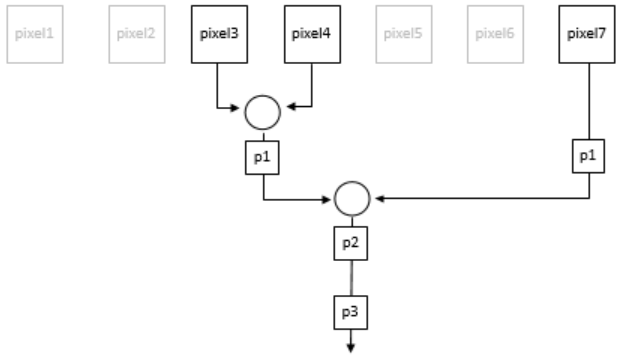
The diagram shows the architecture of the dilation operation. The algorithm finds the maximum pixel value of each row of the neighborhood in parallel. Then it calculates the maximum pixel value of the rows using another comparison tree. The diagram indicates the latency of each computation block.



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.

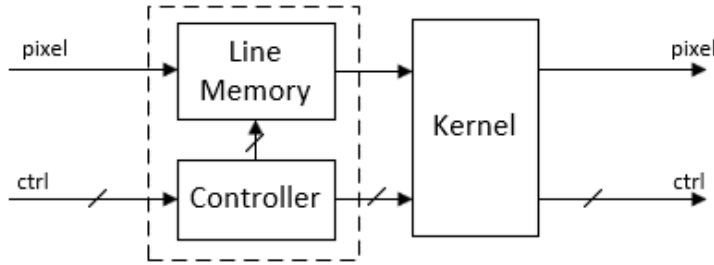


Latency

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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL code for this block if it is inside a Resettable Synchronous Subsystem.

See Also

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

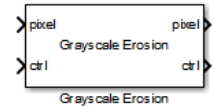
Topics

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

Introduced in R2016a

Grayscale Erosion

Morphological erosion of grayscale pixel data
Library: 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 a scalar that represents grayscale intensity.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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**.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixdt(0,N,M)`

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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 `1`s and `0`s

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The block supports flat neighborhoods of up to 32-by-32 pixels. To use a structuring element, specify the **Neighborhood** as `getnhood(strel(`*shape*`))`. 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 single horizontal line.

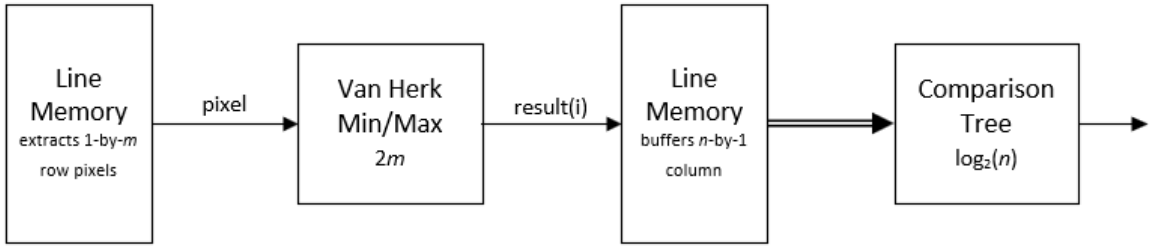
If you specify a value that is not a power of two, the block uses the next largest power of two. The block allocates (*neighborhood lines* – 1)-by-**Line buffer size** memory locations to store the pixels.

Algorithms

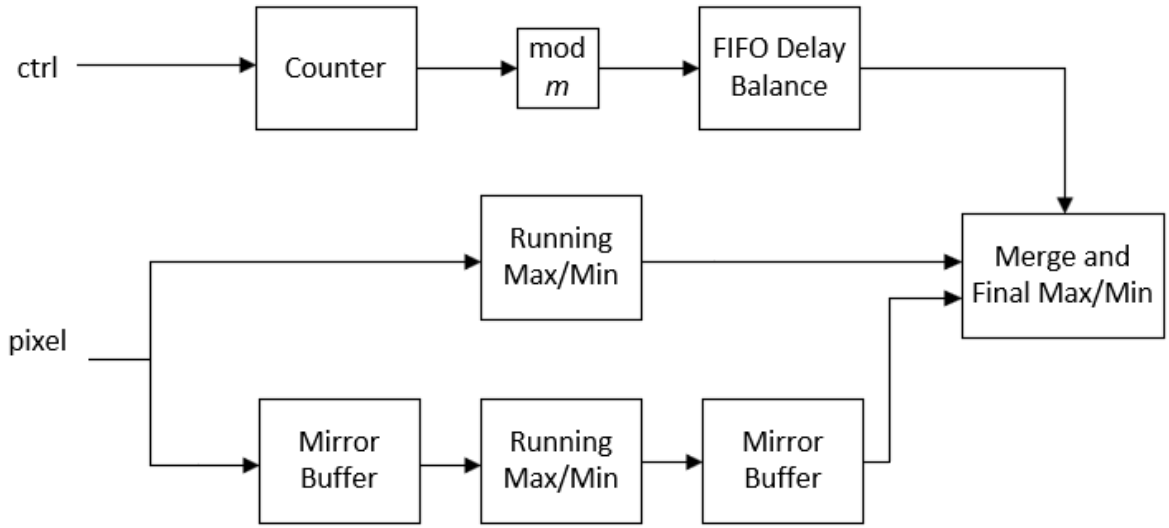
The block pads the image with ones for the erosion operation. For more information, see “Edge Padding”.

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.



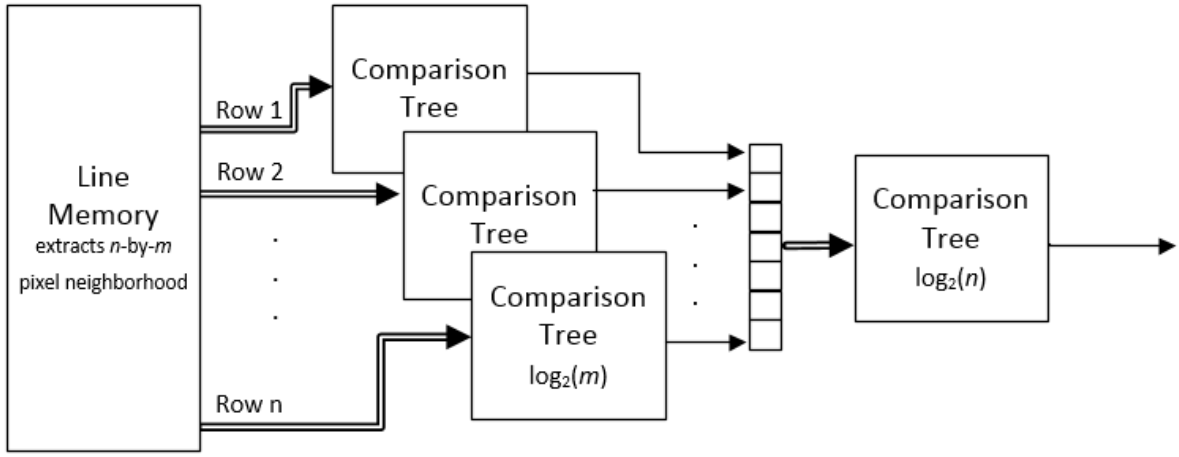
The Van Herk kernel computes a running forward minimum and a running backward minimum of the pixel values in each row of the neighborhood. For this computation, the pixels in the row must be buffered and the order reversed. The buffer adds latency relative to the comparison tree implementation. The Mirror 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.



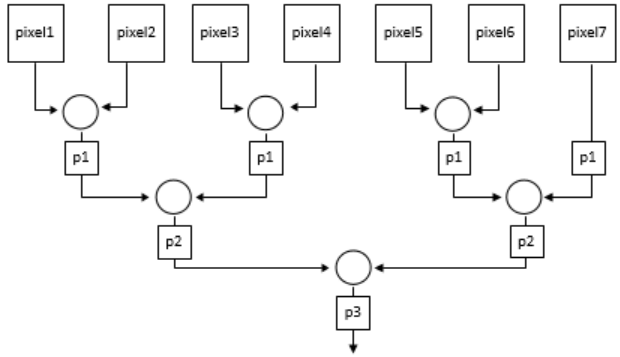
Comparison Tree Implementation

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

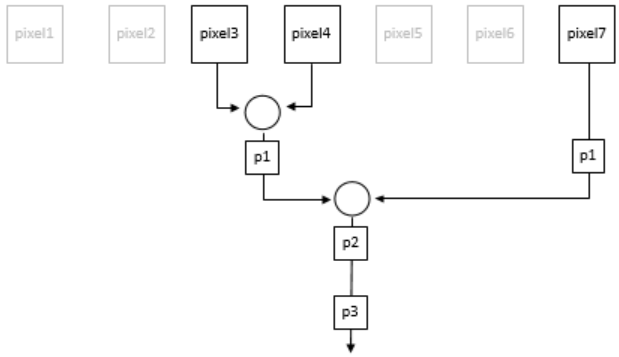
The diagram shows the architecture of the erosion operation. The algorithm finds the minimum pixel value of each row of the neighborhood in parallel. Then it calculates the minimum pixel value of the rows using another comparison tree. The diagram indicates the latency of each computation block.



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.

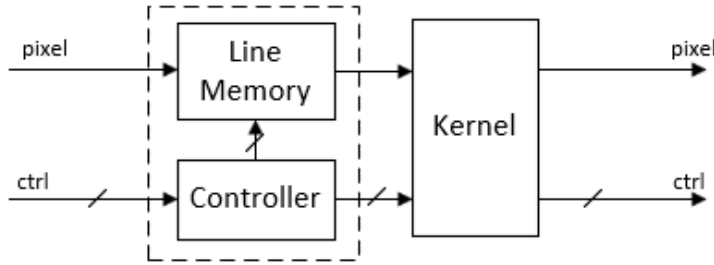


Latency

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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL code for this block if it is inside a Resettable Synchronous Subsystem.

See Also

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

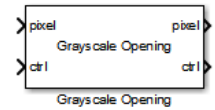
Topics

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

Introduced in R2016a

Grayscale Opening

Morphological opening of grayscale pixel data
Library: 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 a scalar that represents grayscale intensity.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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**.

Data Types: `single` | `double` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixdt(0,N,M)`

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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 `1`s and `0`s

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The block supports flat neighborhoods of up to 32-by-32 pixels. To use a structuring element, specify the **Neighborhood** as `getnhood(strel(`*shape*`))`. 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 single 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 allocates (*neighborhood lines* – 1)-by-**Line buffer size** memory locations to store the pixels.

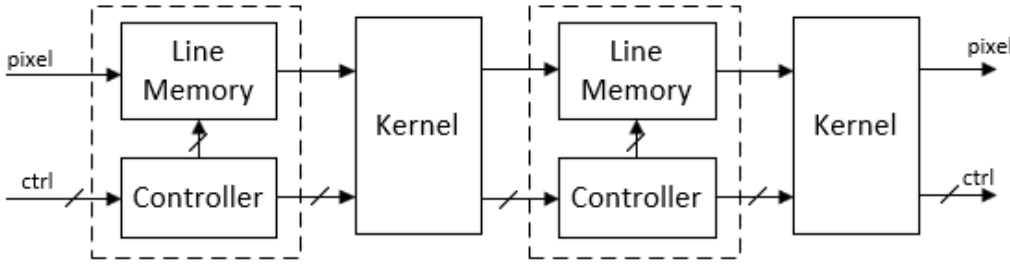
Algorithms

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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL code for this block if it is inside a Resettable Synchronous Subsystem.

See Also

Frame To Pixels | Grayscale Dilation | Grayscale Erosion | Opening | `visionhdl.GrayscaleOpening`

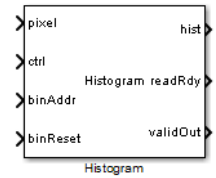
Topics

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

Introduced in R2016a

Histogram

Frequency distribution of pixel values in video stream
Library: 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` — Single image pixel

scalar

Single image pixel, specified as an unsigned integer scalar value.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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`

`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,N,0)`, N = 5,6,...,12. The word length must be $\log_2$(**Number of bins**).

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

`binReset` — Reset histogram bin values

`Boolean` 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

`Boolean` 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
`Boolean` 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.

`double` and `single` data types are supported 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

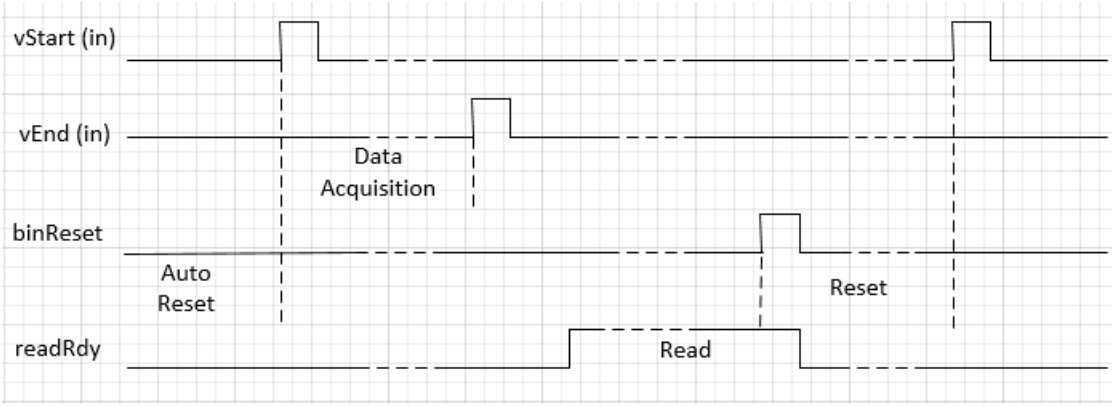
RAM Reset and Ready Sequence

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.

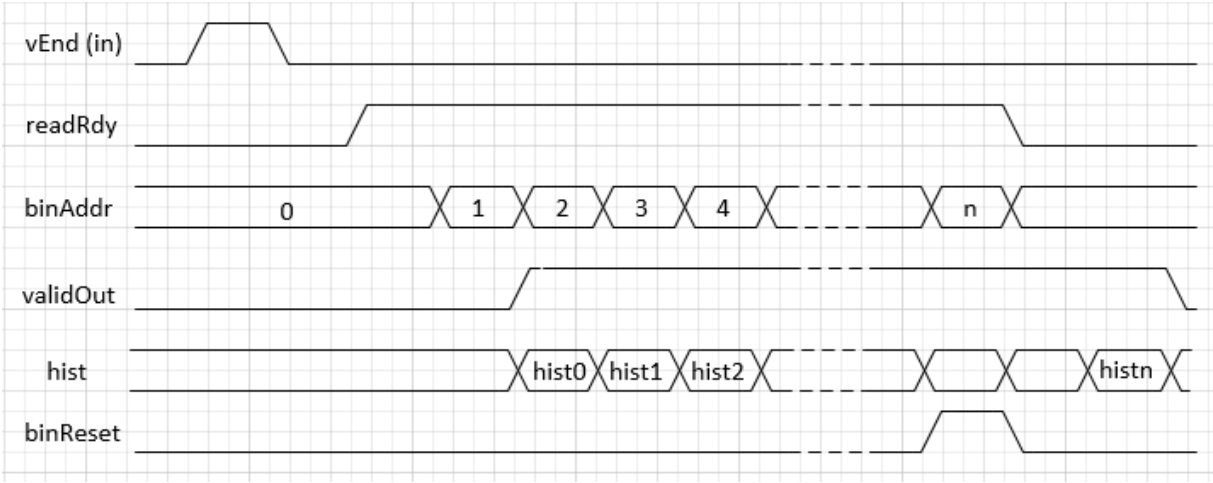
The histogram values persist and accumulate across frames until you assert **binReset**. At that time, the block takes **Number of bins** cycles to clear the RAM and be ready for new input. Other input signals are ignored during reset.

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.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

Blocks
2-D Histogram | Frame To Pixels

Objects
`visionhdl.Histogram`

Functions
`imhist`

Introduced in R2015a

Image Filter

2-D FIR filtering
Library: 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 4 or 8 pixels per clock cycle to support high-frame-rate or high-resolution formats. 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 four or eight 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 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 intensity value. For multipixel streaming, specify a vector of four or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see “Filter Multipixel Video Streams”.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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.

`double` and `single` data types are supported 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: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

Output

`pixel` — 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 input 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.

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

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

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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 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`

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`

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` — Add the padding pixels to mirror the edge of the image.

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

Select the rounding mode for internal fixed-point calculations as `Floor`, `Ceiling`, `Convergent`, `Nearest`, `Round`, and `Zero`. For more information about rounding modes, see “Rounding Modes” (DSP System Toolbox).

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

When this parameter is `off`, fixed-point and integer values wrap around to 0 when the value overflows what is representable with that data type. When this parameter is `on`, the value saturates at the maximum representable value.

`Coefficients` — Filter coefficients data type selection
`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 selection
`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).

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.

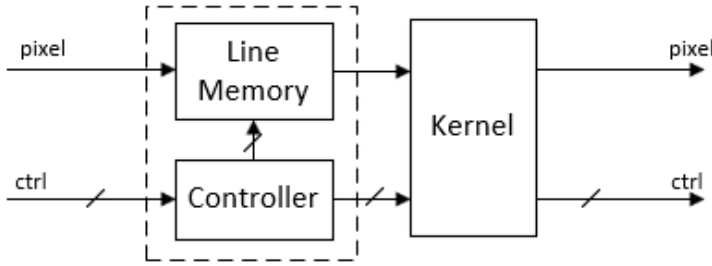
When you use multipixel streaming, the block uses a single line memory and implements *NumberOfPixels* filter kernels in parallel. This increase in hardware resources is a trade off for increasing throughput compared to single-pixel streaming.

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. 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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL code for this block if it is inside a Resettable Synchronous Subsystem.

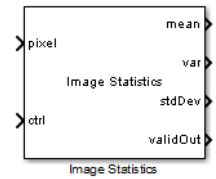
See Also

2-D FIR Filter | Frame To Pixels | `visionhdl.ImageFilter`

Introduced in R2015a

Image Statistics

Calculate mean, variance, and standard deviation
Library: 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` — Single image pixel
scalar

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

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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×*IWL*	2×*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` | `fixed point`

`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` | `fixed point`

`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` | `fixed point`

`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

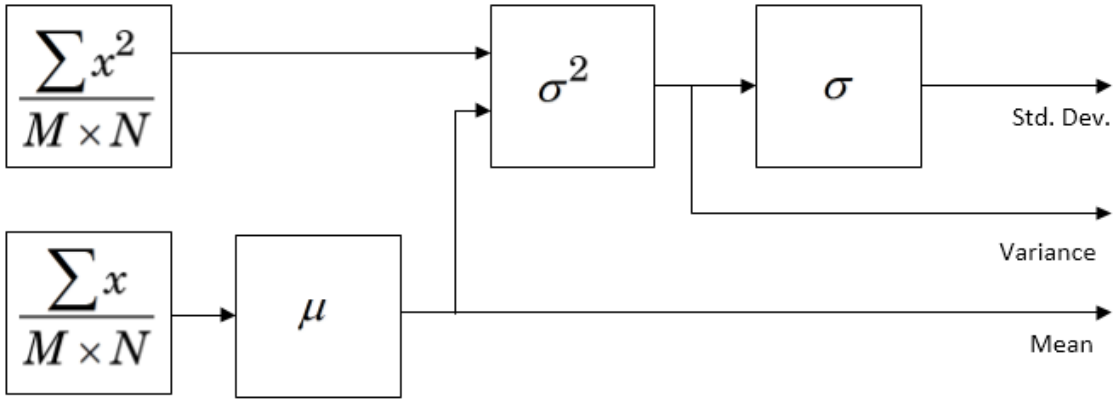
- To change the size and dimensions of the ROI, you can manipulate the input video stream control signals. See “Regions of Interest” on page 1-141.

- 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 1-137.
- 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.

$$\mu_{L_1} = \frac{1}{64}\sum_{n=1}^{64} x_n$$

Then it accumulates 64 of the previous windows, and normalizes.

$$\mu_{L_2} = \frac{1}{64}\sum_{n=1}^{64} \mu_{nL_1}$$

A third accumulator sums 64 of the 64×64 windows, and normalizes the same way.

