# Vision HDL Toolbox™

User's Guide

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™ User's Guide*

**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

## 1 Streaming Pixel Interface

## 2 Algorithms

# 3 Code Generation and Deployment

# 4 Examples

# Streaming Pixel Interface

# Streaming Pixel Interface

**In this section...**

## What Is a Streaming Pixel Interface?

In hardware, processing an entire frame of video at one time has a high cost in memory and area. To save resources, serial processing is preferable in HDL designs. Vision HDL Toolbox blocks and System objects operate on a pixel, line, or neighborhood rather than a frame. The blocks and objects accept and generate video data as a serial stream of pixel data and control signals. The control signals indicate the relative location of each pixel within the image or video frame. The protocol mimics the timing of a video system, including inactive intervals between frames. Each block or object operates without full knowledge of the image format, and can tolerate imperfect timing of lines and frames.

All Vision HDL Toolbox blocks and System objects support single pixel streaming (with 1 pixel per cycle). Some blocks and System objects also support multipixel streaming (with 4 or 8 pixels per cycle) for high-rate or high-resolution video. Multipixel streaming increases hardware resources to support higher video resolutions with the same hardware clock rate as a smaller resolution video. HDL code generation for multipixel streaming is not supported with System objects. Use the equivalent blocks to generate HDL code for multipixel algorithms.

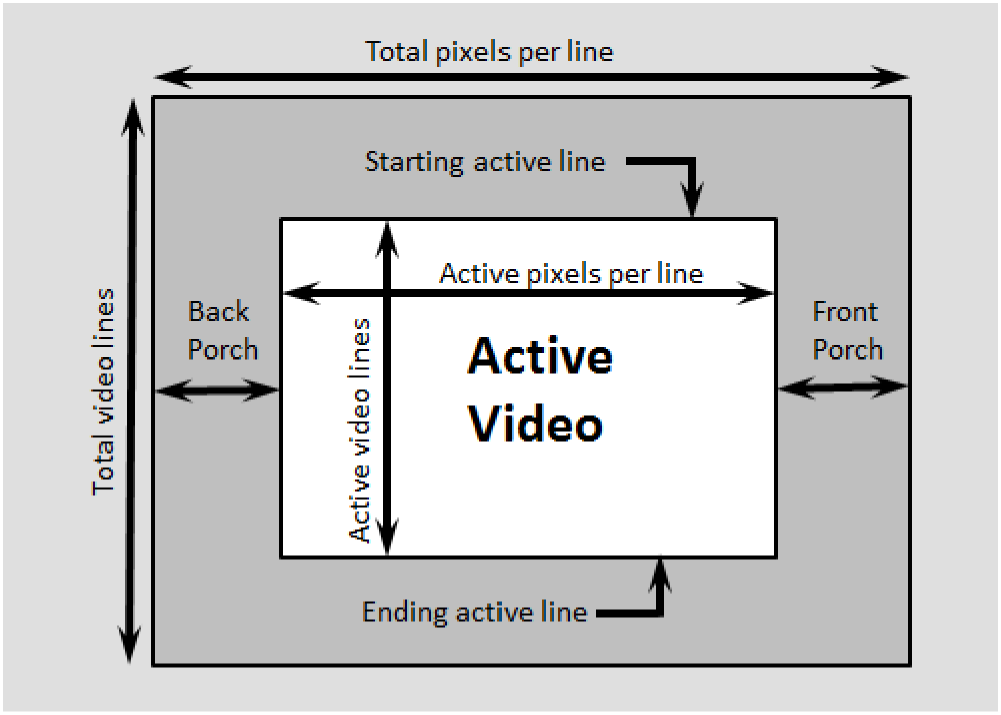
## How Does a Streaming Pixel Interface Work?

Video capture systems scan video signals from left to right and from top to bottom. As these systems scan, they generate inactive intervals between lines and frames of active video.

The *horizontal blanking* interval is made up of the inactive cycles between the end of one line and the beginning of the next line. This interval is often split into two parts: the *front* *porch* and the *back porch*. These terms come from the synchronize pulse between lines in analog video waveforms. The *front porch* is the number of samples between the end of the active line and the synchronize pulse. The *back porch* is the number of samples between the synchronize pulse and the start of the active line.

The *vertical blanking* interval is made up of the inactive cycles between the *ending active line* of one frame and the *starting active line* of the next frame.

The scanning pattern requires start and end signals for both horizontal and vertical directions. The Vision HDL Toolbox streaming pixel protocol includes the blanking intervals, and allows you to configure the size of the active and inactive frame.



## Why Use a Streaming Pixel Interface?

### Format Independence

The blocks and objects using this interface do not need a configuration option for the exact image size or the size of the inactive regions. In addition, if you change the image format for your design, you do not need to update each block or object. Instead, update the image parameters once at the serialization step. Some blocks and objects still require a line buffer size parameter to allocate memory resources.

By isolating the image format details, you can develop a design using a small image for faster simulation. Then once the design is correct, update to the actual image size.

### Error Tolerance

Video can come from various sources such as cameras, tape storage, digital storage, or switching and insertion gear. These sources can introduce timing problems. Human vision cannot detect small variance in video signals, so the timing for a video system does not need to be perfect. Therefore, video processing blocks must tolerate variable timing of lines and frames.

By using a streaming pixel interface with control signals, each Vision HDL Toolbox block or object starts computation on a fresh segment of pixels at the start-of-line or start-of-frame signal. The computation occurs whether or not the block or object receives the end signal for the previous segment.

The protocol tolerates minor timing errors. If the number of valid and invalid cycles between start signals varies, the blocks or objects continue to operate correctly. Some Vision HDL Toolbox blocks and objects require minimum horizontal blanking regions to accommodate memory buffer operations.

## Pixel Stream Conversion Using Blocks and System Objects

In Simulink®, use the Frame To Pixels block to convert framed video data to a stream of pixels and control signals that conform to this protocol. The control signals are grouped in a nonvirtual bus data type called `pixelcontrol`. You can configure the block to return a pixel stream with 1, 4, or 8 pixels per cycle.

In MATLAB®, use the `visionhdl.FrameToPixels` object to convert framed video data to a stream of pixels and control signals that conform to this protocol. The control signals are grouped in a structure data type. You can configure the object to create a pixel stream with 1, 4, or 8 pixels per cycle.

If your input video is already in a serial format, you can design your own logic to generate `pixelcontrol` control signals from your existing serial control scheme. For example, see “Convert Camera Control Signals to pixelcontrol Format” on page 1-23 and “Integrate Vision HDL Blocks Into Camera Link System” on page 1-29.

### Supported Pixel Data Types

Vision HDL Toolbox blocks and objects include ports or arguments for streaming pixel data. Each block and object supports one or more pixel formats. The supported formats vary depending on the operation the block or object performs. This table details common video formats supported by Vision HDL Toolbox.

| Type of Video | Pixel Format |
|---|---|
| Binary | Each pixel is represented by a single `boolean` or `logical` value. Used for true black-and-white video. |
| Grayscale | Each pixel is represented by *luma*, which is the gamma-corrected luminance value. This pixel is a single unsigned integer or fixed-point value. |
| Color | Each pixel is represented by 2 to 4 unsigned integer or fixed-point values representing the color components of the pixel. Vision HDL Toolbox blocks and objects use gamma-corrected color spaces, such as R'G'B' and Y'CbCr.<br><br>To set up multipixel streaming for color video, use a separate Frame To Pixels block for each color component. For example, for a R'G'B' stream with 4 pixels per cycle, use three Frame To Pixels blocks to create three vectors of 4 pixels per cycle. The `pixelcontrol` bus for all three components is identical, so you need to carry only one bus forward through your design. |

Vision HDL Toolbox blocks have an input or output port, `pixel`, for the pixel data. Vision HDL Toolbox System objects expect or return an argument representing the pixel data. The following table describes the format of the pixel data.

| Port or Argument | Description | Data Type |
|---|---|---|
| `pixel` | • Single pixel streaming — A scalar that represents a binary or grayscale pixel value or a row vector of two to four values representing a color pixel<br>• Multipixel streaming — Column vector of four or eight pixel values<br><br>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. | Supported data types can include:<br>• `boolean` or `logical`<br>• `uint` or `int`<br>• `fixdt()`<br><br>`double` and `single` data types are supported for simulation, but not for HDL code generation. |

**Streaming Pixel Control Signals**

Vision HDL Toolbox blocks and objects include ports or arguments for control signals relating to each pixel. These five control signals indicate the validity of a pixel and its location in the frame. For multipixel streaming, each vector of pixel values has one set of control signals.