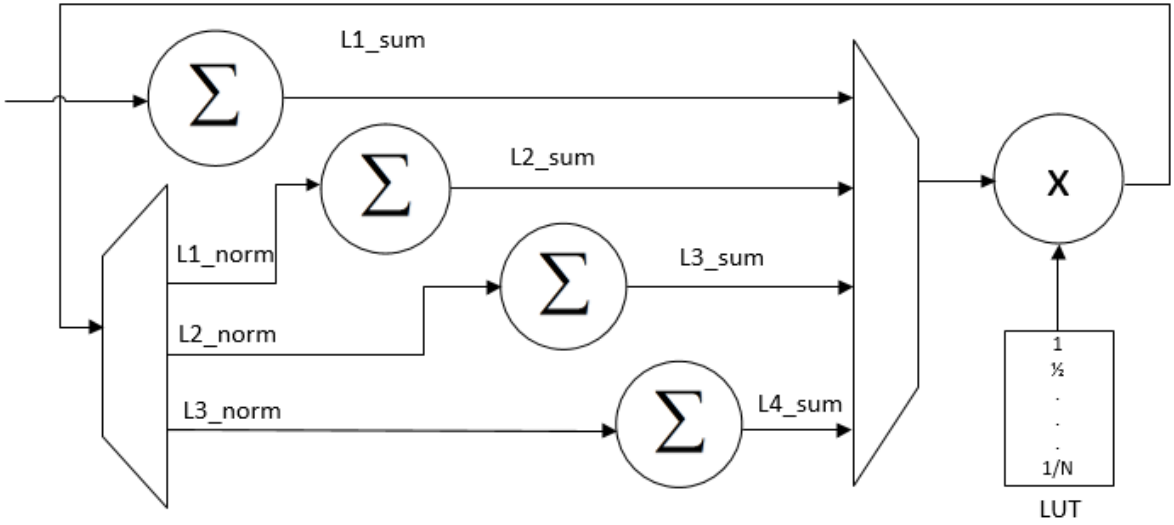
$$\mu_{L_3} = \frac{1}{64}\sum_{n=1}^{64} \mu_{nL_2}$$

The fourth accumulator sums 64 of the 64×64×64 windows and normalizes.

$$\mu_{L_4} = \frac{1}{64}\sum_{n=1}^{64} \mu_{nL_3}$$

Each valid pixel is accumulated as it arrives. Its location within a line or frame does not affect the accumulation logic.

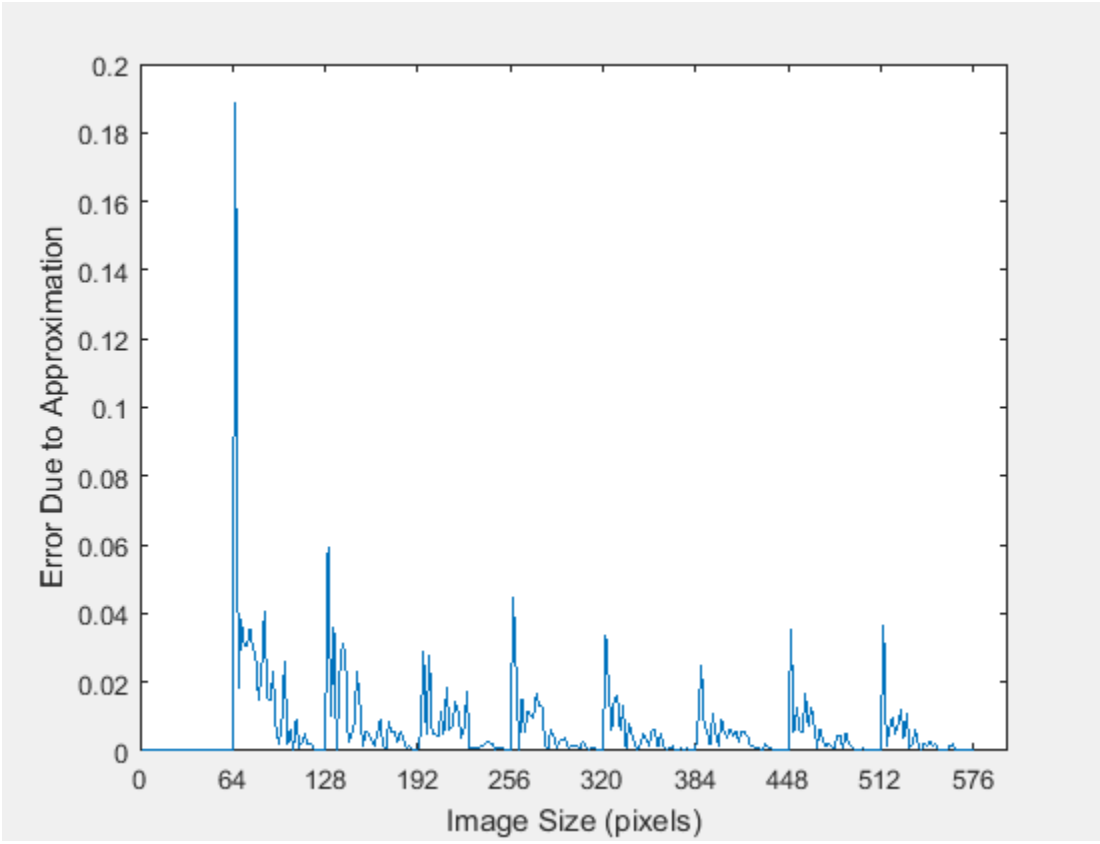
When `vEnd` is received, the block promotes any remaining data in the four levels of mean calculation to calculate the final output. If an accumulator counter is not at 64 when `vEnd` arrives, that level normalizes by the actual value of the counter. The constants for this multiplication are stored in 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)
✓	✓	✓		[4*n*]+4+IWL, (where *n* is the number of accumulator levels required for the input size)
✓			variance and square root	[4*n*]
	✓		square root	[4*n*]+4
		✓		[4*n*]+4+IWL
✓	✓		square root	[4*n*]+4
✓		✓		[4*n*]+4+IWL
	✓	✓		[4*n*]+4+IWL

Note There must be at least 16 cycles between the `vEnd` signals on the input. This timing restriction enables the block to finish processing the current frame before the new one arrives.

If you are using a custom video format, set the horizontal blanking interval by using the parameters of the Frame To Pixels block. The horizontal blanking interval is equal to **Total pixels per line** – **Active pixels per line**, or, equivalently, **Front porch** + **Back porch**. Standard streaming video formats use a horizontal blanking interval of about 25% of the frame width. This interval is much larger than the latency of the statistics operations.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

See Also

2-D Mean | 2-D Standard Deviation | 2-D Variance | Frame To Pixels | `visionhdl.ImageStatistics`

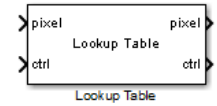
Topics

"Multi-Zone Metering"

Introduced in R2015a

Lookup Table

Map input pixel to output pixel using custom rule
Library: Vision HDL Toolbox / Conversions



Description

The Lookup Table block provides a custom one-to-one map between input pixel values and output 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 of a pixel stream, specified as a scalar value representing intensity. Integer and fixed-point data types larger than 16 bits are not supported.

`double` and `single` data types are supported 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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 image pixel

scalar

Single image pixel of a pixel stream, returned as a scalar value representing intensity. The data type is specified by the **Table data** parameter.

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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

`Table data` — Map between input pixel values and output pixel values

`uint8(0:1:255)` (default) | vector

This mapping determines the one-to-one correspondence between an input **pixel** value and an output **pixel** value.

- The table data is a row or column vector of any data type. The data type of the table data determines the data type of the output **pixel**.

- 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, and so on.

Algorithms

Latency

The latency of the Lookup Table block is 2 cycles.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

See Also

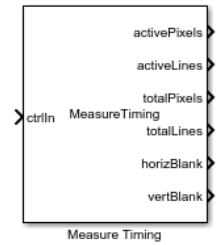
Blocks
Frame To Pixels

Objects
`visionhdl.LookupTable`

Introduced in R2015a

Measure Timing

Measure timing of pixel control bus input
Library: Vision HDL Toolbox / Utilities

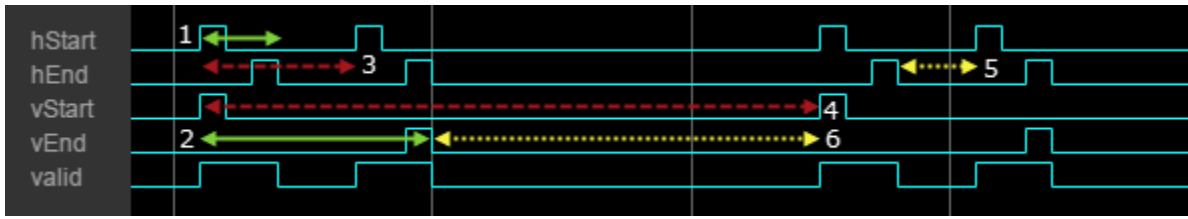


Description

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. 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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 `vStart` 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`

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”

Introduced in R2016b

Median Filter

2-D median filtering

Library: 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 4 or 8 pixels per clock cycle to support high-frame-rate or high-resolution formats. 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 four or eight 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 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 intensity value. For multipixel streaming, specify a vector of four or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see “Filter Multipixel Video Streams”.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 or multipixel vector

scalar | vector

This block supports single pixel streaming or multipixel streaming. When using single pixel streaming, the block returns a single input 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.

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

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

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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 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`

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`.

`Padding method` — Method for padding boundary of input image
`Symmetric` (default) | `Constant` | `Replicate`

Select one of these methods for padding the boundary of the input image.

- `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` — Pad the input matrix with its mirror image.

For more information on these methods, see “Edge 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 single horizontal line.

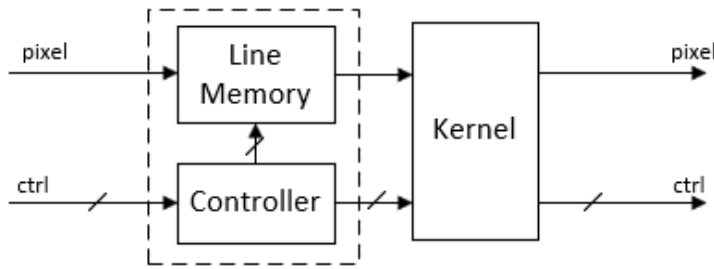
If you specify a value that is not a power of two, the block uses the next largest power of two. The block allocates (*N*— 1) -by-**Line buffer size** memory locations to store the pixels used to compute the median value. *N* is the dimension of the square region specified by the **Neighborhood size** parameter.

Algorithms

When you use multipixel streaming, the block uses a single line memory and implements *NumberOfPixels* filter kernels 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. 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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

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.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL code for this block if it is inside a Resettable Synchronous Subsystem.

See Also

Frame To Pixels | Median Filter | `visionhdl.MedianFilter`

Introduced in R2015a

Opening

Morphological opening of binary pixel data
Library: 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 `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”.

Ports

Input

`pixel` — Single input pixel

Boolean scalar

Single input pixel, specified as a Boolean scalar.

Data Types: `Boolean`

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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

Boolean scalar

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

Data Types: `Boolean`

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, returned as a `pixelcontrol` bus that 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

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

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The block supports neighborhoods of up to 32-by-32 pixels. To use a structuring element, specify **Neighborhood** as `getnhood(strel(`*shape*`))`.

`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 single horizontal line.

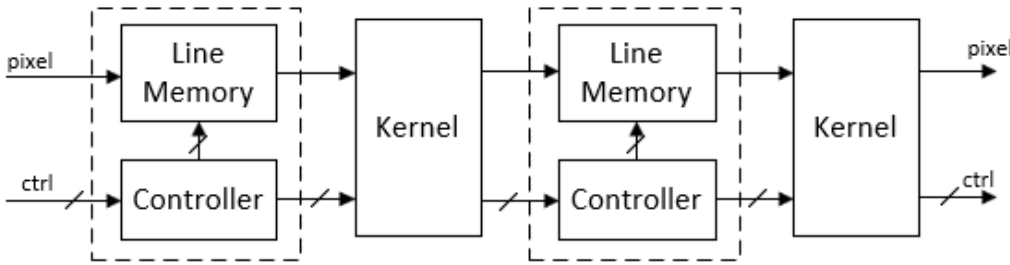
If you specify a value that is not a power of two, the block uses the next largest power of two. The block allocates (*neighborhood lines* – 1)-by-**Line buffer size** memory locations to store the pixels.

Algorithms

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. 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 The horizontal blanking interval of the pixel stream format must be greater than the latency of the block. This interval enables the block to finish processing one line before it starts processing the next one. 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. 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**.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

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

Restrictions

You cannot generate HDL code for this block if it is inside a Resettable Synchronous Subsystem.

See Also

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

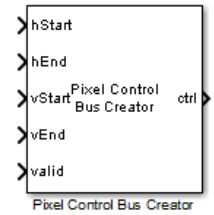
Topics

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

Introduced in R2015a

Pixel Control Bus Creator

Create control signal bus for use with Vision HDL Toolbox blocks
Library: 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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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`

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 a single, default HDL architecture.

HDL Block Properties

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

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

See Also

Frame To Pixels | Pixel Control Bus Selector | Pixels To Frame

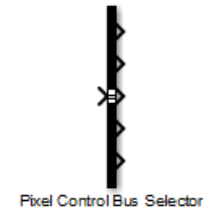
Topics

“Streaming Pixel Interface”

Introduced in R2015a

Pixel Control Bus Selector

Select signals from control signal bus used by Vision HDL Toolbox blocks
Library: 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

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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`

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 a single, default HDL architecture.

HDL Block Properties

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

See Also

Frame To Pixels | Pixel Control Bus Creator | Pixels To Frame

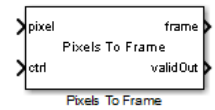
Topics

“Streaming Pixel Interface”

Introduced in R2015a

Pixels To Frame

Convert pixel stream to frame-based video
Library: 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 scalar streaming, or multipixel streaming. Multipixel streaming accepts 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 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` — Image pixel or multipixel vector

scalar | vector

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

Number of pixels-by-1 pixel intensity values. You cannot set **Number of components** greater than 1 when you set **Number of pixels** greater than 1.

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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.

`validOut` — Indicates when output frame is ready

`Boolean` 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) | 3 | 4

Number of values used to represent each pixel. Each pixel can have 1, 3, or 4 components. Set this parameter to `1` for grayscale video. Set this parameter to `3` for color video, for example, {R,G,B} or {Y,Cb,Cr}. Set this parameter to `4` to use color with an alpha channel for transparency.

Dependencies

When **Number of pixels** is greater than 1, you must set **Number of components** to `1`.

`Number of pixels` — Number of pixels streamed per time step

1 (default) | 4 | 8

Number of pixels transferred on the streaming interface for each time step, specified as `1`, `4`, or `8`. To enable multipixel streaming and increase throughput for high-resolution or high-frame-rate video, set this parameter to `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

Dependencies

When **Number of pixels** is greater than 1, you must set **Number of components** to `1`.

`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

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”

Introduced in R2015a

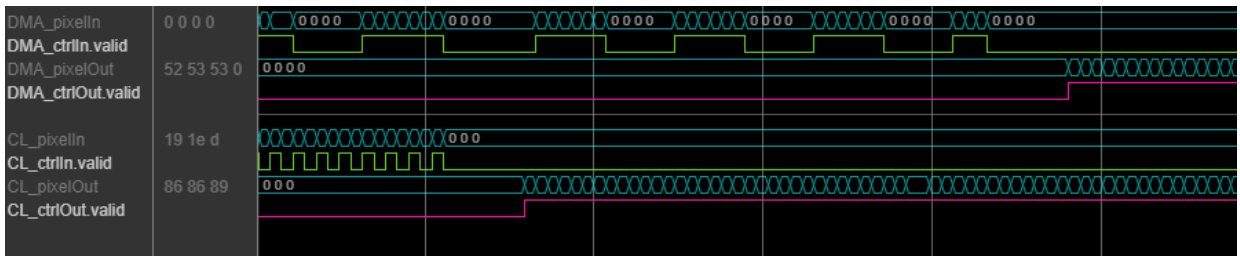
Pixel Stream FIFO

Buffer input stream to create image lines that have contiguous valid pixels
Library: 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 `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 | vector

Single image pixel in a pixel stream, specified as a scalar value that represents intensity, or a vector of 1-by-N color components.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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 image pixel

scalar | vector

Single image pixel in a pixel stream, returned as a scalar value representing intensity, or a vector of 1-by-N color components. The output pixel data type is the same as the data type of the input pixels.

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

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

`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

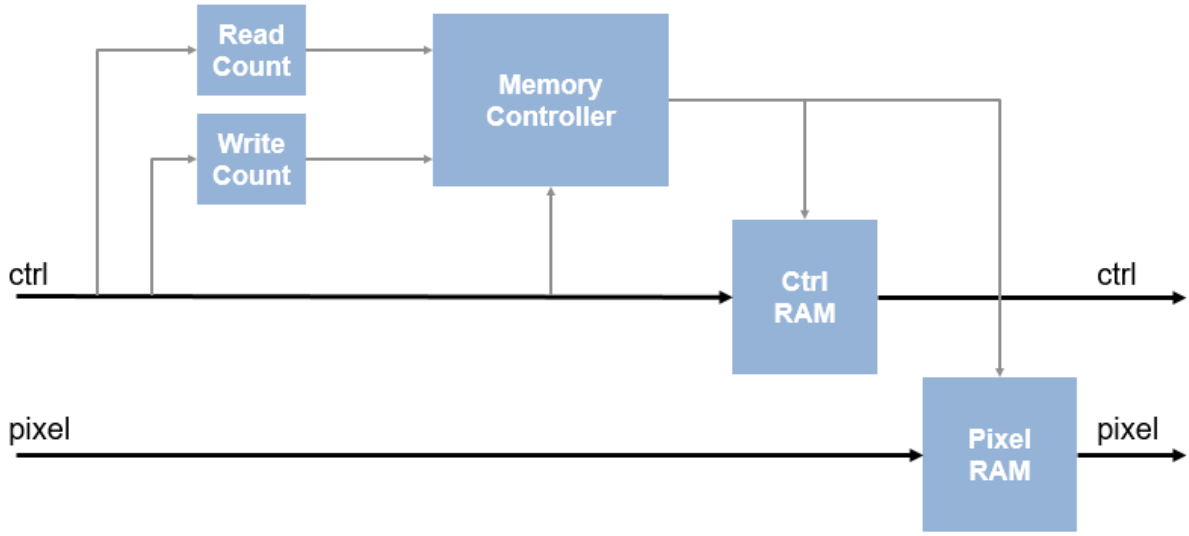
`Line buffer size` — Size of line memory buffer
`2048` (default) | positive integer

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 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.



Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem, or a Triggered Subsystem.

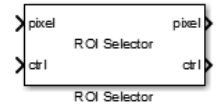
See Also

Blocks
Frame To Pixels | Line Buffer

Introduced in R2018a

ROI Selector

Select a region of interest (ROI) from pixel stream
Library: Vision HDL Toolbox / Utilities



Description

The ROI Selector block selects a portion of the active frame from a video stream. The total size of the frame remains the same. The control signals indicate a new active region of the frame. You can specify a fixed size and location for the new frame, or select the frame location dynamically by using an input port. The diagram shows the inactive pixel regions in blue and the requested output region outlined in orange.

You can select more than one region. Define each region by the upper-left corner coordinates and the dimensions. 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. The diagram shows the output frames for three requested regions. The second output region does not include the inactive region above the frame.

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”.

Ports

Input

`pixel` — Single image pixel

scalar

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

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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`

`region1,...,regionN` — Region of interest

1-by-4 vector of positive integers

Region of interest, specified as a vector of positive integers that define the coordinates of the top-left corner and the dimensions of each desired output frame, `[hPos vPos hSize vSize]`. The block has *N* region ports, where *N* is the **Number of regions**.

Dependencies

To enable this port, set **Regions source** to `Input port`.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output

`pixel1,...,pixelN` — Output image pixels

scalar

Output image pixels, returned as scalar values. The block has *N* output pixel ports, where *N* is the **Number of regions**, or the size of the **Regions** matrix. The output data type is the same data type as the input **pixel**.

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

`ctrl1,...,ctrlN` — Control signals

`pixelcontrol` bus

Control signals, returned as a `pixelcontrol` bus that contains five signals. The block has *N* output control ports, where *N* is the **Number of regions** or the size of the **Regions** matrix.

Data Types: `bus`

Parameters

`Regions source` — Location of output region definitions

`Property` (default) | `Input port`

Specifies the location of the output region definitions. You can set **Regions source** to one of the following values:

- `Property` — Specify the regions by using the **Regions** mask parameter.
- `Input port` — Specify the regions by using input ports. There is one input port for each region. The block samples the **region** input ports when `vStart` is `true` in the input control bus.

`Regions` — Rectangular regions of interest to select from the input frame

N-by-4 vector

Rectangular regions of interest to select from the input frame, specified as a *N*-by-4 matrix.

N is the number of regions. You can select up to 16 regions. The four elements that define each region are the top-left starting coordinates and the dimensions `[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. The regions are independent of each other, so they can overlap.

Dependencies

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

`Number of regions` — Number of region input ports

integer from 1 to 16

Number of region input ports, specified as an integer from 1 to 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. It does not use additional counters for additional regions.

Latency

The block has a latency of three cycles.

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

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

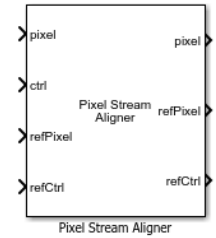
See Also

Frame To Pixels | `visionhdl.ROISelector`

Introduced in R2016a

Pixel Stream Aligner

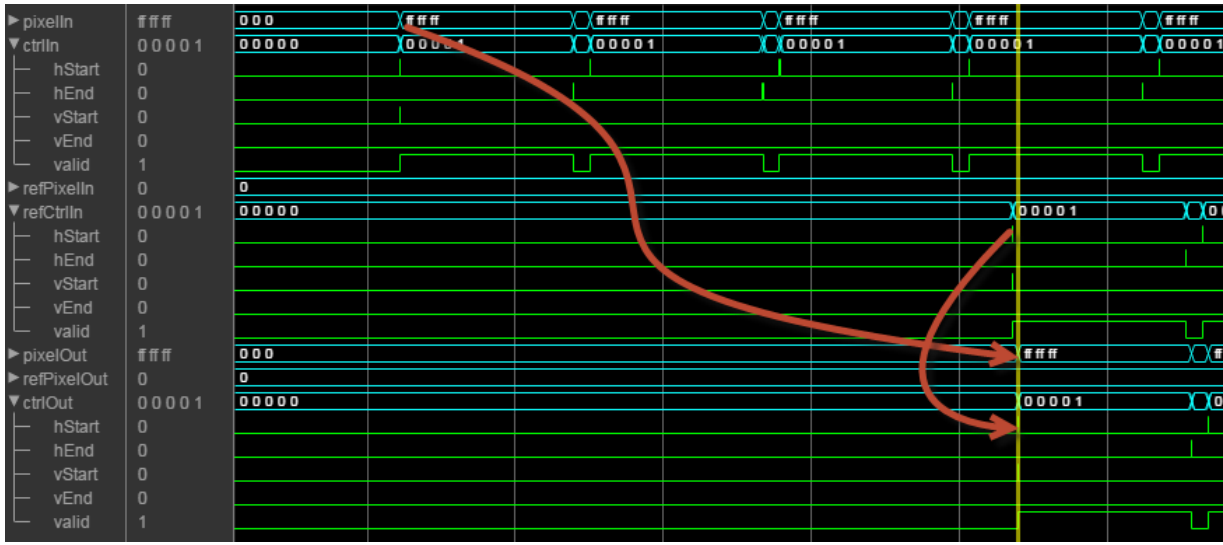
Align two streams of pixel data
Library: Vision HDL Toolbox / Utilities



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. There is a 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 *MaxNumberOfLines* must be set to at least 4.



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

Single image pixel of a pixel stream, specified as a vector of three values representing R'G'B' or Y'CbCr, or a scalar value representing intensity. The block delays this pixel stream to match the control signals of the reference stream, **refPixel**. Therefore, **pixel** must be the earlier of the two streams.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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`

`refPixel` — Reference pixel stream

scalar | vector

Single image pixel of a pixel stream, specified as a vector of three values representing R'G'B' or Y'CbCr, or a scalar value representing intensity. The block delays the **pixel** input stream to match the reference control signals. Therefore, **refPixel** must be the later of the two streams. The reference data and its control signals pass through the block with a small delay.

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

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

`refCtrl` — Reference control signals, accompanying reference pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that contains five signals. The signals describe the validity of the pixel and its location in the frame. For more information, see “Pixel Control Bus”.

The block uses these control signals for the aligned output stream.

Data Types: `bus`

Output

`pixel` — Aligned pixel stream

scalar | vector

Single image pixel, returned as a vector of three values representing R'G'B' or Y'CbCr, or a scalar value representing intensity. The data type is the same as the data type of the **pixel** input stream.

`refPixel` — Reference pixel stream

scalar | vector

Single image pixel, specified as a vector of three values representing R'G'B' or Y'CbCr, or a scalar value representing intensity. The data type is the same as the data type of the **refPixel** input stream. This stream is passed through the block unchanged.

`refCtrl` — Reference control signals

`pixelcontrol` bus

The input **refCtrl** signals pass through the block with a small delay.

Parameters

`Line buffer size` — Size of the line memory buffer

2048 (default) | scalar 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 $MaxNumberOfLines + \log_2(LineBufferSize)$.

`Maximum number of lines` — Buffer depth that accommodates the timing offset between input streams

10 (default) | scalar integer

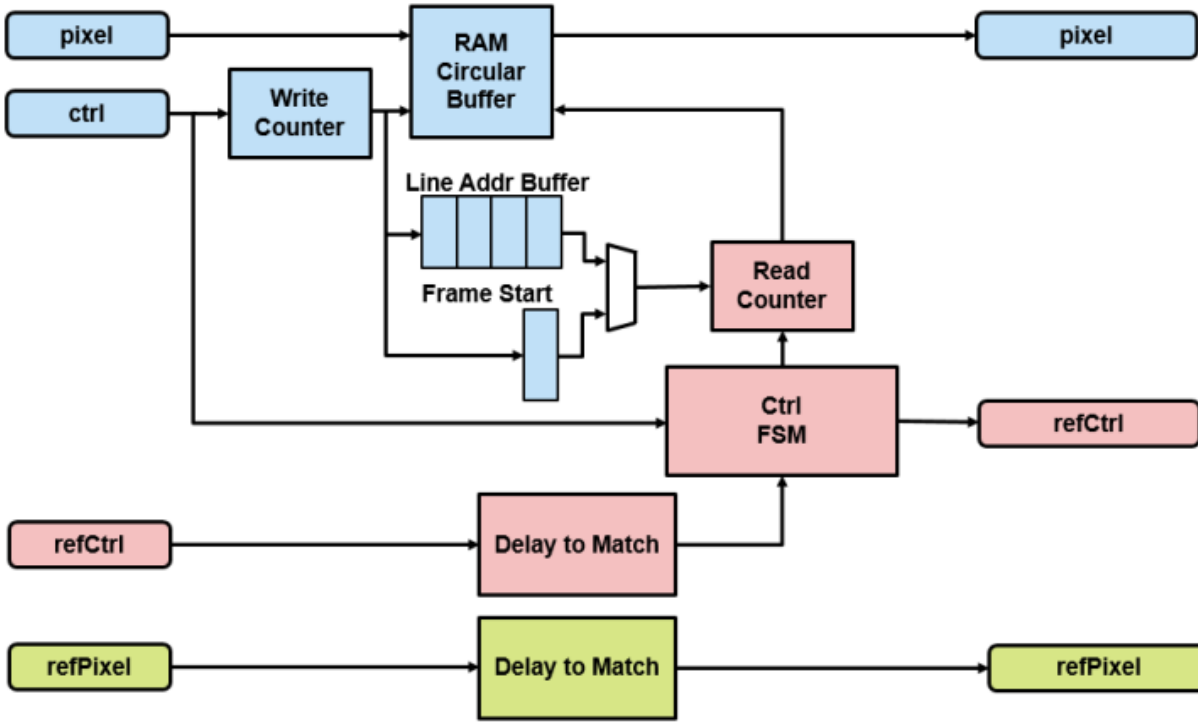
The block implements a circular buffer of 2^M pixels, where M is $MaxNumberOfLines + \log_2(LineBufferSize)$, and a line address buffer of *MaxNumberofLines* locations. The circular memory stores the earlier input lines until the reference control signals arrive. The line address buffer stores the address of the start of each line. When the reference control signals arrive, the 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 internal latency before the block reads the first line. During simulation, the block warns when an overflow occurs. To avoid the overflow condition, increase *MaxNumberofLines*. The delay between streams cannot exceed an entire frame.

Algorithms

The block stores the data from the **pixel** port to a circular buffer, and 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 *RAM Circular Buffer* is a memory of 2^M pixels, where M is *MaxNumberOfLines* + $\log_2$(*LineBufferSize*). The *Line Address Buffer* has *MaxNumberOfLines* 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 your output. In this diagram, *MaxNumberOfLines* is four, so there are four locations to store a line address in the buffer. The diagram also shows a single location to store `ctrl.Vstart`. The delay between the two streams must be less than the time between frame starts.



Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

Restrictions

You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

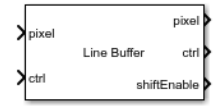
Blocks
Frame To Pixels

Objects
`visionhdl.PixelStreamAligner`

Introduced in R2017a

Line Buffer

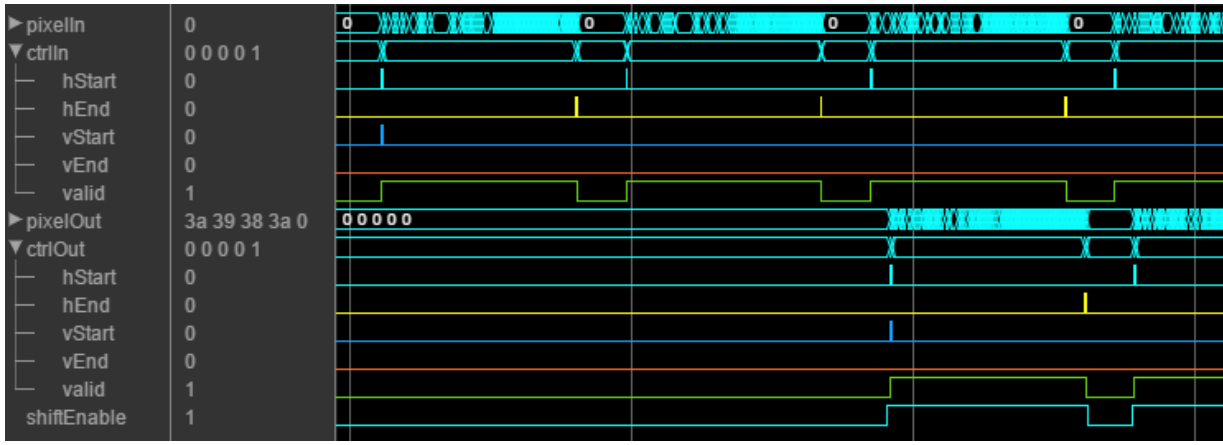
Store video lines and return neighborhood pixels
Library: Vision HDL Toolbox / Utilities



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 4 or 8 pixels per clock cycle to support high-frame-rate or high-resolution formats. 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 four or eight 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 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 intensity value. For multipixel streaming, specify a vector of four or eight pixel intensity values. For details of how to set up your model for multipixel streaming, see “Filter Multipixel Video Streams”.

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

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

`ctrl` — Control signals associated with pixel stream

`pixelcontrol` bus

Control signals associated with the pixel stream, specified as a `pixelcontrol` bus that 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-*M* vector, where *M* is the vertical neighborhood dimension. When using multipixel streaming, the block returns a *NumberOfPixels*-by-*M* matrix, where *NumberOfPixels* is the size of the input **pixel** 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 *NumberOfPixels* regions, each of *N*-by-*M* pixels, and then perform the kernel operation *NumberOfPixels* 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 *N*-by-*M* neighborhood directly. You can :

- Separate a filter into vertical and horizontal components.

- Concatenate the neighborhood pixels into an (*N***M*)-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: `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `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(`*`M`*`/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

`Boolean` 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 [*vertical horizontal*]. The block returns a column vector of *vertical* elements. The *horizontal* dimension is used to determine padding.

`Padding method` — Method for padding the boundary of input image

`Symmetric` (default) | `Constant` | `Replicate`

Method for padding the boundary of the input image.

- `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` — Pad the input matrix with its mirror image.

For more information on these methods, see “Edge 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

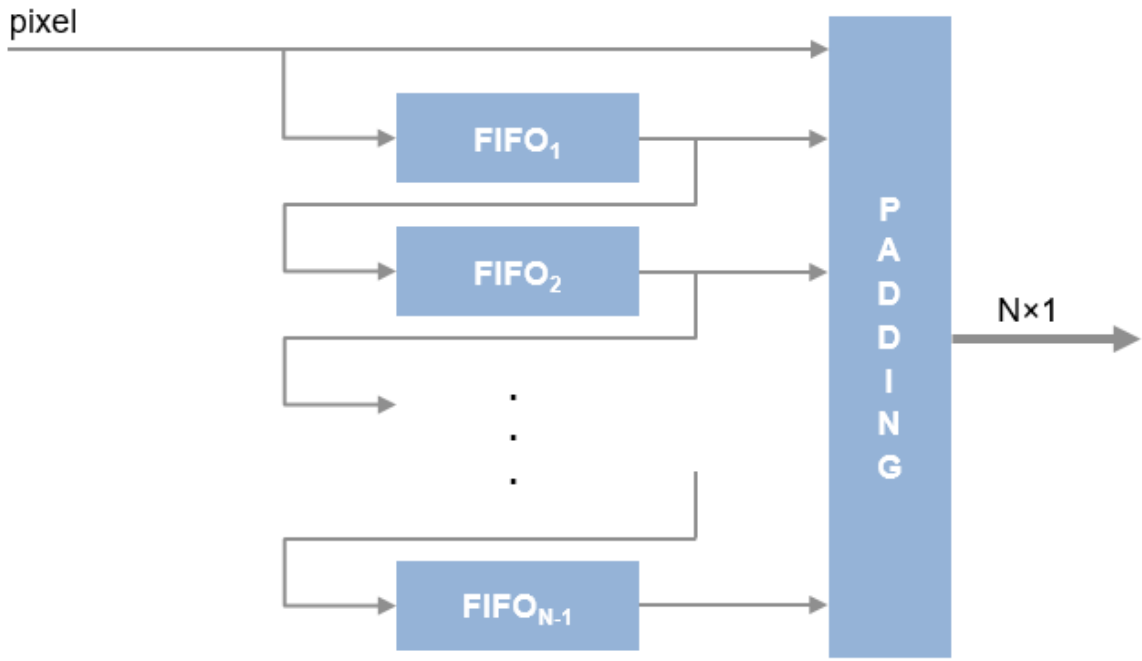
`2048` (default) | positive integer

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 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 block stores M – 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(`M`/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

Extended Capabilities

C/C++ Code Generation

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

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

HDL Code Generation

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

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

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

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

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

See Also

Blocks
Frame To Pixels

Objects
`visionhdl.LineBuffer`

Introduced in R2017b

System Objects — Alphabetical List

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? (MATLAB).

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 (MATLAB).

`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` — Intensity standard deviation target

`'Constant'` (default) | `'Replicate'` | `'Symmetric'`

Method for padding the boundary of the input image.

- `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` — Pad the input matrix with its mirror image.

For more information on 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 single horizontal line.

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

`RoundingMethod` — Rounding mode used for fixed-point operations
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding mode used for fixed-point operations. When the input is any integer or fixed-point data type, the algorithm uses fixed-point arithmetic for internal calculations. This option 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(signed,WL,FL)`

Data type for the output pixels, specified as `numerictype(signed,WL,FL)`, where *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. The object accepts and returns 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` — Single image pixel

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.

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

Data Types: `uint8` | `uint16` | `int8` | `int16` | `fi` | `logical` | `double` | `single`

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see "Pixel Control Structure".

Data Types: `struct`

Output Arguments

`pixelOut` — Single image pixel

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.

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

Data Types: `uint8` | `uint16` | `int8` | `int16` | `fi` | `logical` | `double` | `single`

`ctrlOut` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying the 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

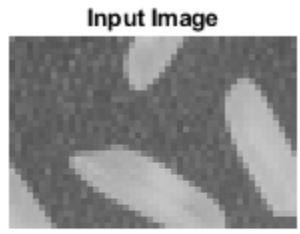
Examples

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);
```



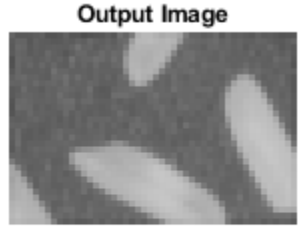
Write a function that creates and calls the System object™. You can generate HDL from this function.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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
```



See Also

Blocks
Bilateral Filter

Objects
`visionhdl.FrameToPixels`

Introduced in R2017b

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? (MATLAB).

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 (MATLAB).

`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 “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. The object accepts and returns 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` — Single image pixel

scalar

Single image pixel in a pixel stream, specified as a scalar value representing intensity.

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

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

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Single image pixel

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: `uint8` | `uint16` | `uint32` | `int8` | `int16` | `int32` | `fi` | `logical` | `double` | `single`

`ctrlOut` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying the 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

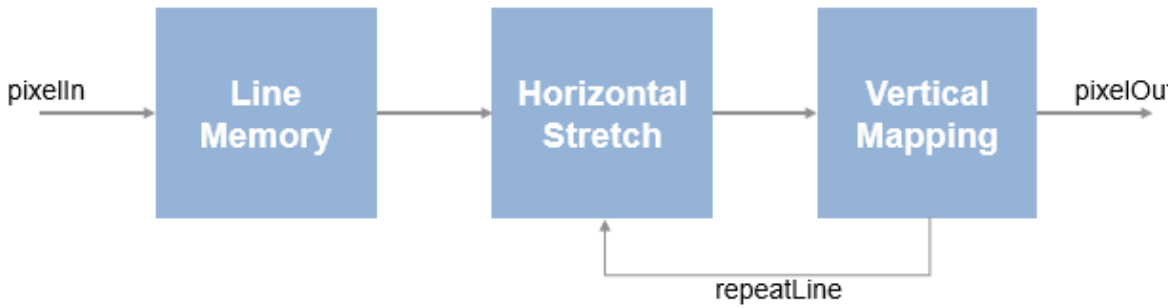
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

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.

See Also

Blocks
Birds-Eye View

Objects
`visionhdl.FrameToPixels`

Functions
`estimateGeometricTransform` | `imwarp`

Topics

“Single Camera Calibrator App” (Computer Vision Toolbox)

Introduced in R2017b

visionhdl.ChromaResampler

Package: `visionhdl`

Downsample or upsample chrominance component

Description

`visionhdl.ChromaResampler` 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.

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. The object accepts and returns 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”.

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.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`CR = visionhdl.ChromaResampler` returns a System object, `CR`, that downsamples from 4:4:4 to 4:2:2 and applies the default antialiasing filter.

`CR = visionhdl.ChromaResampler(Name,Value)` returns a chroma resampler System object, `CR`, with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-21 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Resampling`

Resampling format.

- `4:4:4 to 4:2:2` (default) — Perform a downsampling operation.
- `4:2:2 to 4:4:4` — Perform an upsampling operation.

`AntialiasingFilterSource`

Lowpass filter to accompany a downsample operation.

- `Auto` (default) — Built-in lowpass filter.
- `Property` — Filter using the coefficients in `HorizontalFilterCoefficients` property.
- `None` — No filtering of the input signal.

This property applies when you set `Resampling` to `4:4:4 to 4:2:2`.

`HorizontalFilterCoefficients`

Coefficients for the antialiasing filter.

Enter the coefficients as a vector. This property applies when you set `Resampling` to `4:4:4 to 4:2:2` and `Antialiasing filter` to `Property`.

Default: `[0.2,0.6,0.2]`

`InterpolationFilter`

Interpolation method for an upsample operation.

- `Linear` (default) — Linear interpolation to calculate the missing values.
- `Pixel replication` — Repeat the chrominance value of the preceding pixel to create the missing pixel.

This property applies when you set `Resampling` to `4:2:2 to 4:4:4`.

`RoundingMethod`

Rounding mode used for fixed-point operations.

The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input data type is `single` or `double`.

Default: `Floor`

`OverflowAction`

Overflow action used for fixed-point operations.

The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input data type is `single` or `double`.

Default: `Wrap`

`CustomCoefficientsDataType`

Data type for the antialiasing filter coefficients.

Specify a custom data type as a character vector. This parameter applies when you set `Antialiasing filter` to `Property` or `Auto`.

Default: `'fixdt(1,16,0)'`

Methods

step	Compute next pixel in upsampled or downsampled pixel stream

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Downsample a Y'CbCr Image

Resample a 4:4:4 Y'CbCr image to 4:2:2. The example also shows how to convert a 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 anti-aliasing filter is a 29-tap lowpass filter. This 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.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[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 YCbCr, 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);

pix2frm422 = visionhdl.PixelsToFrame(...
     'NumComponents',3,...
     'VideoFormat','custom',...
     'ActivePixelsPerLine',frmActivePixels,...
     'ActiveVideoLines',frmActiveLines);

[frm444,frmValid] = pix2frm444(pix444,ctrl444);
[frm422,frmValid] = pix2frm422(pix422,ctrl422);
```

There are the same number of pixels in the 4:2:2 and 4:4:4 pixel-streams and frames. 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. Notice that, in the 4:2:2 data, the Cb and Cr elements change only every second sample.

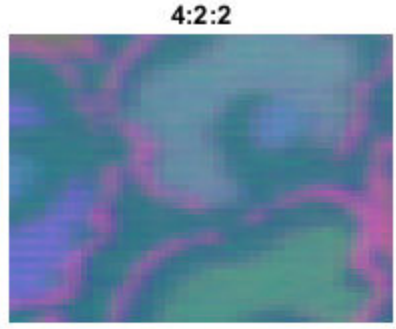
```
YCbCr444 = [frm444(1,1:8,1); frm444(1,1:8,2); frm444(1,1:8,3)]
```

```
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
```

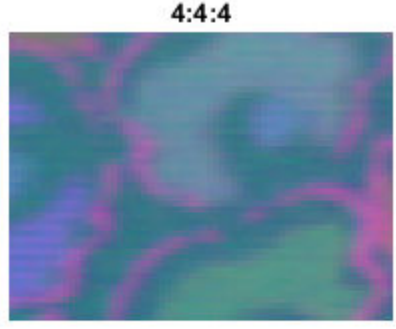
```
YCbCr422 = [frm422(1,1:8,1); frm422(1,1:8,2); frm422(1,1:8,3)]
```

```
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.