In Simulink, the control signal port is a nonvirtual bus data type called `pixelcontrol`. For details of the bus data type, see “Pixel Control Bus” on page 1-20.

In MATLAB, the control signal argument is a structure. For details of the structure data type, see “Pixel Control Structure” on page 1-22.

## Timing Diagram of Single Pixel Serial Interface

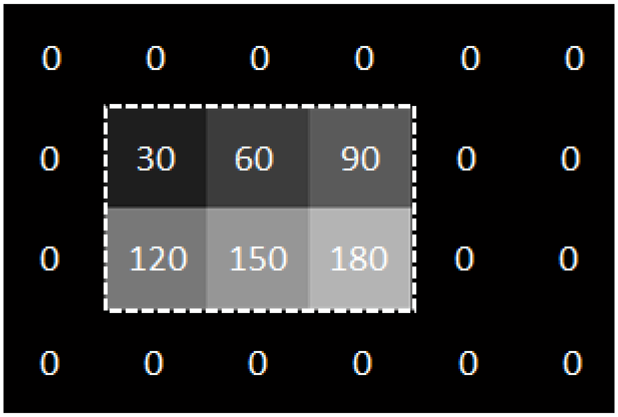
To illustrate the streaming pixel protocol, this example converts a frame to a sequence of control and data signals. Consider a 2-by-3 pixel image. To model the blanking intervals, configure the serialized image to include inactive pixels in these areas around the active image:

- 1-pixel-wide back porch
- 2-pixel-wide front porch
- 1 line before the first active line

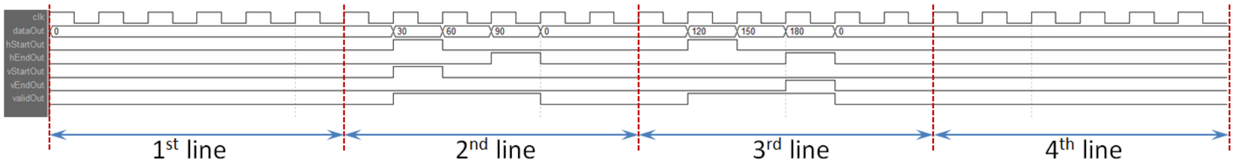
- 1 line after the last active line

You can configure the dimensions of the active and inactive regions with the Frame To Pixels block or the `visionhdl.FrameToPixels` object.

In the figure, the active image area is in the dashed rectangle, and the inactive pixels surround it. The pixels are labeled with their grayscale values.



The block or object serializes the image from left to right, one line at a time. The timing diagram shows the control signals and pixel data that correspond to this image, which is the serial output of the Frame To Pixels block for this frame, configured for single-pixel streaming.



For an example using the Frame to Pixels block to serialize an image, see “Design Video Processing Algorithms for HDL in Simulink”.

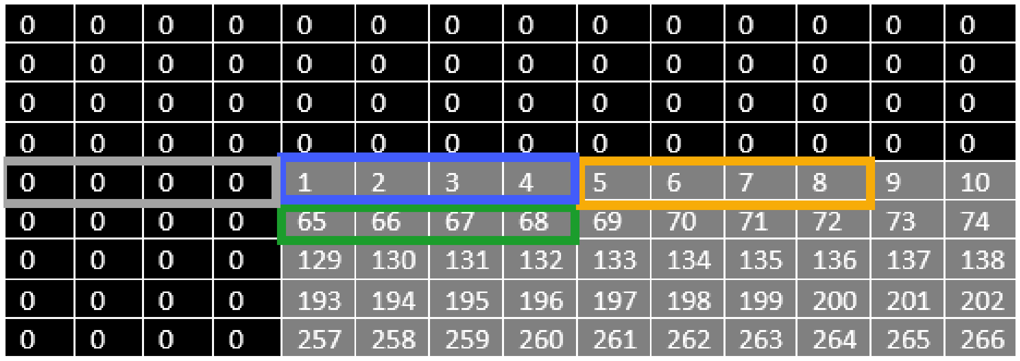
For an example using the `FrameToPixels` object to serialize an image, see “Design a Hardware-Targeted Image Filter in MATLAB”.

## Timing Diagram of Multipixel Serial Interface

This example converts a frame to a multipixel stream with 4 pixels per cycle and corresponding control signals. Consider a 64-pixel-wide frame with these inactive areas around the active image.

- 4-pixel-wide back porch
- 4-pixel-wide front porch
- 4 lines before the first active line
- 4 lines after the last active line

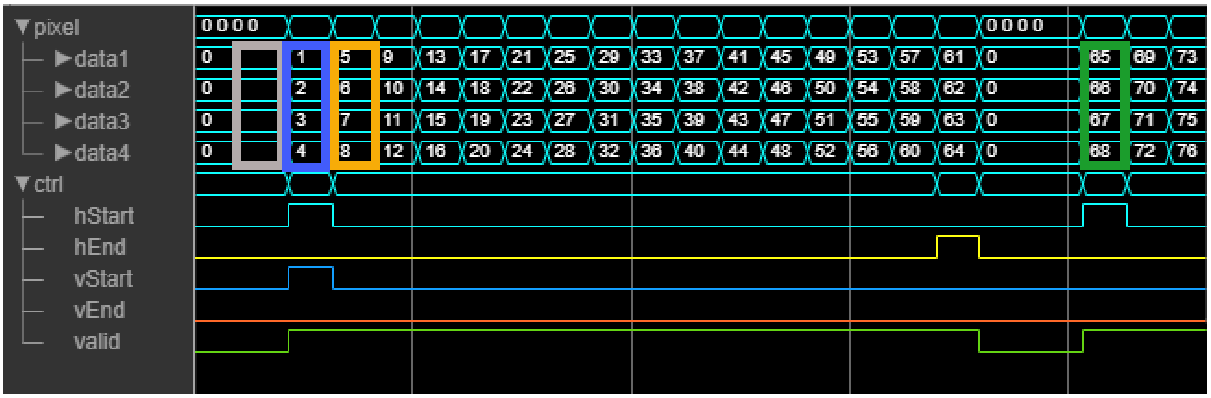
The Frame to Pixels block configured for multipixel streaming returns pixel vectors formed from the pixels of each line in the frame from left to right. This diagram shows the top-left corner of the frame. The gray pixels show the active area of the frame, and the zero-value pixels represent blanking pixels. The label on each active pixel represents the location of the pixel in the frame. The highlighted boxes show the sets of pixels streamed on one cycle. The pixels in the inactive region are also streamed four at a time. The gray box shows the four blanking pixels streamed the cycle before the start of the active frame. The blue box shows the four pixel values streamed on the first valid cycle of the frame, and the orange box shows the four pixel values streamed on the second valid cycle of the frame. The green box shows the first four pixels of the next active line.



| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0 | 0 | 0 | 0 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 |
| 0 | 0 | 0 | 0 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 |
| 0 | 0 | 0 | 0 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 |
| 0 | 0 | 0 | 0 | 257 | 258 | 259 | 260 | 261 | 262 | 263 | 264 | 265 | 266 |

This waveform shows the multipixel streaming data and control signals for the first line of the same frame, streamed with 4 pixels per cycle. The `pixelcontrol` signals that apply to each set of four pixel values are shown below the data 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.

Prior to the time period shown, the initial vertical blanking pixels are streamed four at a time, with all control signals set to `false`. This waveform shows the pixel stream of the first line of the image. The gray, blue, and orange boxes correspond to the highlighted areas of the frame diagram. After the first line is complete, the stream has two cycles of horizontal blanking that contains 8 invalid pixels (front and back porch). Then, the waveform shows the next line in the stream, starting with the green box.