See Also

Chroma Resampler | `vision.ChromaResampler` | `visionhdl.FrameToPixels`

Introduced in R2015a

step

System object: `visionhdl.ChromaResampler`
Package: `visionhdl`

Compute next pixel in upsampled or downsampled pixel stream

Syntax

```
[pixelOut,ctrlOut] = step(resample,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(resample,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 luma component and control signals, `ctrlIn`, are passed through and 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 scalar pixel value. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as

dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`resample` — Resampler

`visionhdl.ChromaResampler` System object

Specify a `visionhdl.ChromaResampler` System object that you created and configured.

`pixelIn` — Input pixel

vector

Single pixel in gamma-corrected Y'CbCr color space, specified as a vector of three values.

Supported data types:

- `uint8` or `uint16`
- `fixdt(0,N,0)`, N = 8,9,...,16
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

vector

Single pixel in gamma-corrected Y'CbCr color space, returned as a vector of three values.

Supported data types:

- `uint8` or `uint16`
- `fixdt(0,N,0)`, N = 8,9,...,16
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlOut` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.ColorSpaceConverter

Package: `visionhdl`

Convert color information between color spaces

Description

`visionhdl.ColorSpaceConverter` converts between R'G'B' and Y'CbCr color spaces, and also converts R'G'B' to intensity.

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. The object accepts and returns 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".

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.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`CSC = visionhdl.ColorSpaceConverter` returns a System object, `CSC`, that converts R'G'B' to Y'CbCr using the Rec. 601 (SDTV) standard.

`CSC = visionhdl.ColorSpaceConverter(Name,Value)` returns a System object, `CSC`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name`

is a property name on page 2-32 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Conversion`

Conversion that the object performs on the input video stream.

- `RGB to YCbCr` (default)
- `YCbCr to RGB`
- `RGB to intensity`

The `step` method 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 to use on the input video stream.

- `Rec. 601 (SDTV)` (default)
- `Rec. 709 (HDTV)`

This property does not apply when you set `Conversion` to `RGB to intensity`.

`ScanningStandard`

Scanning standard to use for HDTV conversion.

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

This property applies when you set `ConversionStandard` to `Rec. 709 (HDTV)`.

Methods

step Convert one pixel between color spaces

Common to All System Objects	
`release`	Allow System object property value changes

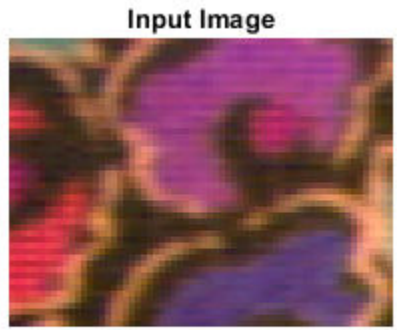
Examples

Convert a 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'
```



Create a serializer object and specify size of 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 RGB 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.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

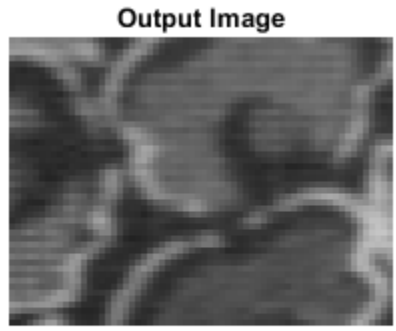
Set up variables, and convert each pixel in the stream to the new color space.

```
[~,~,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 format matching that 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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
```

```
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Color Space Converter block reference page.

See Also

Colorspace Converter | `rgb2gray` | `rgb2ycbcr` | `visionhdl.FrameToPixels` | `ycbcr2rgb`

Introduced in R2015a

step

System object: `visionhdl.ColorSpaceConverter`
Package: `visionhdl`

Convert one pixel between color spaces

Syntax

```
[pixelOut,ctrlOut] = step(convert_color_space,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(convert_color_space,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 `pixelOut` is a vector of three values representing one pixel. If the `Conversion` property is set to `RGB to intensity`, then `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 scalar pixel value. The object accepts and returns 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”.

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.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`convert_color_space` — Color space converter

`visionhdl.ColorSpaceConverter` System object

Specify a `visionhdl.ColorSpaceConverter` System object that you created and configured.

`pixelIn` — Input pixel

vector

Input pixel in gamma-corrected R'G'B' or Y'CbCr color space, specified as a vector of unsigned integer values.

Supported data types:

- `uint8` or `uint16`
- `fixdt(0,N,0)`, N = 8,9,...,16
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

vector

Output pixel specified as a vector of three unsigned integer values, or a scalar unsigned integer value.

- 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.

Supported data types:

- `uint8` or `uint16`
- `fixdt(0,N,0)`, N = 8,9,....,16
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlOut` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.Closing

Package: `visionhdl`

Morphological closing of binary pixel data

Description

`visionhdl.Closing` performs morphological dilation, followed by morphological erosion, using the same neighborhood for both calculations. The object operates on a stream of binary intensity values.

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. The object accepts and returns 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”.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`C = visionhdl.Closing` returns a System object, `C`, that performs morphological closing on a binary pixel stream.

`C = visionhdl.Closing(Name,Value)` returns a System object, `C`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Neighborhood`

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The object supports neighborhoods of up to 32×32 pixels. To use a structuring element, specify `Neighborhood` as `getnhood(strel(`*`shape`*`))`.

Default: `[0,1,0;1,1,1;0,1,0]`

`LineBufferSize`

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 single horizontal line.

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 object uses the next largest power of two. The object allocates (*neighborhood lines* – 1)-by-`LineBufferSize` memory locations to store the pixels.

Default: `2048`

Methods

step	Report closed pixel value based on neighborhood

Common to All System Objects	
`release`	Allow System object property value changes

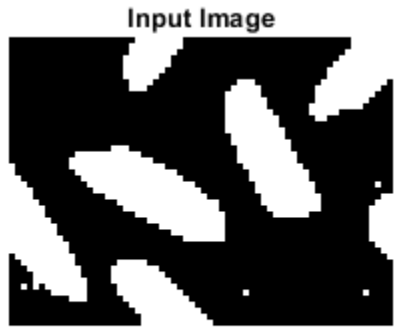
Examples

Morphological Close

Perform morphological close on a thumbnail image.

Load a source image from a file. Select a portion of the image that matches the desired test size. This source image contains `uint8` pixel intensity values. Apply a threshold to convert to binary pixel data.

```
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 object and define inactive pixel regions. Make the number of inactive pixels following each active line at least double the horizontal size of the neighborhood. Make the number of lines following each frame 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 filter object.

```
 mclose = 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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels 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

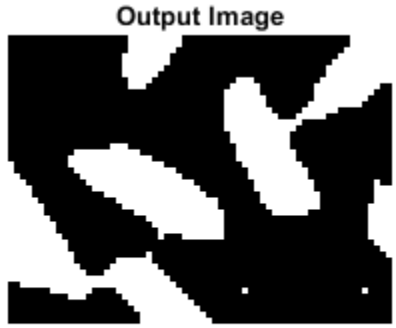
```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = mclose(pixIn(p),ctrlIn(p));
    if (ctrlOut(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
sprintf('object latency is %d cycles',foundValOut-foundValIn)
```

```
ans =
'object latency is 540 cycles'
```

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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Closing block reference page.

See Also

Closing | `imclose` | `visionhdl.Dilation` | `visionhdl.Erosion` | `visionhdl.FrameToPixels` | `visionhdl.Opening`

Introduced in R2015a

step

System object: `visionhdl.Closing`
Package: `visionhdl`

Report closed pixel value based on neighborhood

Syntax

```
[pixelOut,ctrlOut] = step(close,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(close,pixelIn,ctrlIn)` returns the next binary pixel value, `pixelOut`, resulting from a morphological close operation on 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable

property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`close` — Morphological closer

`visionhdl.Closing` System object

Specify a `visionhdl.Closing` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar `logical` value.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

scalar

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

`ctrlOut` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.GrayscaleClosing

Package: `visionhdl`

Morphological closing of grayscale pixel data

Description

`visionhdl.GrayscaleClosing` performs a morphological dilation operation, followed by a morphological erosion operation, 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 32×32 pixels. For line, square, or rectangle structuring elements more than 8 pixels wide, the object uses the Van Herk algorithm to find the maximum and minimum. For structuring elements less than 8 pixels wide, or that contain zero elements, the object implements a pipelined comparison tree to find the maximum and minimum.

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. The object accepts and returns 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”.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`C = visionhdl.GrayscaleClosing` returns a System object, `C`, that performs morphological closing on a pixel stream.

`C = visionhdl.GrayscaleClosing(Name,Value)` returns a System object, `C`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name on page 2-48 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Neighborhood`

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The object supports neighborhoods of up to 32×32 pixels. To use a structuring element, specify `Neighborhood` as `getnhood(strel(`*`shape`*`))`. The minimum neighborhood size is a 2×2 matrix, or a 2×1 column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

Default: `ones(3,3)`

`LineBufferSize`

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 single horizontal line.

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 object uses the next largest power of two. The object allocates (*neighborhood lines* – 1)-by-`LineBufferSize` memory locations to store the pixels.

Default: `2048`

Methods

step	Report closed pixel value based on neighborhood

Common to All System Objects	
`release`	Allow System object property value changes

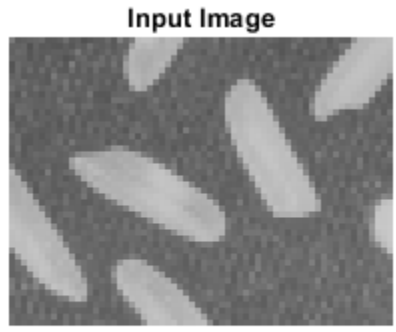
Examples

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'
```



Create a serializer object and define the inactive pixel regions. Make the number of inactive pixels following each active line at least double the horizontal size of the neighborhood. Make the number of lines following each frame 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.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[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.

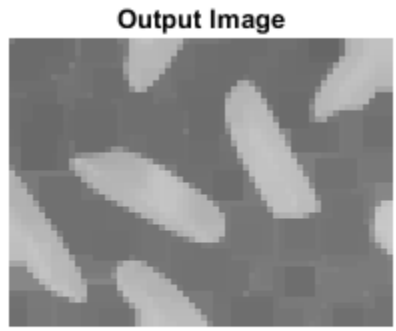
```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = mclose(pixIn(p),ctrlIn(p));
    if (ctrlOut(p).valid && foundValOut==0)
        foundValOut = p;
    end
```

```
end
sprintf('object latency is %d cycles',foundValOut-foundValIn)
```

```
ans =
'object latency is 384 cycles'
```

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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Grayscale Closing block reference page.

See Also

Grayscale Closing | `imclose` | `visionhdl.FrameToPixels` | `visionhdl.GrayscaleDilation` | `visionhdl.GrayscaleErosion` | `visionhdl.GrayscaleOpening`

Introduced in R2016a

step

System object: `visionhdl.GrayscaleClosing`
Package: `visionhdl`

Report closed pixel value based on neighborhood

Syntax

```
[pixelOut,ctrlOut] = step(close,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(close,pixelIn,ctrlIn)` returns the next pixel value, `pixelOut`, resulting from morphological closing on 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable

property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`close` — Morphological closer

`visionhdl.GrayClosing` System object

Specify a `visionhdl.GrayClosing` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar value.

Supported data types:

- `uint8`, `uint16`,`uint32`
- `fixdt(0,N,M)`
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

scalar

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

The data type is the same as the data type of `pixelIn`.

`ctrlOut` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2016a

visionhdl.DemosaicInterpolator

Package: `visionhdl`

Construct full RGB pixel data from Bayer pattern pixels

Description

`visionhdl.DemosaicInterpolator` provides a Bayer pattern interpolation filter for streaming video data. 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 3×3 pixel window using only additions and bit shifts.
- The object performs gradient correction on a 5×5 pixel window. The object implements the calculation using bit shift, addition, and low order Canonical Signed Digit (CSD) multiply.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`D = visionhdl.DemosaicInterpolator` returns a System object, `D`, that interpolates R'G'B' data from a Bayer pattern pixel stream.

`D = visionhdl.DemosaicInterpolator(Name,Value)` returns a System object, `D`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`InterpolationAlgorithm`

Algorithm the object uses to calculate the missing pixel values.

- `Bilinear` — Average of the pixel values in the surrounding 3×3 neighborhood.
- `Gradient-corrected linear` (default) — Bilinear average, corrected for intensity gradient.

`SensorAlignment`

Color sequence of the pixels in the input stream.

Specify 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 value, `RGGB`, represents an image with this pattern.

R	G
G	B

`LineBufferSize`

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 single horizontal line.

Choose a power of 2 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. When you set `InterpolationAlgorithm` to `Bilinear`, the object allocates 2-by-`LineBufferSize` memory locations. When you set `InterpolationAlgorithm` to `Gradient-corrected linear`, the object allocates 4-by-`LineBufferSize` memory locations.

Default: 2048

Methods

step	Demosaic a Bayer pattern video stream

Common to All System Objects	
`release`	Allow System object property value changes

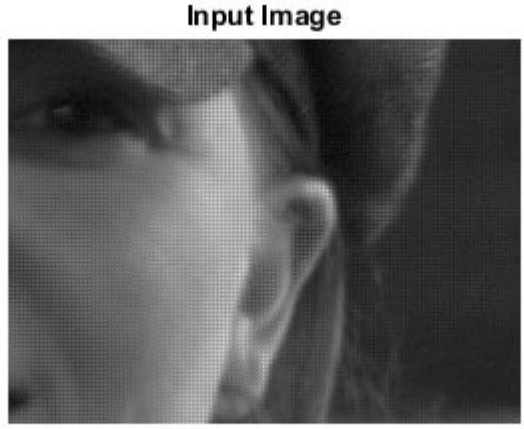
Examples

Demosaic a Bayer Pattern Image

This example constructs 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 in Bayer pattern: each pixel is represented by one value, alternating green values with red and blue values. Then select a portion of the image matching the desired test size. These offsets select the face of the woman 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'
```



Create a serializer object and specify 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.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

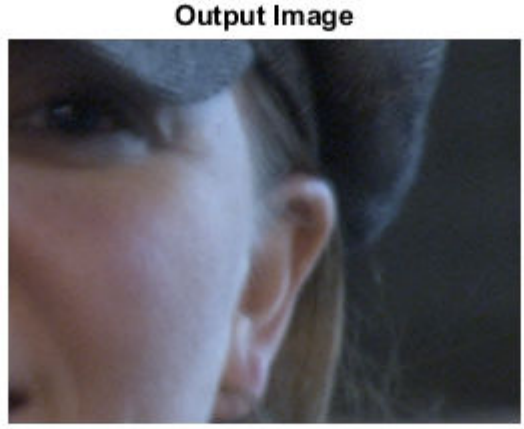
Set up variables, and generate the {R,G,B} triplet for each pixel in the stream. This example prints a progress message every 32 lines.

```
[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 matching that of the serializer. Convert the pixel stream to an image frame, and display the result.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',3,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Demosaic Interpolator block reference page.

See Also

Demosaic Interpolator | `demosaic` | `visionhdl.FrameToPixels`

Introduced in R2015a

step

System object: `visionhdl.DemosaicInterpolator`
Package: `visionhdl`

Demosaic a Bayer pattern video stream

Syntax

```
[pixelOut,ctrlOut] = step(demosaic,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(demosaic,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 R'G'B' 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as

dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`demosaic` — Interpolator

`visionhdl.DemosaicInterpolator` System object

Specify a `visionhdl.DemosaicInterpolator` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar value.

Supported data types:

- `uint` or `int`
- `fixdt(0,N,0)`
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

scalar

Single pixel, returned as a vector of three values in R'G'B' color space.

The data type of `pixelOut` is the same as the data type of `pixelIn`.

`ctrlOut` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.Dilation

Package: `visionhdl`

Morphological dilation of binary pixel data

Description

`visionhdl.Dilation` replaces each pixel with the local maximum of the neighborhood around the pixel. The object operates on a stream of binary intensity values.

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. The object accepts and returns 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".

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`D = visionhdl.Dilation` returns a System object, `D`, that performs morphological dilation on a binary video stream.

`D = visionhdl.Dilation(Name,Value)` returns a System object, `D`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name on page 2-66 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Neighborhood`

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The object supports neighborhoods of up to 32×32 pixels. To use a structuring element, specify `Neighborhood` as `getnhood(strel(`*`shape`*`))`.

Default: `[0,1,0;1,1,1;0,1,0]`

`LineBufferSize`

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 single horizontal line.

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 object uses the next largest power of two. The object allocates (*neighborhood lines* – 1)-by-`LineBufferSize` memory locations to store the pixels.

Default: `2048`

Methods

step	Report dilated pixel value based on neighborhood

Common to All System Objects	
`release`	Allow System object property value changes

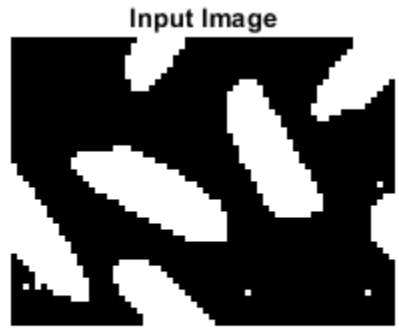
Examples

Morphological Dilate

Perform morphological dilate on a thumbnail image.

Load a source image from a file. Select a portion of the image that matches the desired test size. This source image contains `uint8` pixel intensity values. Apply a threshold to convert to binary pixel data.

```
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 object and define inactive pixel regions. Make the number of inactive pixels following each active line at least double the horizontal size of the neighborhood. Make the number of lines following each frame 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 filter object.

```
 mdilate = 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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels 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

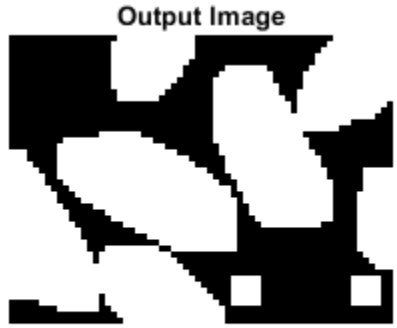
```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = mdilate(pixIn(p),ctrlIn(p));
    if (ctrlOut(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
sprintf('object latency is %d cycles',foundValOut-foundValIn)
```

```
ans =
'object latency is 184 cycles'
```

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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Dilation block reference page.

See Also

Dilation | `imdilate` | `visionhdl.Erosion` | `visionhdl.FrameToPixels`

Introduced in R2015a

step

System object: `visionhdl.Dilation`
Package: `visionhdl`

Report dilated pixel value based on neighborhood

Syntax

```
[pixelOut,ctrlOut] = step(dilate,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(dilate,pixelIn,ctrlIn)` returns the next pixel value, `pixelOut`, resulting from a morphological dilation operation on the neighborhood around each input 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable

property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`dilate` — Morphological dilator

`visionhdl.Dilation` System object

Specify a `visionhdl.Dilation` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar `logical` value.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

scalar

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

`ctrlOut` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.GrayscaleDilation

Package: `visionhdl`

Morphological dilation of grayscale pixel data

Description

`visionhdl.GrayscaleDilation` performs morphological dilation on a stream of pixel intensity values. You can specify a neighborhood, or structuring element, of up to 32×32 pixels. For line, square, or rectangle structuring elements 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 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.

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. The object accepts and returns 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”.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`D = visionhdl.GrayscaleDilation` returns a System object, `D`, that performs morphological dilation on a pixel stream.

`D = visionhdl.GrayscaleDilation(Name,Value)` returns a System object, `D`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Neighborhood`

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The object supports neighborhoods of up to 32×32 pixels. To use a structuring element, specify `Neighborhood` as `getnhood(strel(`*`shape`*`))`. The minimum neighborhood size is a 2×2 matrix, or a 2×1 column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

Default: `ones(5,5)`

`LineBufferSize`

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 single horizontal line.

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 object uses the next largest power of two. The object allocates (*neighborhood lines* – 1)-by-`LineBufferSize` memory locations to store the pixels.

Default: `2048`

Methods

step	Report dilated pixel value based on neighborhood

Common to All System Objects	
`release`	Allow System object property value changes

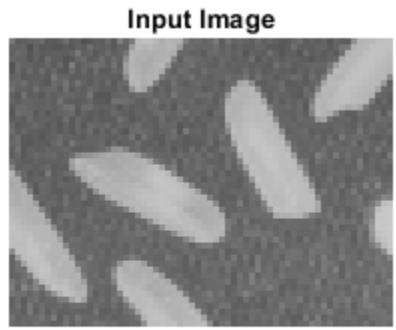
Examples

Grayscale Morphological Dilation

Perform morphological dilation 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'
```



Create a serializer object and define the inactive pixel regions. Make the number of inactive pixels following each active line at least double the horizontal size of the neighborhood. Make the number of lines following each frame 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.

```
mdilate = 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.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[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);
```

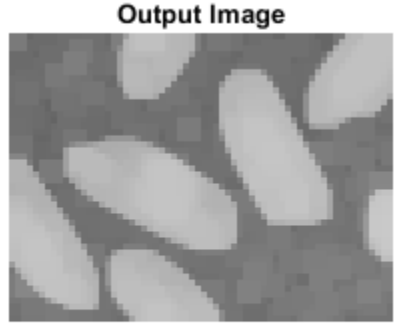
For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
```

```
    end
    [pixOut(p),ctrlOut(p)] = mdilate(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 104 cycles'
```

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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Grayscale Dilation block reference page.

See Also

Grayscale Dilation | `imdilate` | `visionhdl.FrameToPixels` | `visionhdl.GrayscaleErosion`

Introduced in R2016a

step

System object: `visionhdl.GrayscaleDilation`
Package: `visionhdl`

Report dilated pixel value based on neighborhood

Syntax

```
[pixelOut,ctrlOut] = step(dilate,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(dilate,pixelIn,ctrlIn)` returns the next pixel value, `pixelOut`, resulting from morphological dilation on 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable

property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`dilate` — Morphological dilator

`visionhdl.GrayscaleDilation` System object

Specify a `visionhdl.GrayscaleDilation` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar value.

Supported data types:

- `uint8`, `uint16`,`uint32`
- `fixdt(0,N,M)`
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

scalar

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

The data type is the same as the data type of `pixelIn`.

`ctrlOut` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2016a

visionhdl.EdgeDetector

Package: `visionhdl`

Find edges of objects

Description

`visionhdl.EdgeDetector` 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.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ED = visionhdl.EdgeDetector` returns a System object, `ED`, that detects edges using the Sobel method.

`ED = visionhdl.EdgeDetector(Name,Value)` returns a System object, `ED`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Method`

Edge detection algorithm.

Specify `'Sobel'`, `'Prewitt'`, or `'Roberts'` method.

Default: `'Sobel'`

`BinaryImageOutputPort`

Enable the `Edge` output of the `step` method.

When this property is `true`, the `step` method returns a binary pixel value representing whether the object detected an edge at each location in the frame.

Default: `true`

`GradientComponentOutputPorts`

Enable the `G1` and `G2` outputs of the `step` method.

When this property is `true`, the `step` method returns two values representing the gradients calculated in two orthogonal directions at each pixel. Set the data type for this argument in the `GradientDataType` property.

Default: `false`

`ThresholdSource`

Source for the gradient threshold value that indicates an edge.

Set this property to `'Input port'`to set the threshold as an input argument to the `step` method. When this property is set to `'Property'`, set the threshold in the `Threshold` property.

Default: `'Property'`

`Threshold`

Gradient threshold value that indicates an edge, specified as a numeric scalar value.

The object compares the square of this 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'`.

Default: 20

`LineBufferSize`

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 single horizontal line.

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 object 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.

Default: 2048

`RoundingMethod`

Rounding mode used for fixed-point operations.

The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input data type is `single` or `double`.

Default: `Floor`

`OverflowAction`

Overflow action used for fixed-point operations.

The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input data type is `single` or `double`.

Default: `Wrap`

`GradientDataType`

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.

- `'Full precision'` (default) — Use full-precision rules based on the data type of the `pixelIn` argument of the `step` method, and the coefficients of the derivative approximation matrices.
- `'custom'` — Use the data type defined in the`CustomGradientDataType` property.

`CustomGradientDataType`

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.

Default: `numerictype(1,8,0)`

Methods

step	Detect edges at an image pixel

Common to All System Objects	
`release`	Allow System object property value changes

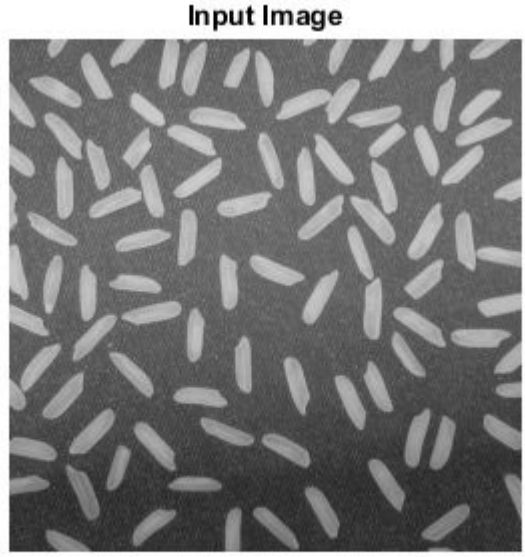
Examples

Edge Detection Using Sobel Method

Detect edges in a thumbnail image using the Sobel method.

Import a test image.

```
frmInput = imread('rice.png');
[frmActivePixels,frmActiveLines] = size(frmInput);
figure
imshow(frmInput)
title '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 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.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

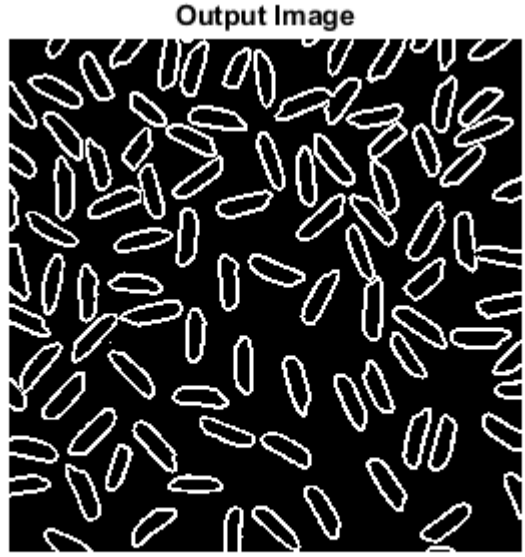
```
[pixIn,ctrlIn] = frm2pix(frmInput);

[~,~,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);

[frmOutput,frmValid] = pix2frm(edgeOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Edge Detector block reference page.

See Also

Edge Detector | `edge` | `visionhdl.FrameToPixels`

Topics

“Enhanced Edge Detection from Noisy Color Video”

Introduced in R2015a

step

System object: `visionhdl.EdgeDetector`
Package: `visionhdl`

Detect edges at an image pixel

Syntax

```
[edge,ctrlOut] = step(detect_edges,pixelIn,ctrlIn)
[G1,G2,ctrlOut] = step(detect_edges,pixelIn,ctrlIn)
[edge,ctrlOut] = step(detect_edges,pixelIn,ctrlIn,thresh)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[edge,ctrlOut] = step(detect_edges,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] = step(detect_edges,pixelIn,ctrlIn)` detects edges in the neighborhood of `pixelIn` by computing the gradient in two orthogonal directions. Use this syntax when you set `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] = step(detect_edges,pixelIn,ctrlIn,thresh)` detects edges in the neighborhood of `pixelIn` by computing the gradient in two orthogonal directions.

Use this syntax when you set `ThresholdSource` property to `'InputPort'`. The `edge` output argument is a binary value indicating whether the sum of the squares of the gradients was above the threshold, `thresh`, squared.

You can use any combination of the optional port 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`detect_edges` — Edge detector

`visionhdl.EdgeDetector` System object

Specify a `visionhdl.EdgeDetector` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Intensity of a single pixel, specified as a scalar value.

Supported data types:

- `uint` or `int`
- `fixdt()`
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel 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. This argument is accepted when you set `ThresholdSource` property to `'InputPort'`.

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.

This argument is returned when you set `GradientComponentOutputPorts` property to `true`. 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.

`G2` — Gradient in second direction
scalar

Gradient calculated in the second direction, returned as a scalar value.

This argument is returned when you set `GradientComponentOutputPorts` property to `true`. 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.

`ctrlOut` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.Erosion

Package: `visionhdl`

Morphological erosion of binary pixel data

Description

`visionhdl.Erosion` replaces each pixel with the local minimum of the neighborhood around the pixel. The object operates on a stream of binary intensity values.

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. The object accepts and returns 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".

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`E = visionhdl.Erosion` returns a System object, `E`, that performs morphological erosion on a binary pixel stream.

`E = visionhdl.Erosion(Name,Value)` returns a System object, `E`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name on page 2-94 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Neighborhood`

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The object supports neighborhoods of up to 32×32 pixels. To use a structuring element, specify `Neighborhood` as `getnhood(strel(`*`shape`*`))`.

Default: `ones(3,3)`

`LineBufferSize`

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 single horizontal line.

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 object uses the next largest power of two. The object allocates (*neighborhood lines* – 1)-by-`LineBufferSize` memory locations to store the pixels.

Default: `2048`

Methods

step	Report eroded pixel value based on neighborhood

Common to All System Objects	
`release`	Allow System object property value changes

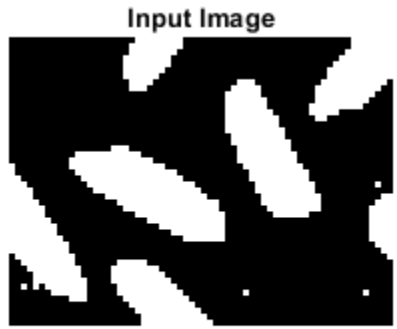
Examples

Morphological Erode

Perform morphological erode on a thumbnail image.

Load a source image from a file. Select a portion of the image that matches the desired test size. This source image contains `uint8` pixel intensity values. Apply a threshold to convert to binary pixel data.

```
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 object and define inactive pixel regions. Make the number of inactive pixels following each active line at least double the horizontal size of the neighborhood. Make the number of lines following each frame 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 filter object.

```
 merode = 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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = false(numPixelsPerFrame,1);
ctrlOut  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

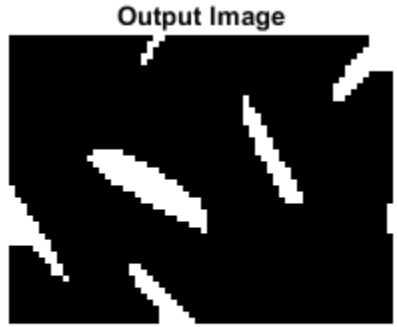
For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = merode(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 102 cycles'
```

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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Erosion block reference page.

See Also

Erosion | `imerode` | `visionhdl.Dilation` | `visionhdl.FrameToPixels`

Introduced in R2015a

step

System object: `visionhdl.Erosion`
Package: `visionhdl`

Report eroded pixel value based on neighborhood

Syntax

```
[pixelOut,ctrlOut] = step(erode,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(erode,pixelIn,ctrlIn)` returns the next pixel value, `pixelOut`, in the pixel stream resulting from a morphological erosion operation on the neighborhood around each input 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable

property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`erode` — Morphological eroder

`visionhdl.Erosion` System object

Specify a `visionhdl.Erosion` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar `logical` value.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

scalar

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

`ctrlOut` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.GrayscaleErosion

Package: `visionhdl`

Morphological erosion of grayscale pixel data

Description

`visionhdl.GrayscaleErosion` performs morphological erosion on a stream of pixel intensity values. You can specify a neighborhood, or structuring element, of up to 32×32 pixels. For line, square, or rectangle structuring elements 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 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.

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. The object accepts and returns 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”.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`E = visionhdl.GrayscaleErosion` returns a System object, `E`, that performs a morphological erosion on a pixel stream.

`E = visionhdl.GrayscaleErosion(Name,Value)` returns a System object, `E`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name on page 2-102 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Neighborhood`

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The object supports neighborhoods of up to 32×32 pixels. To use a structuring element, specify `Neighborhood` as `getnhood(strel(`*`shape`*`))`. The minimum neighborhood size is a 2×2 matrix, or a 2×1 column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

Default: `ones(3,3)`

`LineBufferSize`

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 single horizontal line.

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 object uses the next largest power of two. The object allocates (*neighborhood lines* – 1)-by-`LineBufferSize` memory locations to store the pixels.

Default: `2048`

Methods

step Report eroded pixel value based on neighborhood

Common to All System Objects	
`release`	Allow System object property value changes

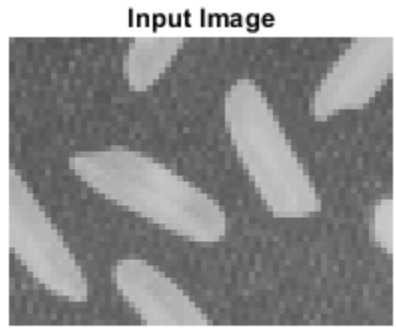
Examples

Grayscale Morphological Erosion

Perform morphological erosion 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'
```



Create a serializer object and define the inactive pixel regions. Make the number of inactive pixels following each active line at least double the horizontal size of the neighborhood. Make the number of lines following each frame 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.

```
merode = 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.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[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);
```

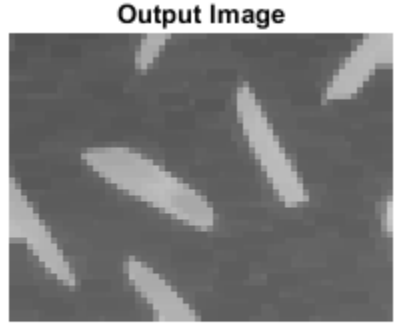
For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
```

```
    end
    [pixOut(p),ctrlOut(p)] = merode(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 107 cycles'
```

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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Grayscale Erosion block reference page.

See Also

Grayscale Erosion | `imerode` | `visionhdl.FrameToPixels` | `visionhdl.GrayscaleDilation`

Introduced in R2016a

step

System object: `visionhdl.GrayscaleErosion`
Package: `visionhdl`

Report eroded pixel value based on neighborhood

Syntax

```
[pixelOut,ctrlOut] = step(erode,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(erode,pixelIn,ctrlIn)` returns the next pixel value, `pixelOut`, resulting from a morphological erosion of 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable

property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`erode` — Morphological eroder

`visionhdl.GrayscaleErosion` System object

Specify a `visionhdl.GrayscaleErosion` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar value.

Supported data types:

- `uint8`, `uint16`,`uint32`
- `fixdt(0,N,M)`
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

scalar

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

The data type is the same as the data type of `pixelIn`.

`ctrlOut` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2016a

visionhdl.FrameToPixels

Package: `visionhdl`

Convert frame-based video to pixel stream

Description

`visionhdl.FrameToPixels` 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 specifying custom dimensions. See “Streaming Pixel Interface” for details of the pixel stream format.

Use this object to generate input for a function targeted for HDL code generation. This block 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.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`F2P = visionhdl.FrameToPixels` returns a System object, `F2P`, that serializes a grayscale 1080×1920 frame into a 1080p pixel stream with standard padding around the active data.

`F2P = visionhdl.FrameToPixels(Name,Value)` returns a System object, `F2P`, with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-111 and `Value` is the corresponding value. `Name` must appear

inside single quotes (''). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`NumComponents`

Components of each pixel, specified as 1, 3, or 4. Set to 1 for grayscale video. Set to 3 for color video, for example, {R,G,B} or {Y,Cb,Cr}. Set to 4 to use color with an alpha channel for transparency. The default is 1. The `visionhdl.FrameToPixels` object returns a *P*-by-`NumComponents` matrix, where *P* is the total number of pixels.

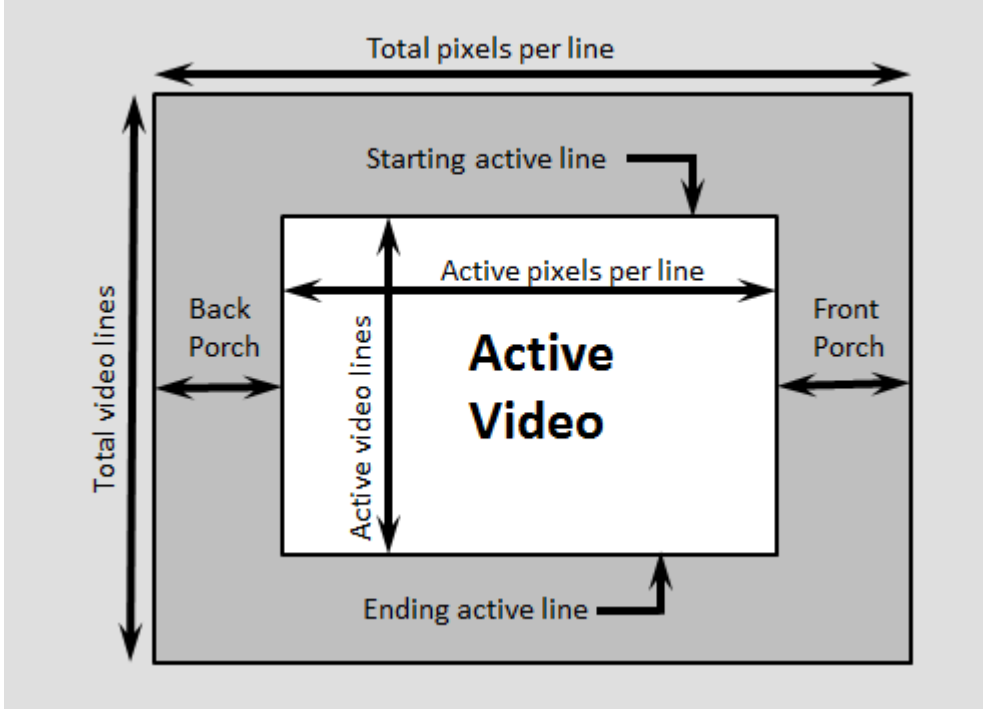
`NumPixels`

Number of pixels transferred on the streaming interface for each cycle, specified as `1`, `4`, or `8`. The default is `1`. To enable multipixel streaming and increase throughput for high-resolution or high-frame-rate video, set this property to `4` or `8`. The `visionhdl.FrameToPixels` object returns a *P*-by-`NumPixels` matrix, where *P* is the total number of pixels. When you set `NumPixels>1`, you must set the `NumComponents` property to `1`.

Note You can simulate System objects with a multipixel streaming interface, but they are not supported for HDL code generation. Use the equivalent blocks to generate HDL code for multipixel algorithms.

`VideoFormat`

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

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
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 block calculates *BackPorch* = `TotalPixelsPerLine` − `FrontPorch` − `ActivePixelsPerLine`.

For the vertical direction, `TotalVideoLines` must be greater than or equal to `StartingActiveLine` + `ActiveVideoLines` − 1. The block 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 < 4, the total porch must be at least 8 pixels.
- The `BackPorch` must be at least 6 pixels. This parameter is the number of inactive pixels before the first valid pixel in a frame.

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, 720p, 1080p, and 4k UHD support `NumPixels` equal to 4 or 8.
- The minimum input frame size for multipixel streaming (either 4 or 8 pixels) is 18 rows-by-32 columns.
- Choose your kernel size and `ActivePixelsPerLine` such that (`ActivePixelsPerLine`)/(`NumPixels`) is at least the kernel width.

Methods

step	Convert image frame to pixel stream

Common to All System Objects	
`release`	Allow System object property value changes

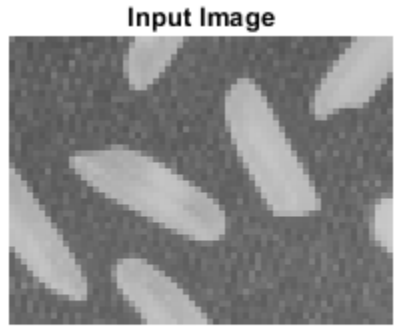
Examples

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 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 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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

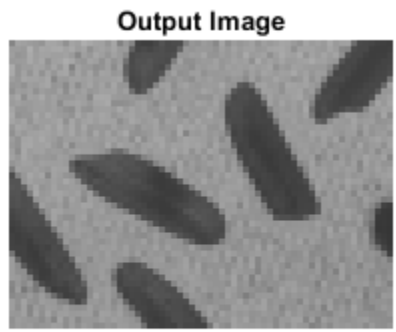
```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

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 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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```



See Also

Frame To Pixels | `visionhdl.PixelsToFrame`

Topics

“Pixel-Streaming Design in MATLAB”
“Streaming Pixel Interface”

Introduced in R2015a

step

System object: `visionhdl.FrameToPixels`
Package: `visionhdl`

Convert image frame to pixel stream

Syntax

```
[pixels,ctrlOut] = step(F2P,frm)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixels,ctrlOut] = step(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.

See “Streaming Pixel Interface” for details of the pixel stream format.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

F2P — Serializer
`visionhdl.FrameToPixels` System object

Specify a `visionhdl.PixelStreamAligner` System object that you created and configured.

`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 that specified by `VideoFormat`, the object returns a warning.

Supported data types:

- `uint` or `int`
- `fixdt()`
- `logical`
- `double` or `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, calculated as `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 they are not supported for HDL code generation. Use the equivalent blocks to generate HDL code for multipixel algorithms.