For an example model that uses multipixel streaming, see “Filter Multipixel Video Streams” on page 1-10.

## See Also

Frame To Pixels | Pixels To Frame | `visionhdl.FrameToPixels` | `visionhdl.PixelsToFrame`

### Related Examples

- “Design Video Processing Algorithms for HDL in Simulink”
- “Design a Hardware-Targeted Image Filter in MATLAB”
- “Filter Multipixel Video Streams” on page 1-10

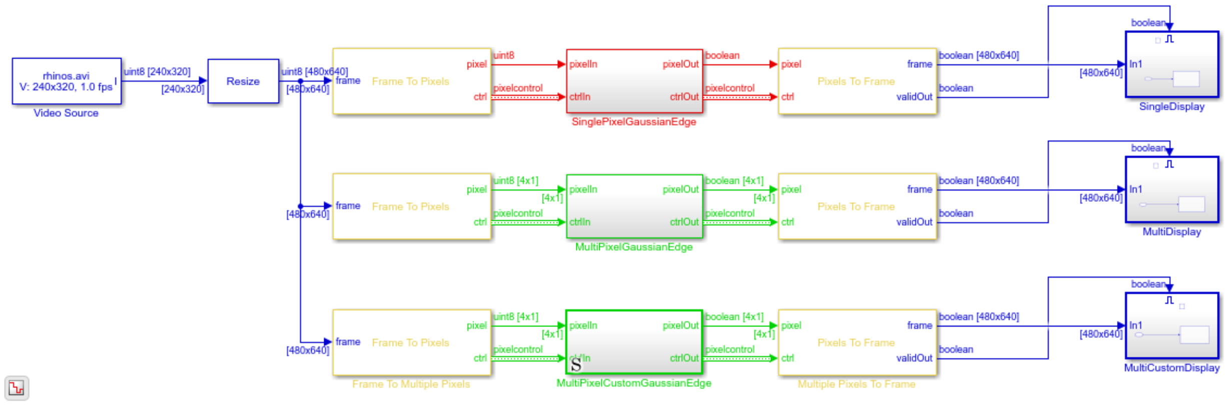
## Filter Multipixel Video Streams

This example shows how to design filters that operate on a multipixel input video stream. Use multipixel streaming to process high-resolution or high-frame-rate video with the same synthesized clock frequency as a single-pixel streaming interface. Multipixel streaming also improves simulation speed and throughput because fewer iterations are required to process each frame, while maintaining the hardware benefits of a streaming interface.

The example model has three subsystems which each perform the same algorithm:

- **SinglePixelGaussianEdge**: Uses the Image Filter and Edge Detector blocks to operate on a single-pixel stream. This subsystem shows how the rates and interfaces for single-pixel streaming compare with multipixel designs.
- **MultiPixelGaussianEdge**: Uses the Image Filter and Edge Detector blocks to operate on a multipixel stream. This subsystem shows how to use the multipixel interface with library blocks.
- **MultiPixelCustomGaussianEdge**: Uses the Line Buffer block to build a Gaussian filter and Sobel edge detection for a multipixel stream. This subsystem shows how to use the Line Buffer output for multipixel design.

Processing multipixel video streams allows for higher frame rates to be achieved without a corresponding increase to the clock frequency. Each of the subsystems can achieve 200MHz clock frequency on a Xilinx ZC706 board. The 480p video stream has **Total pixels per line** x **Total video lines** = 800*525 cycles per frame. With a single pixel stream you can process 200M/(800*525) = 475 frames per second. In the multipixel subsystem, 4 pixels are processed on each cycle, which reduces the number of cycles per line to 200. This means that with a multipixel stream operating on 4 pixels at a time, at 200MHz, on a 480p stream, 1900 frames can be processed per second. If the resolution is increased from 480p to 1080p, 80 frames per second can be achieved in the single pixel case versus 323 frames per second for 4 pixels at a time or 646 frames per second for 8 pixels at a time.

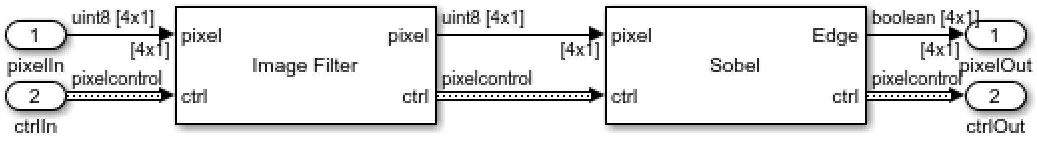


## Multipixel Streaming Using Library Blocks

Generate a multipixel stream from the Frame to Pixels block by setting **Number of pixels** to 4 or 8. The default value of 1 returns a scalar pixel stream with a sample rate of **Total pixels per line** * **Total video lines** faster than the frame rate. This rate shows red in the example model. The two multipixel subsystems use a multipixel stream with **Number of pixels** set to 4. This configuration returns 4 pixels on each clock cycle and has a sample rate of (**Total pixels per line**/4) * **Total video lines**. The lower output rate, which is green in the model, shows that you can increase either the input frame rate or resolution by a factor of 4 and therefore process 4 times as many pixels in the same frame period using the same clock frequency as the single pixel case.

The **SinglePixelGaussianEdge** and **MultiPixelGaussianEdge** subsystems compute the same result using the Image Filter and Edge Detector blocks.

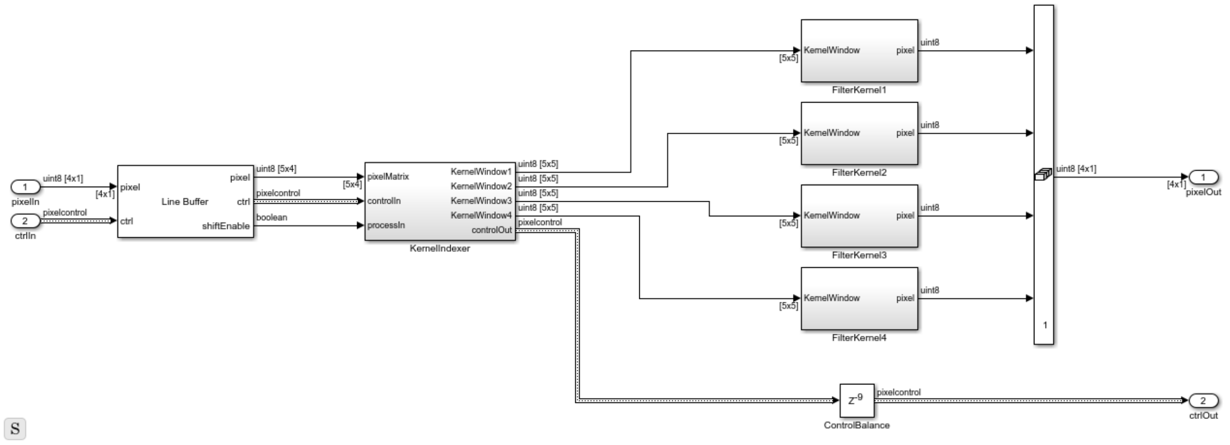
In **MultiPixelGaussianEdge**, the blocks accept and return four pixels on each clock cycle. You do not have to configure the blocks for multipixel streaming, they detect the input size on the port. The `pixelcontrol` bus indicates the validity and location in the frame of each set of four pixels. The blocks buffer the [4x1] stream to form four [ *KernelHeight* x *KernelWidth* ] kernels, and compute four convolutions in parallel to give a [4x1] output.