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, calculated as `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”.

Introduced in R2015a

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? (MATLAB).

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 (MATLAB).

`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. The object accepts and returns 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

scalar

Intensity of a single pixel, specified as a scalar value. For fixed-point data types, the input word length must be less than or equal to 16.

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

Data Types: `single` | `double` | `int8` | `int16` | `uint8` | `uint16` | `fi`

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

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`.

`ctrlOut` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying the 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

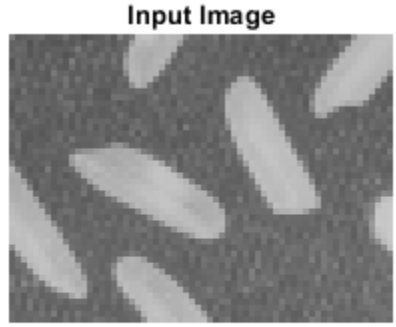
Examples

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'
```



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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

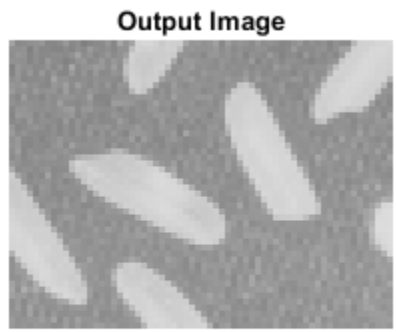
```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut  = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

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',...
      'ActivePixelsPerLine',frmActivePixels,...
      'ActiveVideoLines',frmActiveLines);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



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.

See Also

Blocks
Gamma Corrector

Objects
`vision.GammaCorrector` | `visionhdl.FrameToPixels`

Functions
`imadjust`

Topics
“Pixel-Streaming Design in MATLAB”

Introduced in R2015a

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? (MATLAB).

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 (MATLAB).

`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'`.

`double` and `single` data types are supported 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. This 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 1-122 section of the Histogram block page.

Input Arguments

`pixelIn` — Single image pixel

unsigned scalar

Single image pixel, specified as an unsigned scalar.

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

Data Types: `single` | `double` | `uint8` | `uint16` | `fixdt(0,N,0)`

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel 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,log2(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

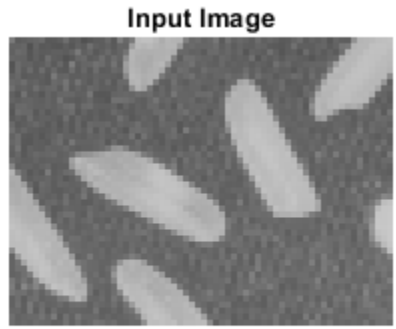
step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

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'
```



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
```

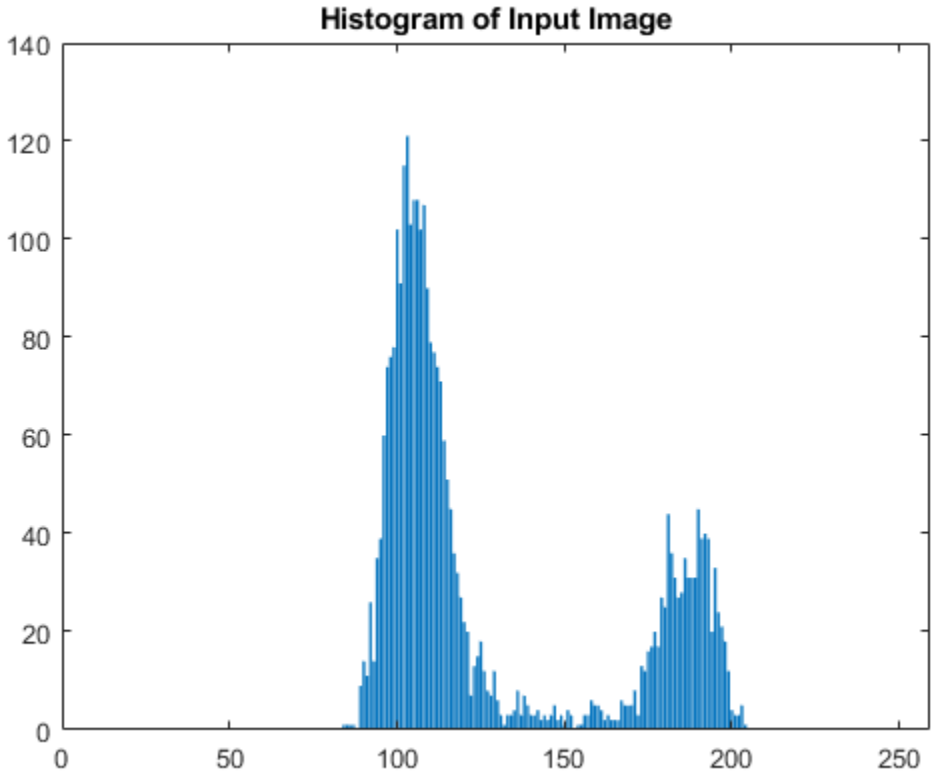
Once the frame is complete, as indicated by `readRdy`, read the bin values. The bin addresses are `0:bins-1`. The object returns each bin value after two cycles of latency, so call the object `bins+2` times.

```
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.

See Also

Histogram | `imhist` | `visionhdl.FrameToPixels`

Introduced in R2015a

visionhdl.HVCounter

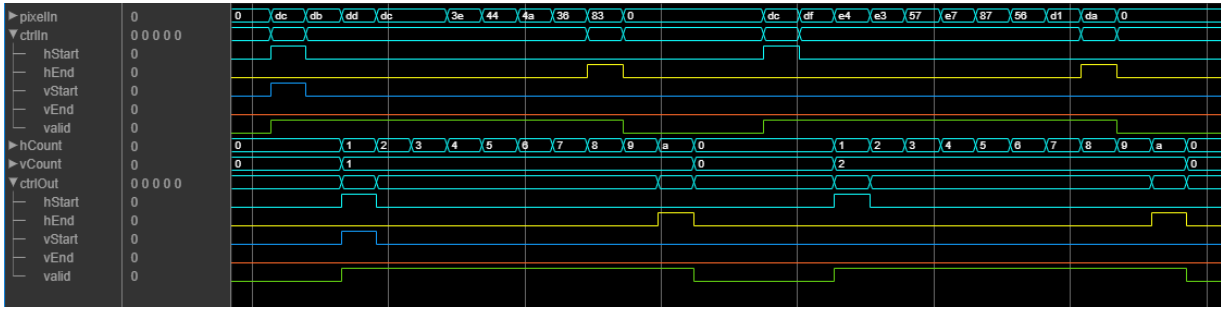
Package: `visionhdl`

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? (MATLAB).

Creation

Syntax

```
HVcnt = visionhdl.HVCounter
HVcnt = visionhdl.HVCounter(Name,Value)
```

Description

`HVcnt = 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).

`HVcnt = 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 (MATLAB).

`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(log2(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(log2(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] = HVcnt(ctrlIn)
```

Description

`[hCount,vCount,ctrlOut] = HVcnt(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 pixel stream
`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel 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 pixel stream

`pixelcontrol` structure

Control signals accompanying the 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

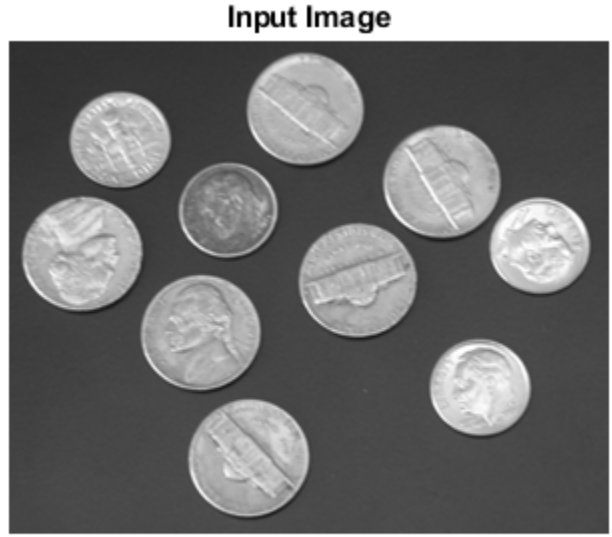
Examples

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'
```



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 accomodate 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
```

See Also

Blocks
HV Counter

Introduced in R2019a

visionhdl.ImageFilter

Package: `visionhdl`

2-D FIR filtering

Description

`visionhdl.ImageFilter` performs two-dimensional finite-impulse-response (FIR) filtering on a pixel stream. It supports the use of programmable filter coefficients.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`F = visionhdl.ImageFilter` returns a System object, `F`, that performs two-dimensional FIR filtering on an input pixel stream.

`F = visionhdl.ImageFilter(Name,Value)` returns a 2-D FIR filter System object, `F`, with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

`F = visionhdl.ImageFilter(coeff,lineSize,Name,Value)` returns a 2-D FIR filter System object, `F`, with the `Coefficients` property set to `coeff`, the `LineBufferSize` property to `lineSize`, and additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

`coeff`

Filter coefficients, specified as a matrix. Each dimension of the matrix must have at least 2 elements and no more than 64 elements. This argument sets the Coefficients property value.

`lineSize`

Size of the line memory buffer, specified as a power of two that accommodates the number of active pixels in a horizontal line. This argument sets the `LineBufferSize` property value.

Output Arguments

`F`

`visionhdl.ImageFilter` System object

Properties

`CoefficientsSource`

Select the source for specifying the filter coefficients.

- `'Property'` (default) — Select this value to specify filter coefficients using the `Coefficients` property.
- `'Input port'` — Select this value to specify filter coefficients using the `coeff` argument.

`Coefficients`

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. This property applies when you set `CoefficientsSource` to `'Property'`.

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

Default: `[1,0;0,-1]`

`PaddingMethod`

Method for padding the boundary of the input image. See “Edge Padding”.

- `'Constant'` (default) — Interpret pixels outside the image frame as having a constant value.
- `'Replicate'` — Repeats the value of pixels at the edge of the image.
- `'Symmetric'` — Pads the input matrix with its mirror image.

`PaddingValue`

Constant value used to pad the boundary of the input image. This property applies when you set `PaddingMethod` to `'Constant'`. The object casts this value to the same data type as the input pixel.

Default: 0

`LineBufferSize`

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 single horizontal line.

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 object uses the next largest power of two. The object allocates (*coefficient rows – 1*)-by-`LineBufferSize` memory locations to store the pixels.

Default: 2048

`CoefficientsDataType`

Select the method for determining the data type of the filter coefficients. This property applies when you set `CoefficientsSource` to `'Property'`.

- `'Same as first input''` (default) — Sets the data type of the coefficients to match the data type of the `pixelIn` argument of the `step` method.
- `'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.

`CustomCoefficientsDataType`

Data type for the filter coefficients, specified as `numerictype(signed,WL,FL)`, where *WL* is the word length and *FL* is the fraction length in bits. This property applies when you set `CoefficientsDataType` to `'custom'`.

Default: `numerictype(true,16,15)`

`OutputDataType`

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

- `'Same as first input'` (default) — Sets the data type of the output pixels to match the data type of the `pixelIn` argument of the `step` method.
- `'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.
- `'custom'` — Sets the data type of the output pixels to match the data type you define in the `CustomOutputDataType` property.

`CustomOutputDataType`

Data type for the output pixels, specified as `numerictype(signed,WL,FL)`, where *WL* is the word length and *FL* is the fraction length in bits. This property applies only when you set `OutputDataType` to `custom`.

Default: `numerictype(true,8,0)`

`OverflowAction`

Overflow action used for fixed-point operations.

The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input data type is `single` or `double`.

Default: `Wrap`

`RoundingMethod`

Rounding mode used for fixed-point operations.

The object uses fixed-point arithmetic for internal calculations when the input is any integer or fixed-point data type. This option does not apply when the input data type is `single` or `double`.

Default: `Floor`

Methods

step	2-D FIR filtering

Common to All System Objects	
`release`	Allow System object property value changes

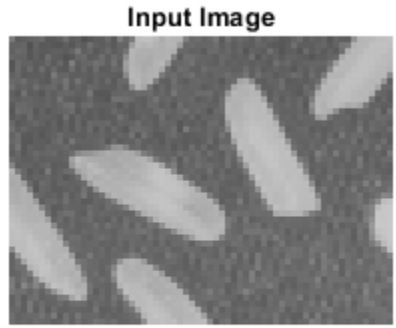
Examples

Filter a 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'
```



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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[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 filtered value. Monitor the 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 filter kernel.
- Optimization of symmetric or duplicate coefficients.

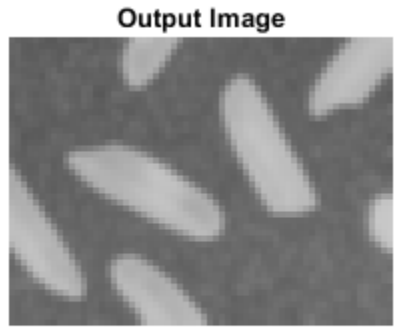
```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = filt2d(pixIn(p),ctrlIn(p));
    if (ctrlOut(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
sprintf('object latency is %d cycles',foundValOut-foundValIn)
```

```
ans =
'object latency is 101 cycles'
```

Create a deserializer object with a format 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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
```

```
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Image Filter block reference page.

See Also

Image Filter | `imfilter` | `visionhdl.FrameToPixels`

Introduced in R2015a

step

System object: `visionhdl.ImageFilter`
Package: `visionhdl`

2-D FIR filtering

Syntax

```
[pixelOut,ctrlOut] = step(filt,pixelIn,ctrlIn)
[pixelOut,ctrlOut] = step(filt,pixelIn,ctrlIn,coeff)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(filt,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] = step(filt,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'`.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable

property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`filt` — Filter

`visionhdl.ImageFilter` System object

Specify a `visionhdl.ImageFilter` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar value.

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

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

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel 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.

To enable this argument, set the `CoefficientsSource` property to `'Input port'`.

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

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.

`ctrlOut` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.ImageStatistics

Package: `visionhdl`

Mean, variance, and standard deviation

Description

`visionhdl.ImageStatistics` 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 object implements the calculations using hardware-efficient algorithms.

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 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".

- To change the size and dimensions of the ROI, you can manipulate the input video stream control signals. See "Regions of Interest" on page 1-141.
- The number of valid pixels in the input image affect the accuracy of the mean approximation. To avoid approximation error, 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 1-137.
- The object calculates statistics over frames up to 64^4 (16,777,216) pixels in size.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`S = visionhdl.ImageStatistics` returns a System object, `S`, that calculates the mean, variance, and standard deviation of each frame of a video stream.

`S = visionhdl.ImageStatistics(Name,Value)` returns a System object, `S`, with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-159 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`mean`

Calculate the mean of each input frame. If you set this property to `false`, the `step` method does not return this output.

Default: `true`

`variance`

Calculate the variance of each input frame. If you set this property to `false`, the `step` method does not return this output.

Default: `true`

`stdDev`

Calculate the standard deviation of each input frame. If you set this property to `false`, the `step` method does not return this output.

Default: `true`

Methods

step Calculate the contribution of one pixel to the mean, variance, and standard deviation of a video stream

Common to All System Objects	
`release`	Allow System object property value changes

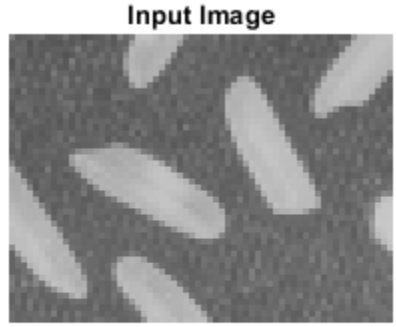
Examples

Compute Statistics of an 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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[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.

See Also

Image Statistics | `mean2` | `std2` | `vision.Mean` | `vision.StandardDeviation` | `vision.Variance` | `visionhdl.FrameToPixels`

Introduced in R2015a

step

System object: `visionhdl.ImageStatistics`
Package: `visionhdl`

Calculate the contribution of one pixel to the mean, variance, and standard deviation of a video stream

Syntax

```
[mean,variance,stdDeviation,validOut] =
step(statistics,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[mean,variance,stdDeviation,validOut] = step(statistics,pixelIn,ctrlIn)` incorporates the new pixel value, `pixelIn`, into 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. This 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`statistics` — Analyzer

`visionhdl.ImageStatistics` System object

Specify a `visionhdl.ImageStatistics` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar value.

Supported data types:

- `uint8` or `uint16`
- `fixdt(0,N,0)`, N = 8,9,...,16
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel 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`.

Introduced in R2015a

visionhdl.LineBuffer

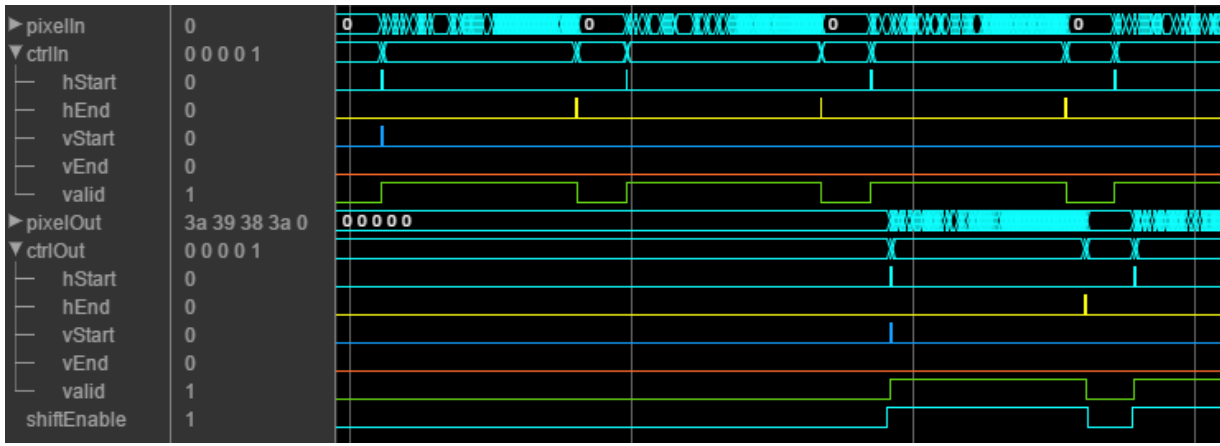
Package: `visionhdl`

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? (MATLAB).

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 (MATLAB).

`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'`

Method for padding the boundary of the input image.

- `'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'` — Pad the input matrix with its mirror image.

For more information on 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 property is valid 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 single horizontal line.

If you specify a value that is not a power of two, the object 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. The object accepts and returns 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` — Single image pixel

scalar

Single image pixel in a pixel stream, specified as a scalar value representing intensity.

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

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

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see "Pixel Control Structure".

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

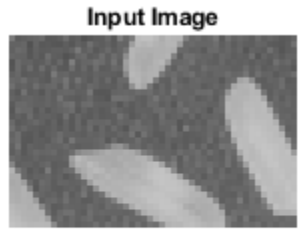
Construct an Averaging Filter Using a Line Buffer

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);
```



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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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
```

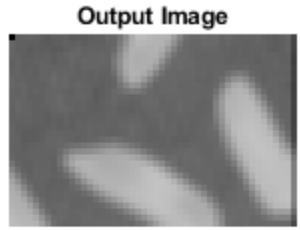
Process the image by calling the function for each pixel.

```
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
```

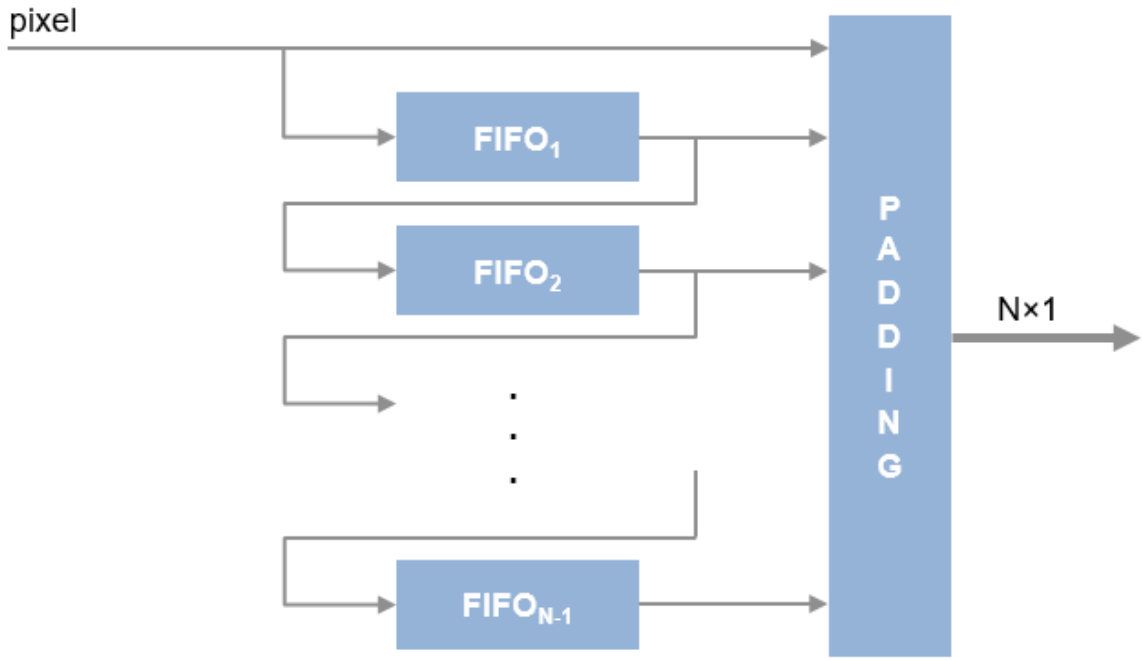
Recreate the filtered frame.