### Custom Multipixel Algorithms

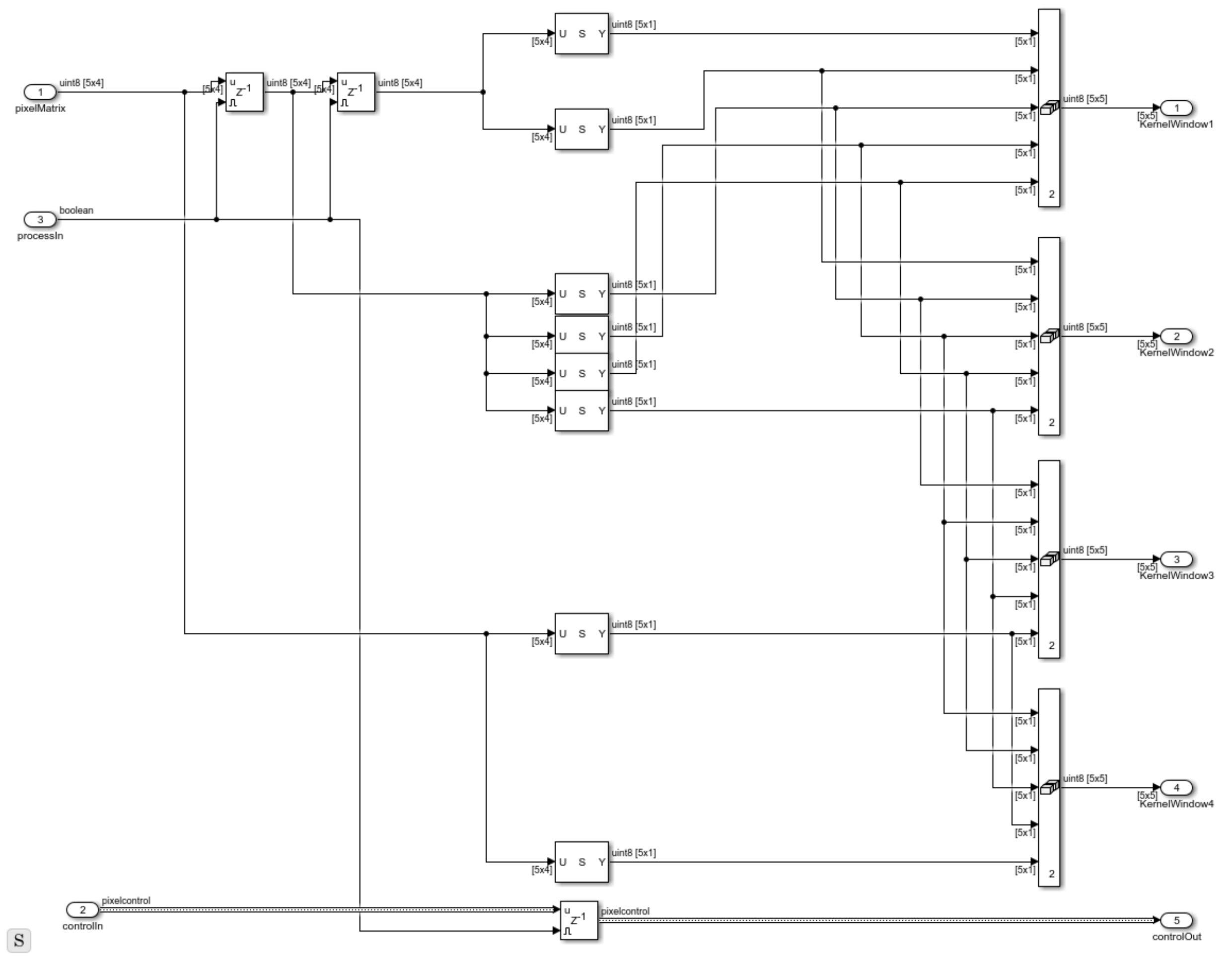
The **MultiPixelCustomGaussianEdge** subsystem uses the Line Buffer block to implement a custom filtering algorithm. This subsystem is similar to how the library blocks internally implement multipixel kernel operations. The Image Filter and Edge Detector blocks use more detailed optimizations than are shown here. This implementation shows a starting point for building custom multipixel algorithms using the output of the Line Buffer block.

The custom filter and custom edge detector use the Line Buffer block to return successive [ *KernelHeight* x *NumberofPixels* ] regions. Each region is passed to the KernelIndexer subsystem which uses buffering and indexing logic to form Number of Pixels * [ *KernelHeight* x *KernelWidth* ] filter kernels. Then each kernel is passed to a separate FilterKernel subsystem to perform convolutions in parallel.

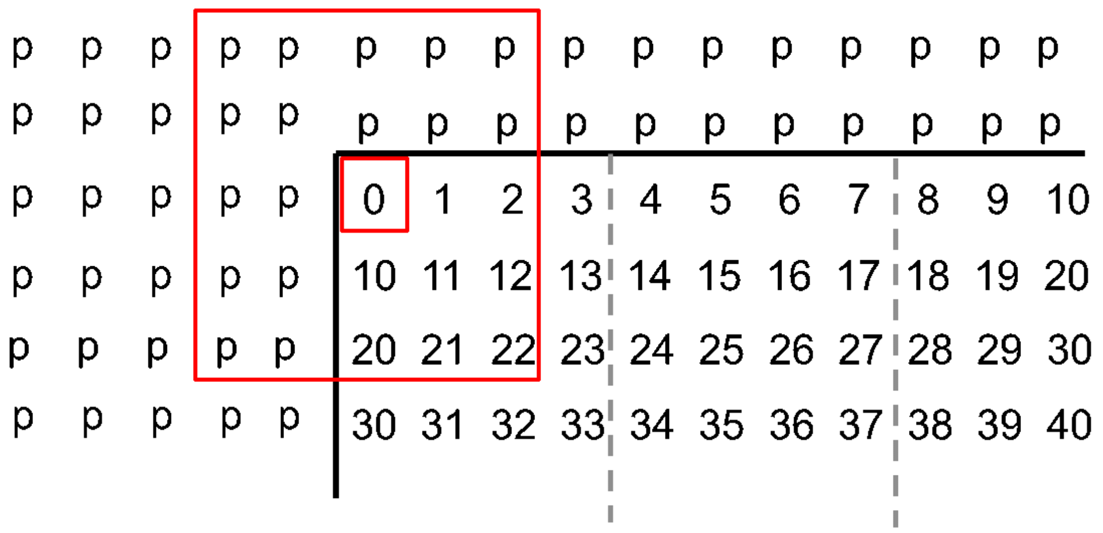


### Form Kernels from Line Buffer Output

The KernelIndexer subsystem forms 4 [5x5] filter kernels from the 2-D output of the Line Buffer block.

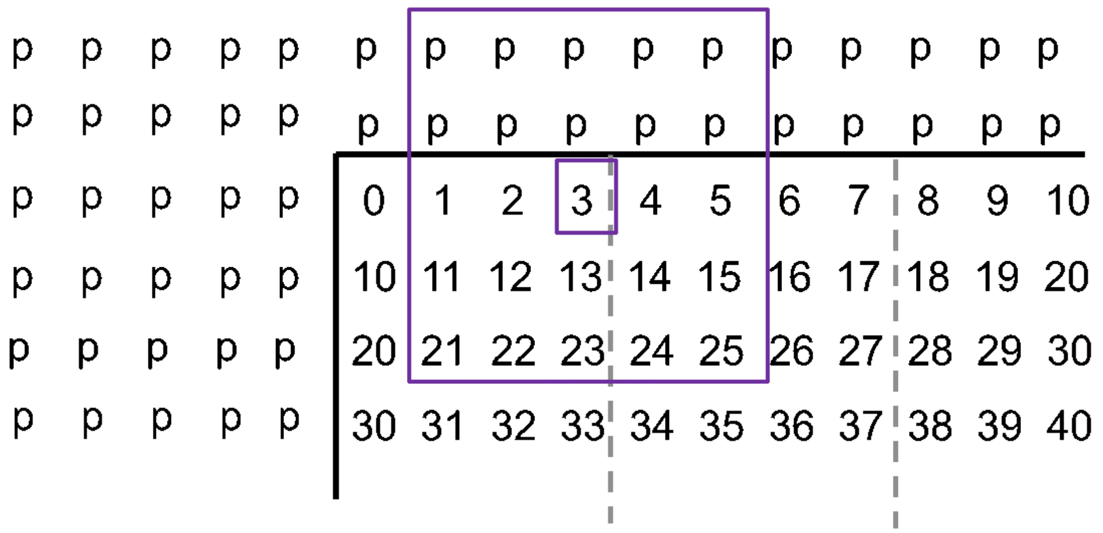


The diagram shows how the filter kernel is extracted from the [5x4] output stream, for the kernel that is centered on the first pixel in the [4x1] output. This first kernel includes pixels from 2 adjacent [5x4] Line Buffer outputs.



p p p p p p p p p p p p p p p p
p p p p p p p p p p p p p p p p
p p p p p 0 1 2 3 4 5 6 7 8 9 10
p p p p p 10 11 12 13 14 15 16 17 18 19 20
p p p p p 20 21 22 23 24 25 26 27 28 29 30
p p p p p 30 31 32 33 34 35 36 37 38 39 40

The kernel centered on the last pixel in the [4x1] output also includes the third adjacent [5x4] output. So, to form four [5x5] kernels, the subsystem must access columns from three [5x4] matrices.