```
[frmOut,frmValid] = step(pix2frm,pixOutVec,ctrlOutVec);

if frmValid
    figure;
    imshow(frmOut,'InitialMagnification',300)
    title 'Output Image'
end
```



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)`.



See Also

Blocks
Line Buffer

Objects
`visionhdl.FrameToPixels`

Introduced in R2017b

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.

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. The object accepts and returns 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".

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`LUT = visionhdl.LookupTable` returns a System object, `LUT`, that performs a one-to-one mapping between the input pixel and output pixel, according to the lookup table contents.

`LUT = visionhdl.LookupTable(tabledata)` returns a lookup table System object, `LUT`, with the table contents set to `TABLEDATA`.

Input Arguments

`tabledata`

One-to-one correspondence between input pixels and output pixels, specified as a vector. This argument sets the Table property value.

Output Arguments

`LUT`

`visionhdl.LookupTable` System object

Properties

`Table`

Map between input pixel values and output pixel values.

- The table data is a vector, row or column, of any data type. The data type of the table data determines that of `pixelOut`. See `step` method.
- The length of the table data must equal $2^{WordLength}$, where *WordLength* is the size, in bits, of `pixelIn`. See `step` method.
- 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, input value 0.5 maps to the second element, 1 maps to the third, and so on.

Default: `uint8(0:1:255)`

Methods

step	Map input pixel to output pixel based on table contents

Common to All System Objects	
`release`	Allow System object property value changes

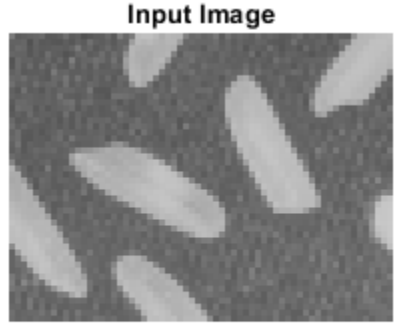
Examples

Compute Negative Image

This example creates the negative of an image by looking up the opposite pixel values in a table.

Set dimensions of 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'
```



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 is uint8 type, 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.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

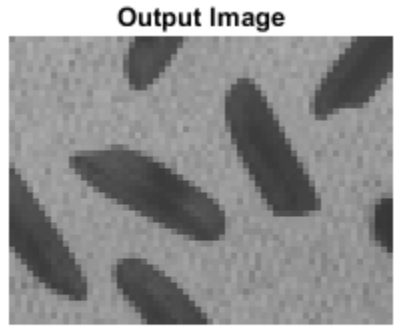
Prepare variables to process pixels. Then, for each pixel in the padded frame, look up the negative value.

```
[~,~,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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
```

```
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Lookup Table block reference page.

See Also

Lookup Table | `visionhdl.FrameToPixels`

Introduced in R2015a

step

System object: `visionhdl.LookupTable`
Package: `visionhdl`

Map input pixel to output pixel based on table contents

Syntax

```
[pixelOut,ctrlOut] = step(LUT,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(LUT,pixelIn,ctrlIn)` returns the pixel value, `pixelOut`, located in the table at the address specified by the input pixel value, `pixelIn`. The object passes the control signals, `ctrlIn`, through and aligns 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as

dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`LUT` — Look up table
`visionhdl.LookupTable` System object

Specify a `visionhdl.LookupTable` System object that you created and configured.

`pixelIn` — Input pixel
scalar

Input pixel, specified as a scalar value. For unsigned fixed-point data types, the input word length must be less than or equal to 16.

Supported data types:

- `logical`
- `uint8` or `uint16`
- `fixdt()`

`ctrlIn` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel
scalar

Output pixel, returned as a scalar value. The data type of the output is the same as the data type of the entries you specify in the `Table` property.

`ctrlOut` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.MeasureTiming

Package: `visionhdl`

Measure timing of pixel control structure input

Description

The `visionhdl.MeasureTiming` 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”.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`measure = visionhdl.MeasureTiming` returns a System object, `measure`, that measures the average frame timing of a video stream.

Methods

step Measure timing of pixel control structure input

Common to All System Objects	
`release`	Allow System object property value changes

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.

See Also

Measure Timing | `visionhdl.FrameToPixels`

Introduced in R2016b

step

System object: `visionhdl.MeasureTiming`
Package: `visionhdl`

Measure timing of pixel control structure input

Syntax

```
[activePixels,activeLines,totalPixels,totalLines,horizBlank,vertBlan
k] = step(measure,ctrlIn)
```

Description

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[activePixels,activeLines,totalPixels,totalLines,horizBlank,vertBlank] = step(measure,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.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`measure` — Timing analyzer
`visionhdl.MeasureTiming` System object

Specify a `visionhdl.MeasureTiming` System object that you created and configured.

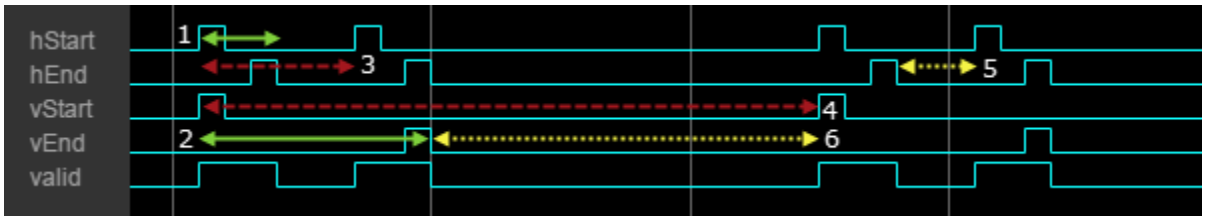
`ctrlIn` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel 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.



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.

`activePixels` — Number of active pixels per line
integer

This value is measured between `hStart` and `hEnd`. See marker `1` in the diagram.

`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.

`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.

`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.

`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.

`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.

Introduced in R2016b

visionhdl.MedianFilter

Package: `visionhdl`

2-D median filtering

Description

`visionhdl.MedianFilter` performs 2-D median filtering on a pixel stream. The object replaces each pixel value with the median value of the adjacent pixels.

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. The object accepts and returns 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".

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`MF = visionhdl.MedianFilter` returns a System object, `MF`, that performs two-dimensional median filtering of serial pixel data.

`MF = visionhdl.MedianFilter(Name,Value)` returns a median filter System object, `MF`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

`MF = visionhdl.MedianFilter(size,Name,Value)` returns a median filter System object, `MF`, with the `NeighborhoodSize` property set to `size` and additional options specified by one or more `Name`,`Value` pair arguments.

Input Arguments

`size`

Size in pixels of the image region used to compute the median. This argument sets the NeighborhoodSize property value.

Output Arguments

`MF`

`visionhdl.MedianFilter` System object.

Properties

`NeighborhoodSize`

Neighborhood size, in pixels.

- `'3×3'` (default)
- `'5×5'`
- `'7×7'`

`LineBufferSize`

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 single horizontal line.

Choose a power of 2 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 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 **Neighborhood size**.

Default: 2048

`PaddingMethod`

Method for padding the boundary of the input image

- `'Constant'` — Pad input matrix with a constant value.
- `'Replicate'` — Repeat the value of pixels at the edge of the image.
- `'Symmetric'` (default) — Pad image edge with its mirror image.

`PaddingValue`

Constant value used to pad the boundary of the input image. This property applies when you set `PaddingMethod` to `'Constant'`. The object casts this value to the same data type as the input pixel.

Default: 0

Methods

step	Median pixel value of neighborhood

Common to All System Objects	
`release`	Allow System object property value changes

Examples

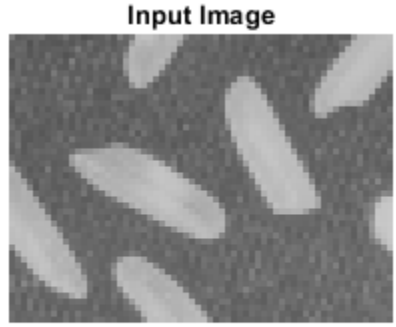
Median Filter on a Pixel-Stream

This example implements a 5×5 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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[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:

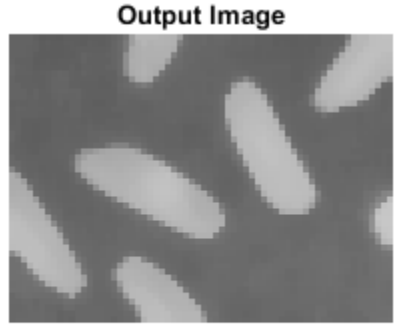
- The number of active pixels in a line.
- The size of the neighbourhood.

```
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 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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Median Filter block reference page.

See Also

Median Filter | `medfilt2` | `visionhdl.FrameToPixels`

Introduced in R2015a

step

System object: `visionhdl.MedianFilter`
Package: `visionhdl`

Median pixel value of neighborhood

Syntax

```
[pixelOut,ctrlOut] = step(medfilt,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(medfilt,pixelIn,ctrlIn)` returns the next pixel value, `pixelOut`, in the filtered pixel stream resulting from calculating the median of the neighborhood around each input pixel, `pixelIn`. Before filtering, the object pads 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as

dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`medfilt` — Filter
`visionhdl.MedianFilter` System object

Specify a `visionhdl.MedianFilter` System object that you created and configured.

`pixelIn` — Input pixel
scalar

Single pixel, specified as a scalar value.

Supported data types:

- `uint` or `int`
- `fixdt(~,N,0)`
- `logical`
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel
scalar

Single pixel value representing the median of its neighborhood, returned as a scalar value.

The data type is the same as the data type of `pixelIn`.

`ctrlOut` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.Opening

Package: `visionhdl`

Morphological opening of binary pixel data

Description

`visionhdl.Opening` performs morphological erosion, followed by morphological dilation, using the same neighborhood for both calculations. The object operates on a stream of binary intensity values.

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. The object accepts and returns 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".

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`O = visionhdl.Opening` returns a System object, `O`, that performs morphological opening on a binary pixel stream.

`O = visionhdl.Opening(Name,Value)` returns a System object, `O`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Neighborhood`

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The object supports neighborhoods of up to 32×32 pixels. To use a structuring element, specify `Neighborhood` as `getnhood(strel(`*shape*`))`.

Default: `ones(3,3)`

`LineBufferSize`

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 single horizontal line.

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 object uses the next largest power of two. The object allocates (*neighborhood lines* – 1)-by-`LineBufferSize` memory locations to store the pixels.

Default: `2048`

Methods

step	Report opened pixel value based on neighborhood

Common to All System Objects	
`release`	Allow System object property value changes

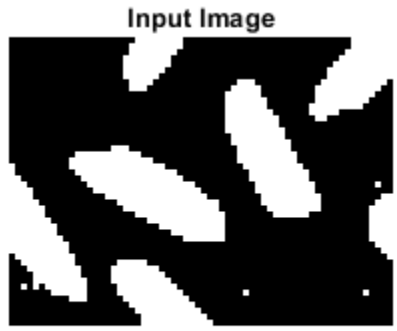
Examples

Morphological Open

Perform morphological open on a thumbnail image.

Load a source image from a file. Select a portion of the image that matches the desired test size. This source image contains `uint8` pixel intensity values. Apply a threshold to convert to binary pixel data.

```
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 object and define inactive pixel regions. Make the number of inactive pixels following each active line at least double the horizontal size of the neighborhood. Make the number of lines following each frame 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 filter object.

```
 mopen = 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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels 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

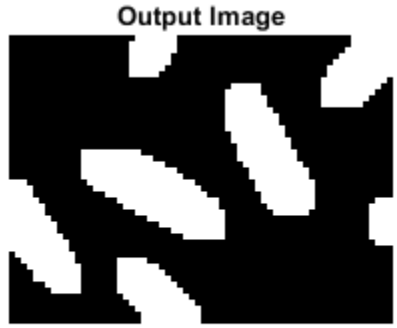
```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = mopen(pixIn(p),ctrlIn(p));
    if (ctrlOut(p).valid && foundValOut==0)
        foundValOut = p;
    end
end
sprintf('object latency is %d cycles',foundValOut-foundValIn)
```

```
ans =
'object latency is 368 cycles'
```

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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Opening block reference page.

See Also

Opening | `imopen` | `visionhdl.Closing` | `visionhdl.Dilation` | `visionhdl.Erosion` | `visionhdl.FrameToPixels`

Introduced in R2015a

step

System object: `visionhdl.Opening`
Package: `visionhdl`

Report opened pixel value based on neighborhood

Syntax

```
[pixelOut,ctrlOut] = step(open,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(open,pixelIn,ctrlIn)` returns the next pixel value, `pixelOut`, resulting from a morphological open operation on the neighborhood around each input 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable

property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`open` — Morphological opener

`visionhdl.Opening` System object

Specify a `visionhdl.Opening` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar `logical` value.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

scalar

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

`ctrlOut` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2015a

visionhdl.GrayscaleOpening

Package: `visionhdl`

Morphological opening of grayscale pixel data

Description

`visionhdl.GrayscaleOpening` performs morphological erosion, followed by morphological dilation, 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 32×32 pixels. For line, square, or rectangle structuring elements more than 8 pixels wide, the object uses the Van Herk algorithm to find the maximum and minimum. For structuring elements less than 8 pixels wide, or that contain zero elements, the object implements a pipelined comparison tree to find the maximum and minimum.

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. The object accepts and returns 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”.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`O = visionhdl.GrayscaleOpening` returns a System object, `O`, that performs morphological opening on a pixel stream.

`O = visionhdl.GrayscaleOpening(Name,Value)` returns a System object, `O`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name on page 2-209 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`Neighborhood`

Pixel neighborhood, specified as a vector or matrix of `1`s and `0`s.

The object supports neighborhoods of up to 32×32 pixels. To use a structuring element, specify `Neighborhood` as `getnhood(strel(`*`shape`*`))`. The minimum neighborhood size is a 2×2 matrix, or a 2×1 column vector. If the neighborhood is a row vector, it must be at least 8 columns wide and contain no zeros.

Default: `ones(3,3)`

`LineBufferSize`

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 single horizontal line.

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 object uses the next largest power of two. The object allocates (*neighborhood lines* – 1)-by-`LineBufferSize` memory locations to store the pixels.

Default: `2048`

Methods

step	Report opened pixel value based on neighborhood

Common to All System Objects	
`release`	Allow System object property value changes

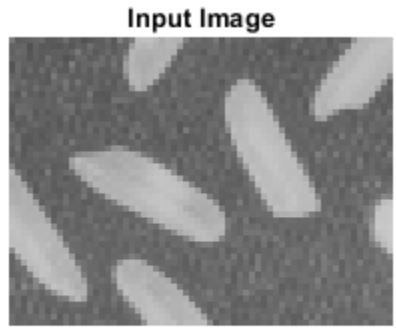
Examples

Grayscale Morphological Opening

Perform morphological opening 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'
```



Create a serializer object and define the inactive pixel regions. Make the number of inactive pixels following each active line at least double the horizontal size of the neighborhood. Make the number of lines following each frame 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.

```
mopen = 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.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[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);
```

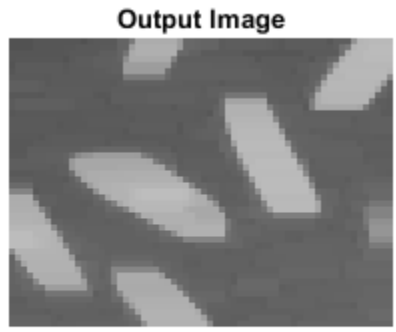
For each pixel in the padded frame, compute the morphed value. Monitor the control signals to determine the latency of the object. The latency of a configuration depends on the number of active pixels in a line and the size of the neighborhood.

```
foundValIn = false;
foundValOut = false;
for p = 1:numPixelsPerFrame
    if (ctrlIn(p).valid && foundValIn==0)
        foundValIn = p;
    end
    [pixOut(p),ctrlOut(p)] = mopen(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 216 cycles'
```

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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



Algorithms

This object implements the algorithms described on the Grayscale Opening block reference page.

See Also

Grayscale Opening | `imopen` | `visionhdl.FrameToPixels` | `visionhdl.GrayscaleClosing` | `visionhdl.GrayscaleDilation` | `visionhdl.GrayscaleErosion`

Introduced in R2016a

step

System object: `visionhdl.GrayscaleOpening`
Package: `visionhdl`

Report opened pixel value based on neighborhood

Syntax

```
[pixelOut,ctrlOut] = step(open,pixelIn,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,ctrlOut] = step(open,pixelIn,ctrlIn)` returns the next pixel value, `pixelOut`, resulting from morphological opening on 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. The object accepts and returns 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”.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable

property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`open` — Morphological opener

`visionhdl.GrayscaleOpening` System object

Specify a `visionhdl.GrayscaleOpening` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Single pixel, specified as a scalar value.

Supported data types:

- `uint8`, `uint16`,`uint32`
- `fixdt(0,N,M)`
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

Output Arguments

`pixelOut` — Output pixel

scalar

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

The data type is the same as the data type of `pixelIn`.

`ctrlOut` — Control signals accompanying pixel stream
`pixelcontrol` structure

Control signals accompanying the 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`

Introduced in R2016a

visionhdl.PixelStreamAligner

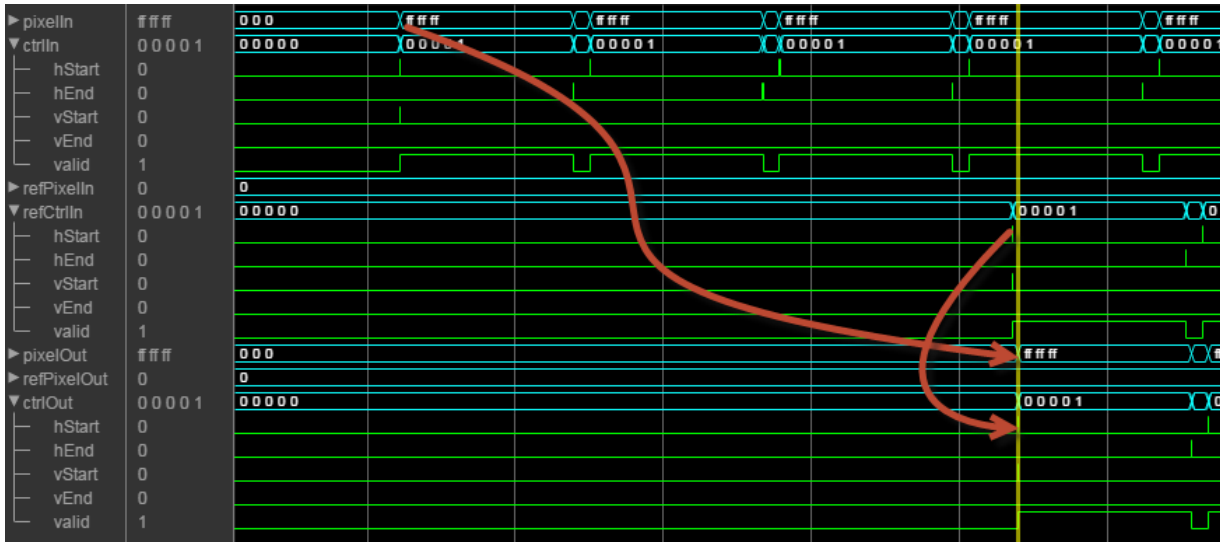
Package: `visionhdl`

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 `refPixelIn`, 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. There is a 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 `MaxNumberofLines` 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”.

Construction

`align = visionhdl.PixelStreamAligner` returns a System object, `align`, that synchronizes a pixel stream with a reference pixel stream.

Properties

`LineBufferSize` — Size of the line memory buffer

2048 (default) | scalar 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 object uses the next largest power of two. The object implements a circular buffer of 2^M, where M is `MaxNumberofLines` + $\log_2$(`LineBufferSize`) pixels.

`MaxNumberOfLines` — Buffer depth that accommodates the timing offset between input streams
10 (default) | scalar integer

The object implements a circular buffer of 2^M, where M is `MaxNumberofLines` + $\log_2$(`LineBufferSize`) pixels, 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 parameter must accommodate the difference in timing between the two input streams, including 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.

Methods

step	Align two streams of pixel data

Common to All System Objects	
`release`	Allow System object property value changes

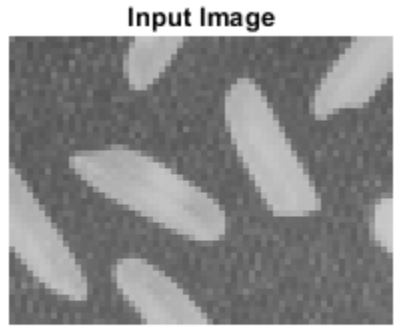
Examples

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'
```



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);
```

Serialize the test image 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);
overlayOut = zeros(numPixelsPerFrame,1,'uint8');
```

Write a function that creates and calls the System objects to detect edges and then align the edge data with the original pixel data. The edge results are delayed by the latency of the `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
% Detects edges in an input stream, and overlays the edge data onto the
% original stream.
% pixelIn, ctrlIn are a scalar pixel and its associated pixelcontrol structure
% 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
```

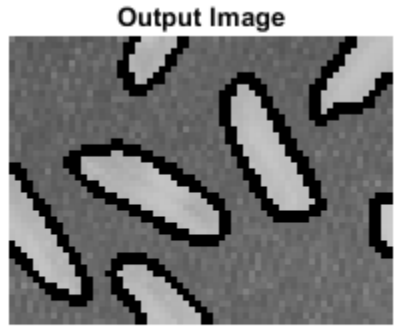
For each pixel in the frame, call your function to search for edges and align with the input stream.