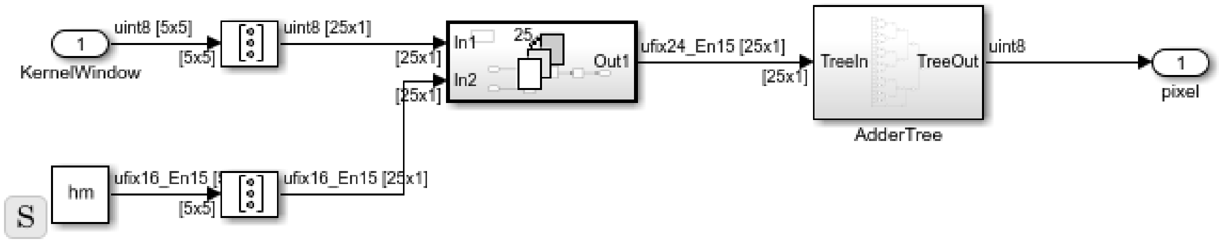


p p p p p p p p p p p p p p p p
p p p p p p p p p p p p p p p p
p p p p p 0 1 2 3 4 5 6 7 8 9 10
p p p p p 10 11 12 13 14 15 16 17 18 19 20
p p p p p 20 21 22 23 24 25 26 27 28 29 30
p p p p p 30 31 32 33 34 35 36 37 38 39 40

The KernelIndexer subsystem uses the current [5x4] input, and stores two more [5x4] matrices using registers enabled by `shiftEnable`. This design is similar to the tapped delay line used with a Line Buffer using single pixel streaming. The subsystem then accesses pixel data across the columns to form the four [5x5] kernels. The Image Filter block uses this same logic internally when the block has multipixel input. The block automatically designs this logic at compile time for any supported kernel size.

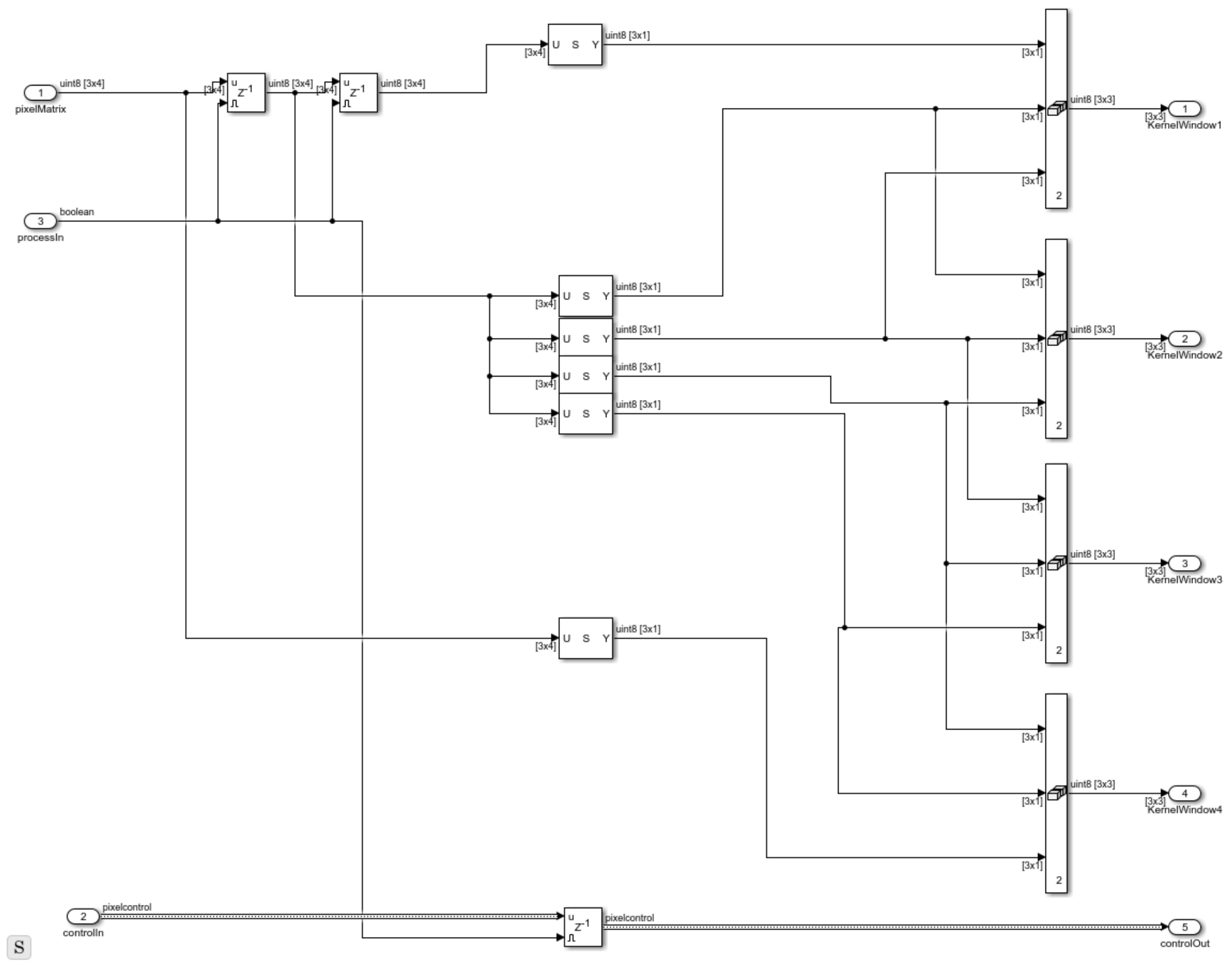
### Implement Filters

Since the input multipixel stream is a [4x1] vector, the filters must perform four convolutions on each cycle to keep pace with the incoming data. There are four parallel FilterKernel subsystems that each perform the same operation. The [5x5] matrix multiply is implemented as a [25x1] vector multiply by flattening the input matrix and using a For Each subsystem containing a pipelined multiplier. The output is passed to an adder tree. The adder tree is also pipelined, and the pipeline latency is applied to the `pixelcontrol` signal to match. The results of the four FilterKernel subsystems are then concatenated into a [4x1] output vector.



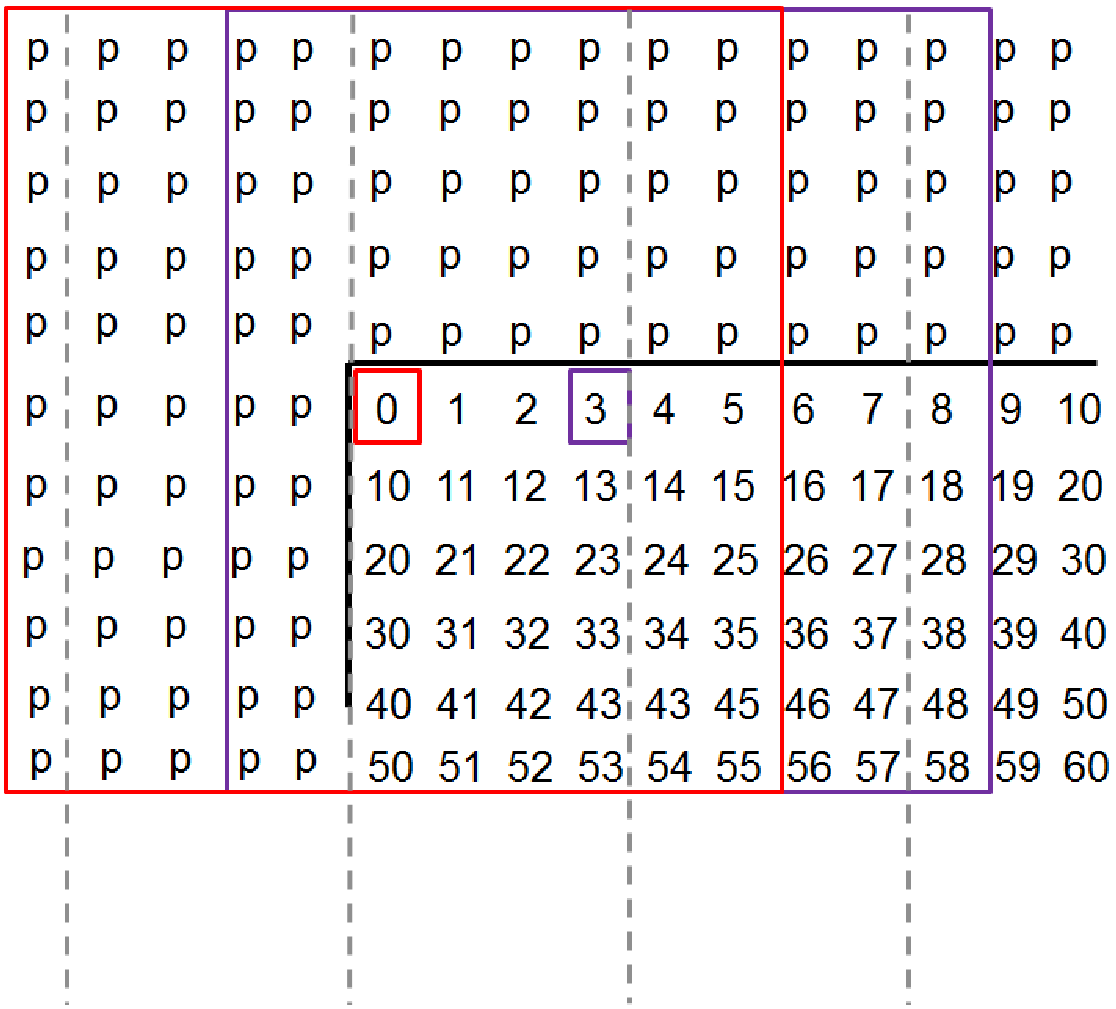
### Implement Edge Detectors

To match the algorithm of the Edge Detector block, this custom edge detector uses a [3x3] kernel size. Compare this KernelIndexer subsystem for the [3x3] edge detection with the [5x5] kernel described above. The algorithm still must access three successive matrices from the output of the Line Buffer block (including padding on either side of the kernel). However, the algorithm saves fewer columns to form a smaller filter kernel.



## Extending to Larger Kernel Sizes

For a [4x1] multipixel stream, the KernelIndexer logic will look similar up to [11x11] kernel size. At that size, the number of padding pixels, `(floor(11/2)) = 5`, will overlap on two [11x4] matrices returned from the Line Buffer. This overlap means the algorithm would need to store five [5x4] matrices from the Line Buffer to form four [11x11] kernels on each cycle.



### Improving Simulation Time

In the default example configuration, the single pixel, multipixel, and custom multipixel subsystems all run in parallel. The simulation speed is limited by the time processing the single-pixel path because it requires more iterations to process the same size of frame. To observe the simulation speed improvement for multipixel streaming, comment out the single-pixel data path.

### HDL Implementation Results

HDL was generated from both the **MultiPixelGaussianEdge** subsystem and the **MultiPixelCustomGaussianEdge** subsystem and put through Place and Route on a Xilinx™ ZC706 board. The **MultiPixelCustomGaussianEdge** subsystem, which does not attempt to optimize coefficients, had the following results -

```
T =

  4x2 table

    Resource      Usage
    _________     _____

    DSP48         108
    Flip Flop     4195
    LUT           4655
    BRAM          12
```

The **MultiPixelGaussianEdge** subsystem, which uses the optimized Image Filter and Edge Detector blocks uses less resources, as shown in the table below. This comparison shows the resource savings achieved because the blocks analyze the filter structure and pre-add repeated coefficients.

```
T =

  4x2 table

    Resource      Usage
    _________     _____

    DSP48         16
    Flip Flop     3959
    LUT           1797
    BRAM          10
```

## See Also

Edge Detector | Frame To Pixels | Image Filter | Pixels To Frame

## More About

- "Streaming Pixel Interface" on page 1-2

## Pixel Control Bus

Vision HDL Toolbox blocks use a nonvirtual bus data type, `pixelcontrol`, for control signals associated with serial pixel data. The bus contains 5 `boolean` signals indicating the validity of a pixel and its location within a frame. You can easily connect the data and control output of one block to the input of another, because Vision HDL Toolbox blocks use this bus for input and output. To convert an image into a pixel stream and a `pixelcontrol` bus, use the Frame to Pixels block.

| Signal | Description | Data Type |
|---|---|---|
| `hStart` | `true` for the first pixel in a horizontal line of a frame | `boolean` |
| `hEnd` | `true` for the last pixel in a horizontal line of a frame | `boolean` |
| `vStart` | `true` for the first pixel in the first (top) line of a frame | `boolean` |
| `vEnd` | `true` for the last pixel in the last (bottom) line of a frame | `boolean` |
| `valid` | `true` for any valid pixel | `boolean` |

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.

**Troubleshooting:** When you generate HDL code from a Simulink model that uses this bus, you may need to declare an instance of `pixelcontrol` bus in the base workspace. If you encounter the error `Cannot resolve variable 'pixelcontrol'` when you generate HDL code in Simulink, use the `pixelcontrolbus` function to create an instance of the bus type. Then try generating HDL code again.

To avoid this issue, the Vision HDL Toolbox model template includes this line in the `InitFcn` callback.

```
evalin('base','pixelcontrolbus')
```

## See Also

Frame To Pixels | Pixels To Frame | `pixelcontrolbus`

### More About

- “Streaming Pixel Interface” on page 1-2

## Pixel Control Structure

Vision HDL Toolbox System objects use a structure data type for control signals associated with serial pixel data. The structure contains five `logical` signals indicating the validity of a pixel and its location within a frame. You can easily pass the data and control output arguments of one Vision HDL Toolbox System object™ as the input arguments to another Vision HDL Toolbox System object, because the objects use this structure for input and output control signal arguments. To convert an image into a pixel stream and control signals, use the `visionhdl.FrameToPixels` System object.

| Signal | Description | Data Type |
|---|---|---|
| `hStart` | `true` for the first pixel in a horizontal line of a frame | `logical` |
| `hEnd` | `true` for the last pixel in a horizontal line of a frame | `logical` |
| `vStart` | `true` for the first pixel in the first (top) line of a frame | `logical` |
| `vEnd` | `true` for the last pixel in the last (bottom) line of a frame | `logical` |
| `valid` | `true` for any valid pixel | `logical` |

## See Also

`pixelcontrolsignals` | `pixelcontrolstruct` | `visionhdl.FrameToPixels` | `visionhdl.PixelsToFrame`

### More About

- “Streaming Pixel Interface” on page 1-2

# Convert Camera Control Signals to pixelcontrol Format

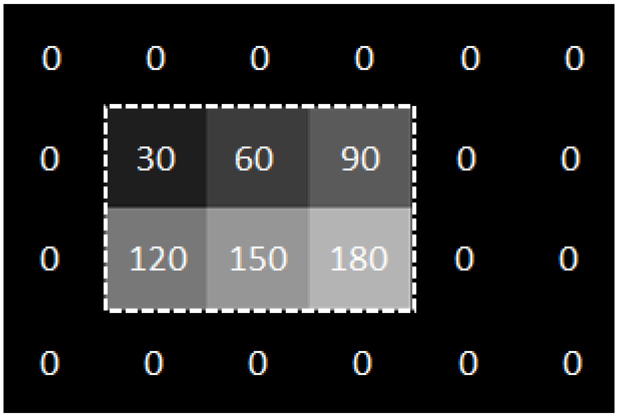
This example converts Camera Link® signals to the `pixelcontrol` structure, inverts the pixels with a Vision HDL Toolbox object, and converts the control signals back to the Camera Link format.