```
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);
[frmOutput,frmValid] = pix2frm(overlayOut,ctrlOut);
```

```
if frmValid
    figure
    imshow(frmOutput, 'InitialMagnification',300)
    title 'Output Image'
end
```



See Also

Blocks
Pixel Stream Aligner

Objects
`visionhdl.FrameToPixels`

Introduced in R2017a

step

System object: `visionhdl.PixelStreamAligner`
Package: `visionhdl`

Align two streams of pixel data

Syntax

```
[pixelOut,refOut,ctrlOut] = step(align,pixelIn,ctrlIn,refPixel,
refCtrl)
```

Description

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixelOut,refOut,ctrlOut] = step(align,pixelIn,ctrlIn,refPixel, refCtrl)` synchronizes a pixel stream to a reference stream by delaying the first input, `pixel`, to align with the reference input, `refPixel`. The resulting aligned pixel streams share control signals. You can use this object to align streams for overlay or comparison.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`align` — Aligner
`visionhdl.PixelStreamAligner` System object

Specify a `visionhdl.PixelStreamAligner` System object that you created and configured.

`pixelIn` — Input pixel

scalar | vector

Input pixel, specified as a vector of three values representing R'G'B' or Y'CbCr, or a scalar value representing intensity. The object delays this pixel stream to match the control signals of the reference stream, `refPixel`. Therefore, `pixelIn` must be the earlier of the two streams.

Supported data types:

- `logical`
- `uint` and `int`
- `fixdt()`
- `double` and `single` data types are supported for simulation, but not for HDL code generation.

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

`refPixel` — Input pixel

scalar | vector

Input pixel, specified as a vector of three values representing R'G'B' or Y'CbCr, or a scalar value representing intensity. The object delays the `pixelIn` input stream to match the reference control signals. Therefore, `refPixel` must be the later of the two streams. The reference data and its control signals pass through the object with a small delay.

Supported data types:

- `logical`
- `uint` and `int`

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

`refCtrl` — Pixel stream control signals
structure

Control signals indicating the validity of the reference pixel and the location of the pixel within the frame, specified as a structure containing five `logical` signals. The object uses these control signals for the aligned output stream. See “Pixel Control Structure”.

Output Arguments

`pixelOut` — Output pixel
scalar | vector

Single pixel, returned as a vector of three values representing R'G'B' or Y'CbCr, or a scalar value representing intensity.

The data type is the same as the data type of `pixelIn`.

`refOut` — Output reference pixel
scalar | vector

Single pixel, returned as a vector of three values representing R'G'B' or Y'CbCr, or a scalar value representing intensity.

The data type is the same as the data type of `refIn`.

`ctrlOut` — Pixel stream control signals
structure

Control signals for both output streams, returned as a structure containing five `logical` signals. See “Pixel Control Structure”. These signals are the same as the `refCtrl` input.

Introduced in R2017a

visionhdl.PixelsToFrame

Package: `visionhdl`

Convert pixel stream to frame-based video

Description

`visionhdl.PixelsToFrame` 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 specifying custom dimensions. See “Streaming Pixel Interface” for details of the pixel stream format.

Use this object to convert the output of a function targeted for HDL code generation back to frames. This object 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.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`P2F = visionhdl.PixelsToFrame` returns a System object, `P2F`, that converts a 1080p pixel stream, with standard padding, to a grayscale 1080×1920 frame.

`P2F = visionhdl.PixelsToFrame(Name,Value)` returns a System object, `P2F`, with additional options specified by one or more `Name,Value` pair arguments. `Name` is a property name on page 2-227 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order

as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`NumComponents`

Components of each pixel, specified as 1, 3, or 4. Set to 1 for grayscale video. Set to 3 for color video, for example, {R,G,B} or {Y,Cb,Cr}. Set to 4 to use color with an alpha channel for transparency. The `visionhdl.PixelsToFrame` object expects a matrix of *P*-by-`NumComponents` values, where *P* is the total number of pixels. The default is 1.

`NumPixels`

Number of pixels transferred on the streaming interface for each cycle, specified as `1`, `4`, or `8`. The default is `1`. To enable multipixel streaming and increase throughput for high-resolution or high-frame-rate video, set this property to `4` or `8`. The `visionhdl.PixelsToFrame` object expects a *P*-by-`NumPixels` matrix, where *P* is the total number of pixels. When you set `NumPixels>1`, you must set the `NumComponents` property to `1`.

Note You can simulate System objects with a multipixel streaming interface, but they are not supported for HDL code generation. Use the equivalent blocks to generate HDL code for multipixel algorithms.

`VideoFormat`

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

Video Format	Active Pixels Per Line	Active Video Lines
`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

Methods

step Convert pixel stream to image frame

Common to All System Objects	
`release`	Allow System object property value changes

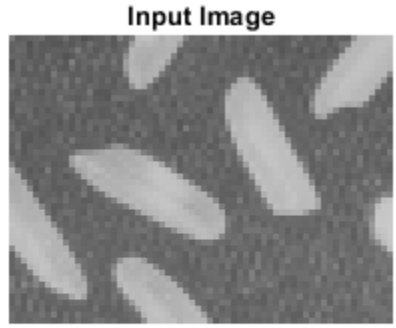
Examples

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 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 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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[pixIn,ctrlIn] = frm2pix(frmInput);
```

Prepare to process pixels by preallocating output vectors.

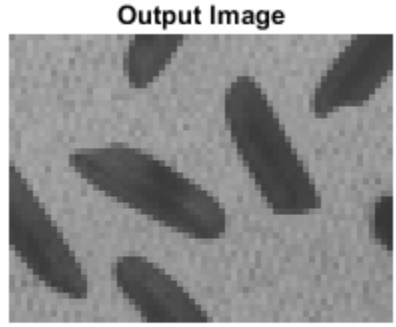
```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

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 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);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput,'InitialMagnification',300)
    title 'Output Image'
end
```



See Also

Pixels To Frame | `visionhdl.FrameToPixels`

Topics

“Pixel-Streaming Design in MATLAB”
“Streaming Pixel Interface”

Introduced in R2015a

step

System object: `visionhdl.PixelsToFrame`
Package: `visionhdl`

Convert pixel stream to image frame

Syntax

```
[frm,validOut] = step(P2F,pixels,ctrlIn)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[frm,validOut] = step(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`.

See “Streaming Pixel Interface” for details of the pixel stream format.

Note The System object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

P2F — Deserializer

`visionhdl.PixelsToFrame` System object

Specify a `visionhdl.PixelsToFrame` System object that you created and configured.

`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, calculated as `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 they are not supported for HDL code generation. Use the equivalent blocks to generate HDL code for multipixel algorithms.

Set the size of the padded image using the `VideoFormat` property. If the number of elements in `pixels` does not match that specified by `VideoFormat`, The object returns a warning.

Supported data types:

- `uint` or `int`
- `fixdt()`
- `logical`
- `double` or `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, calculated as `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”. If the dimensions indicated by `ctrlIn` do not match that specified by `VideoFormat`, the object returns a warning.

Output Arguments

`frm` — Image frame

matrix

Image frame, 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

Set the size of the active image using the VideoFormat property. The data type of the pixel values is the same as `pixels`.

`validOut` — Indicates valid output data

scalar `logical`

Frame status, returned as a logical value. When `validOut` is `true`, the frame is reassembled and ready for use.

Introduced in R2015a

visionhdl.ROISelector

Package: `visionhdl`

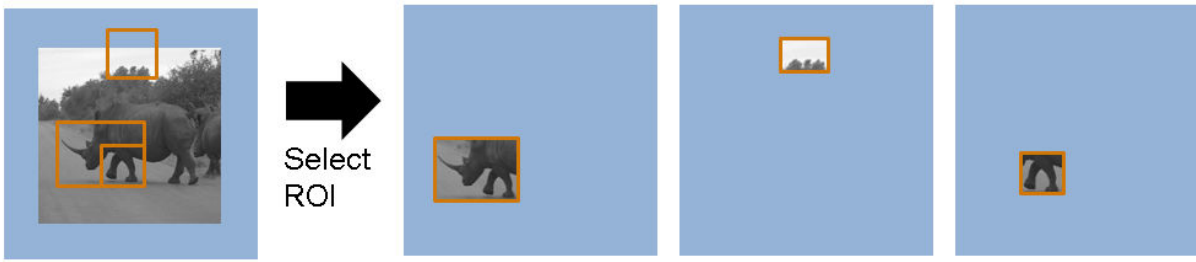
Select region of interest (ROI) from pixel stream

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 control signals indicate a new active region of the frame. The 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 new frame, or select the frame location in real time via an input argument. You can select more than one region. Define each region by the upper-left corner coordinates and the 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. The diagram shows the output frames for three requested

regions. The second output region (treetops) does not include the inactive region above the frame.



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. The object accepts and returns 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".

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ROI = visionhdl.ROISelector` returns a System object, `ROI`, that selects a default region of the active frame from an input stream.

`ROI = visionhdl.ROISelector(Name,Value)` returns a System object, `ROI`, with additional options specified by one or more `Name`,`Value` pair arguments. `Name` is a property name on page 2-237 and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. Properties not specified retain their default values.

Properties

`RegionsSource`

Location of the output region definitions.

`'Property'` — Specify the regions in the `Regions` property.

`'Input port'` — Specify the regions using arguments to the `step` method. Each argument is a 1-by-4 vector specifying coordinates for a single region. The object captures the value of the `region` input ports when it receives `vStart` set to `true` in the input control structure.

Default: `'Property'`

`Regions`

Rectangular regions of interest to select from the input frame, specified as a *N*-by-4 matrix.

N is the number of regions. You can select up to 16 regions. The four elements that define each region are the top-left starting coordinates and the dimensions, `[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. The regions are independent of each other, so they can overlap. This property applies when you set `RegionsSource` to `'Property'`.

Default: `[100 100 50 50]`

`NumberofRegions`

Number of `region` arguments to the `step` method, specified as a positive integer.

You can select up to 16 regions. This property applies when you set `RegionsSource` to `'Input port'`.

Default: 1

Methods

step	Return next pixel in reselected frame

Common to All System Objects	
`release`	Allow System object property value changes

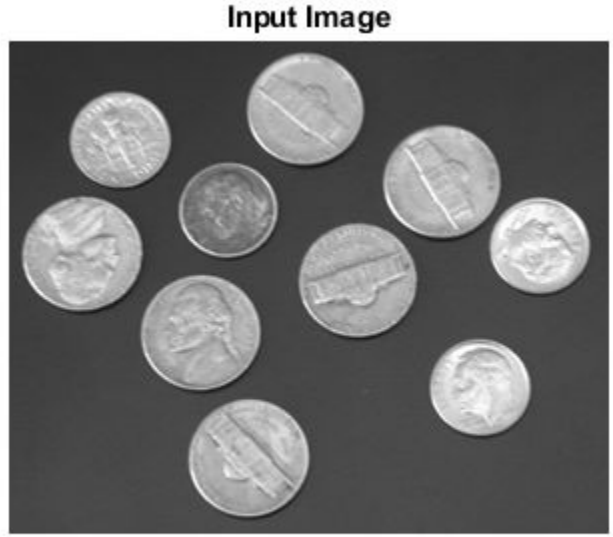
Examples

Select Region of Interest

Select a fixed region of interest (ROI) from an input frame.

Load a source image from a file.

```
frmOrig = imread('coins.png');
[frmActiveLines,frmActivePixels] = size(frmOrig);
imshow(frmOrig)
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+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 the top-left corner and the dimensions.

```
hPos = 80;
vPos = 60;
hSize = 65;
vSize = 50;
roicoin = visionhdl.ROISelector('Regions',[hPos vPos hSize vSize])
```

```
roicoin =
  visionhdl.ROISelector with properties:

    RegionsSource: 'Property'
          Regions: [80 60 65 50]
```

Serialize the test image by calling `step` on the serializer object. `pixIn` is a vector of intensity values. `ctrlIn` is a vector of control signal structures.

Note: This syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
[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.

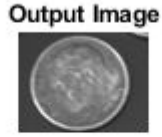
```
[~,~,numPixelsPerFrame] = getparamfromfrm2pix(frm2pix);
pixOut = uint8(zeros(numPixelsPerFrame,1));
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
```

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 format matching the new region. Convert the pixel stream to an image frame by calling `step` on the deserializer object. Display the resulting image.

```
pix2frm = visionhdl.PixelsToFrame(...
      'NumComponents',1,...
      'VideoFormat','custom',...
      'ActivePixelsPerLine',hSize,...
      'ActiveVideoLines',vSize);
[frmOutput,frmValid] = pix2frm(pixOut,ctrlOut);
if frmValid
    figure
    imshow(frmOutput)
    title 'Output Image'
end
```



Algorithm

The generated HDL code for the `visionhdl.ROISelector` System object uses two 32-bit counters. It does not use additional counters for additional regions.

Latency

The object has a latency of three cycles. The object returns the output pixel and associated control signals on the third call to the `step` method after the pixel value was applied.

See Also

ROI Selector | `visionhdl.FrameToPixels`

Introduced in R2016a

step

System object: `visionhdl.ROISelector`
Package: `visionhdl`

Return next pixel in reselected frame

Syntax

```
[pixel1,ctrl1] = step(ROI,pixelIn,ctrlIn)
[pixel1,ctrl1,...,pixelN,ctrlN] = step(ROI,pixelIn,ctrlIn)
[pixel1,ctrl1,...,pixelN,ctrlN] =
step(ROI,pixelIn,ctrlIn,region1,...,regionN)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[pixel1,ctrl1] = step(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 1-by-4 vector, `[hPos vPos hSize vSize]`.

`[pixel1,ctrl1,...,pixelN,ctrlN] = step(ROI,pixelIn,ctrlIn)` returns the next pixel values, `pixel1,...,pixelN`, and control signals, `ctrl1,...,ctrlN`, of each stream resulting from masking the active image frame into 1 to *N* new active regions, as directed by the `Regions` property. Set the `Regions` property to a *N*-by-4 matrix of region coordinates.

`[pixel1,ctrl1,...,pixelN,ctrlN] = step(ROI,pixelIn,ctrlIn,region1,...,regionN)` returns the next pixel values of each stream, `pixel1,...,pixelN`, resulting from masking the active image frame into 1

to *N* new regions, as directed by the `region1,...,regionN` arguments. Each `region` input is a 1-by-4 vector of region coordinates. Use this syntax when you set the `RegionsSource` property to `'Input Port'`, and the `NumberOfRegions` property to *N*.

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. The object accepts and returns 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".

Note The object performs an initialization the first time you call the `step` method. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, first call the `release` method to unlock the object.

Input Arguments

`ROI` — Region-of-interest selector

`visionhdl.ROISelector` System object

Specify a `visionhdl.ROISelector` System object that you created and configured.

`pixelIn` — Input pixel

scalar

Input pixel, specified as a scalar integer value.

- Any `numeric` data type.

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

`ctrlIn` — Control signals accompanying pixel stream

`pixelcontrol` structure

Control signals accompanying pixel stream, specified as a `pixelcontrol` structure containing five `logical` data type signals. The signals describe the validity of the pixel and the location of the pixel in the frame. For more details, see “Pixel Control Structure”.

Data Types: `struct`

`region1,...,regionN` — Regions of interest coordinates
1-by-4 vector

The four elements that define each region are `[hPos vPos hSize vSize]`. Use this argument when `RegionsSource` is set to `'Input port'`. You can specify *N* regions, where *N* is the `NumberOfRegions` property value.

Output Arguments

`pixel1,...,pixelN` — Output pixels
scalar

Output pixels, specified as 1 to *N* scalar integers.

If you set `RegionsSource` to `'Input port'`, *N* is the value in `NumberOfRegions`. If you set `RegionsSource` to `'Property'`, *N* is the number of columns in the `Regions` property.

`ctrl1,...,ctrlN` — Pixel stream control signals
structure

Control signals indicating the validity of each output pixel and the location of each pixel within the frame, returned as 1 to *N* structures of five `logical` signals. See “Pixel Control Structure”.

If you set `RegionsSource` to `'Input port'`, *N* is the value in `NumberOfRegions`. If you set `RegionsSource` to `'Property'`, *N* is the number of columns in the `Regions` property.

Introduced in R2016a

Functions — Alphabetical List

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.

See Also
Frame To Pixels | Pixels To Frame

Topics

“Streaming Pixel Interface”

Introduced in R2015a

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'
```

See Also

“Pixel Control Bus” | Frame To Pixels | Pixels To Frame

Topics

“Configure the Simulink Environment for HDL Video Processing”
“Streaming Pixel Interface”

Introduced in R2015a

pixelcontrolsignals

Extract signals from pixel-streaming control signal structure

Syntax

```
[hStart,hEnd,vStart,vEnd,valid] = pixelcontrolsignals(ctrl)
```

Description

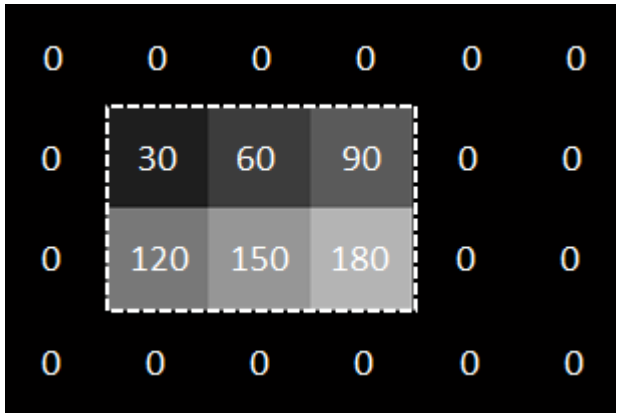
`[hStart,hEnd,vStart,vEnd,valid] = pixelcontrolsignals(ctrl)` extracts five scalar `logical` control signals from a structure.

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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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
```

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.

See Also

`pixelcontrolstruct` | `visionhdl.FrameToPixels` | `visionhdl.PixelsToFrame`

Topics

“Streaming Pixel Interface”

Introduced in R2015a

pixelcontrolstruct

Create pixel-streaming control signal structure

Syntax

```
ctrl = pixelcontrolstruct(hStart,hEnd,vStart,vEnd,valid)
```

Description

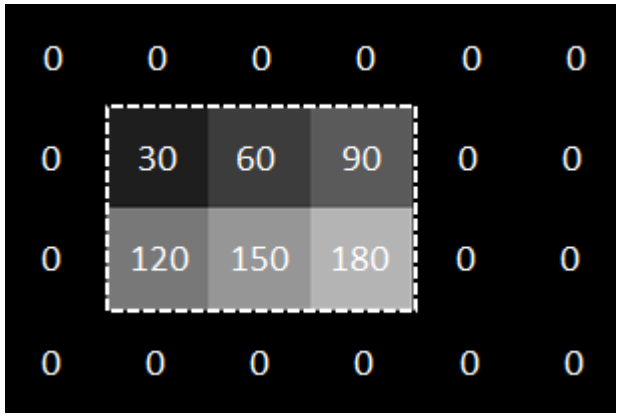
`ctrl = pixelcontrolstruct(hStart,hEnd,vStart,vEnd,valid)` creates a structure containing the five control signals used by Vision HDL Toolbox objects. The input arguments must be five scalars of `logical` type. See “Pixel Control Structure”.

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 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.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
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
```

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”.

See Also

`pixelcontrolsignals` | `visionhdl.FrameToPixels` | `visionhdl.PixelsToFrame`

Topics

“Streaming Pixel Interface”

Introduced in R2015a

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`

See Also

`hdlsetup`

Introduced in R2019b