Vision HDL Toolbox™ blocks and objects use a custom streaming video format. If your system operates on streaming video data from a camera, you must convert the camera control signals into this custom format. Alternatively, if you integrate Vision HDL Toolbox algorithms into existing design and verification code that operates in the camera format, you must also convert the output signals from the Vision HDL Toolbox design back to the camera format.

You can generate HDL code from the three functions in this example. To create local copies of all the files in this example, so you can view and edit them, click the Open Script button.

**Create Input Data in Camera Link Format**

The Camera Link format consists of three control signals: F indicates the valid frame, L indicates each valid line, and D indicates each valid pixel. For this example, create input vectors in the Camera Link format to represent a basic padded video frame. The vectors describe this 2-by-3, 8-bit grayscale frame. In the figure, the active image area is in the dashed rectangle, and the inactive pixels surround it. The pixels are labeled with their grayscale values.



```
F = logical([0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0]);
L = logical([0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0]);
```

```
D = logical([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]);
```

### Design Vision HDL Toolbox Algorithm

Create a function to invert the image using Vision HDL Toolbox algorithms. The function contains a System object that supports HDL code generation. This function expects and returns a pixel and associated control signals in Vision HDL Toolbox format.

```
function [pixOut,ctrlOut] = InvertImage(pixIn,ctrlIn)

  persistent invertI;
  if isempty(invertI)
      tabledata = linspace(255,0,256);
      invertI = visionhdl.LookupTable(uint8(tabledata));
  end

  % *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)|.
  [pixOut,ctrlOut] = invertI(pixIn,ctrlIn);
end
```

### Convert Camera Link Control Signals to `pixelcontrol` Format

Write a custom System object to convert Camera Link signals to the Vision HDL Toolbox control signal format. The object converts the control signals, and then calls the `pixelcontrolstruct` function to create the structure expected by the Vision HDL Toolbox System objects. This code snippet shows the logic to convert the signals.

```
  ctrl = pixelcontrolstruct(obj.hStartOutReg,obj.hEndOutReg,...
                     obj.vStartOutReg,obj.vEndOutReg,obj.validOutReg);

  vStart = obj.FReg && ~obj.FPrevReg;
  vEnd = ~F && obj.FReg;
  hStart = obj.LReg && ~obj.LPrevReg;
  hEnd = ~L && obj.LReg;

  obj.vStartOutReg = vStart;
  obj.vEndOutReg = vEnd;
  obj.hStartOutReg = hStart;
  obj.hEndOutReg = hEnd;
  obj.validOutReg = obj.DReg;
```

The object stores the input and output control signal values in registers. vStart goes high for one cycle at the start of F. vEnd goes high for one cycle at the end of F. hStart and hEnd are derived similarly from L. The object returns the current value of ctrl each time you call it.

This processing adds two cycles of delay to the control signals. The object passes through the pixel value after matching delay cycles. For the complete code for the System object, see CAMERALINKtoVHT_Adapter.m.

### Convert pixelcontrol to Camera Link

Write a custom System object to convert Vision HDL Toolbox signals back to the Camera Link format. The object calls the pixelcontrolsignals function to flatten the control structure into its component signals. Then it computes the equivalent Camera Link signals. This code snippet shows the logic to convert the signals.

```
[hStart,hEnd,vStart,vEnd,valid] = pixelcontrolsignals(ctrl);

Fnew = (~obj.FOutReg && vStart) || (obj.FPrevReg && ~obj.vEndReg);
Lnew = (~obj.LOutReg && hStart) || (obj.LPrevReg && ~obj.hEndReg);

obj.FOutReg = Fnew;
obj.LOutReg = Lnew;
obj.DOutReg = valid;
```

The object stores the input and output control signal values in registers. F is high from vStart to vEnd. L is high from hStart to hEnd. The object returns the current values of FOutReg, LOutReg, and DOutReg each time you call it.

This processing adds one cycle of delay to the control signals. The object passes through the pixel value after a matching delay cycle. For the complete code for the System object, see VHTtoCAMERALINKAdapter.m.

### Create Conversion Functions That Support HDL Code Generation

Wrap the converter System objects in functions, similar to InvertImage, so you can generate HDL code for these algorithms.

```
function [ctrl,pixelOut] = CameraLinkToVisionHDL(F,L,D,pixel)
% CameraLink2VisionHDL : converts one cycle of CameraLink control signals
% to Vision HDL format, using a custom System object.
% Introduces two cycles of delay to both ctrl signals and pixel data.
```

```
persistent CL2VHT;
  if isempty(CL2VHT)
      CL2VHT = CAMERALINKtoVHT_Adapter();
  end

  [ctrl,pixelOut] = CL2VHT(F,L,D,pixel);
```

See `CameraLinkToVisionHDL.m`, and `VisionHDLToCameraLink.m`.

### Write a Test Bench

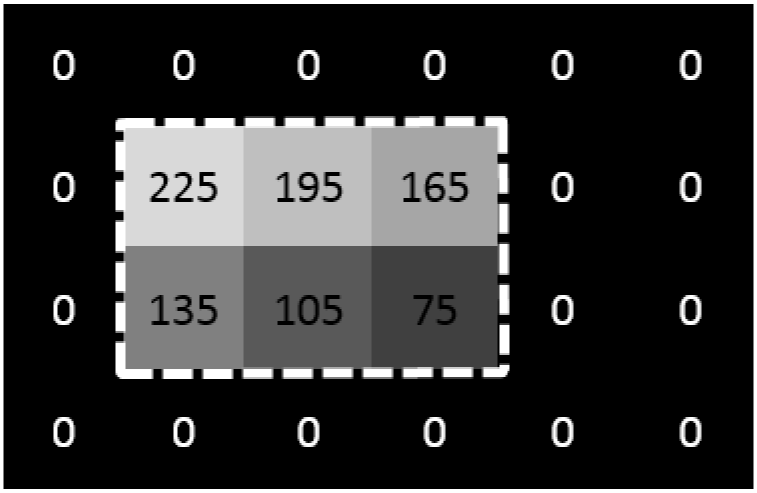
To invert a Camera Link pixel stream using these components, write a test bench script that:

1. Preallocates output vectors to reduce simulation time
2. Converts the Camera Link control signals for each pixel to the Vision HDL Toolbox format
3. Calls the `Invert` function to flip each pixel value
4. Converts the control signals for that pixel back to the Camera Link format

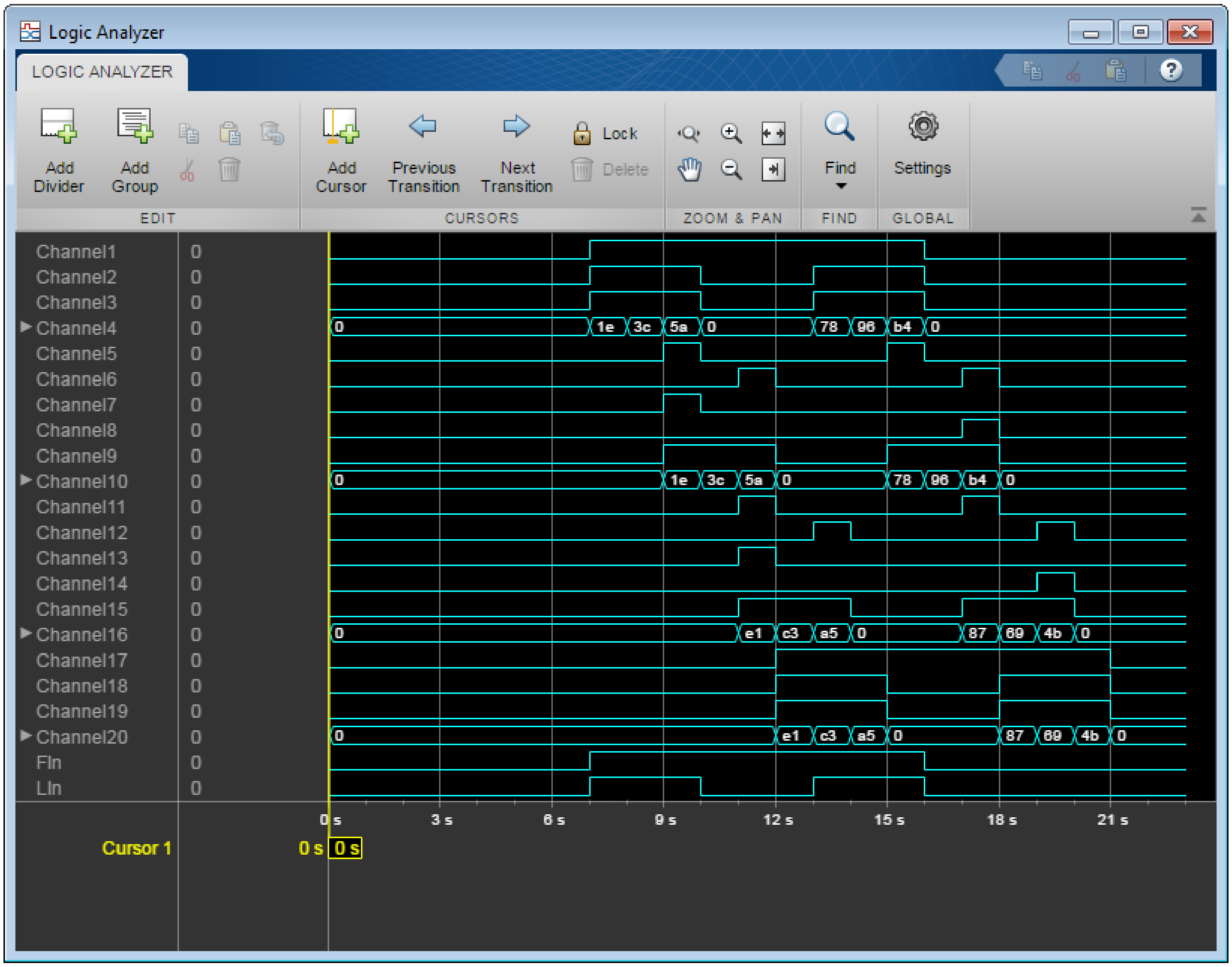
```
[~,numPixelsPerFrame] = size(pixel);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
pixel_d = zeros(numPixelsPerFrame,1,'uint8');
pixOut_d = zeros(numPixelsPerFrame,1,'uint8');
DOut = false(numPixelsPerFrame,1);
FOut = false(numPixelsPerFrame,1);
LOut = false(numPixelsPerFrame,1);
ctrl = repmat(pixelcontrolstruct,numPixelsPerFrame,1);
ctrlOut = repmat(pixelcontrolstruct,numPixelsPerFrame,1);

for p = 1:numPixelsPerFrame
  [pixel_d(p),ctrl(p)] = CameraLinkToVisionHDL(pixel(p),F(p),L(p),D(p));
  [pixOut(p),ctrlOut(p)] = Invert(pixel_d(p),ctrl(p));
  [pixOut_d(p),FOut(p),LOut(p),DOut(p)] = VisionHDLToCameraLink(pixOut(p),ctrlOut(p));
end
```

### View Results

The resulting vectors represent this inverted 2-by-3, 8-bit grayscale frame. In the figure, the active image area is in the dashed rectangle, and the inactive pixels surround it. The pixels are labeled with their grayscale values.



If you have a DSP System Toolbox™ license, you can view the vectors as signals over time using the Logic Analyzer. This waveform shows the `pixelcontrol` and Camera Link control signals, the starting pixel values, and the delayed pixel values after each operation.



## See Also

`pixelcontrolsignals` | `pixelcontrolstruct`

## More About

- "Streaming Pixel Interface" on page 1-2

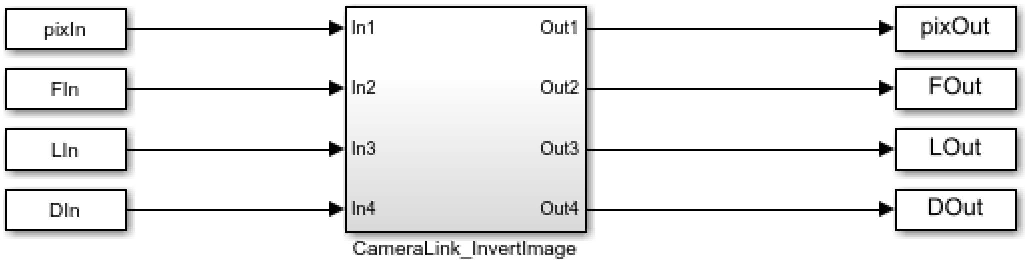
# Integrate Vision HDL Blocks Into Camera Link System

This example shows how to design a Vision HDL Toolbox algorithm for integration into an existing system that uses the Camera Link® signal protocol.

Vision HDL Toolbox™ blocks use a custom streaming video format. If you integrate Vision HDL Toolbox algorithms into existing design and verification code that operates in a different streaming video format, you must convert the control signals at the boundaries. The example uses custom System objects to convert the control signals between the Camera Link format and the Vision HDL Toolbox `pixelcontrol` format. The model imports the System objects to Simulink® by using the MATLAB System block.

### Structure of the Model

This model imports pixel data and control signals in the Camera Link format from the MATLAB® workspace. The `CameraLink_InvertImage` subsystem is designed for integration into existing systems that use Camera Link protocol. The `CameraLink_InvertImage` subsystem converts the control signals from the Camera Link format to the `pixelcontrol` format, modifies the pixel data using the Lookup Table block, and then converts the control signals back to the Camera Link format. The model exports the resulting data and control signals to workspace variables.

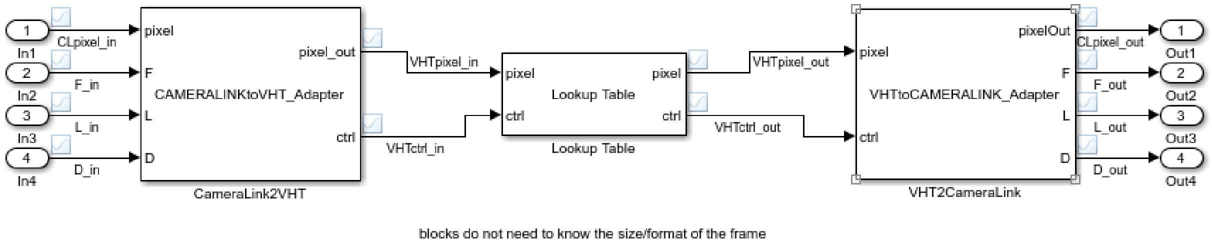


### Structure of the Subsystem

The `CameraLink2VHT` and `VHT2CameraLink` blocks are MATLAB System blocks that point to custom System objects. The objects convert between Camera Link signals and the `pixelcontrol` format used by Vision HDL Toolbox blocks and objects.

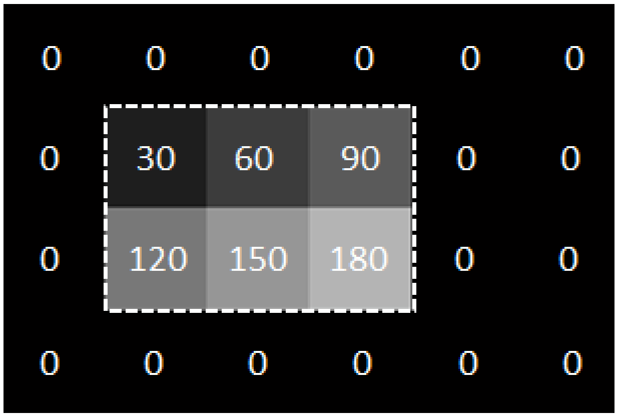
You can put any combination of Vision HDL Toolbox blocks into the middle of the subsystem. This example uses an inversion Lookup Table.

You can generate HDL from this subsystem.



## Import Data in Camera Link Format

Camera Link consists of three control signals: F indicates the valid frame, L indicates each valid line, and D indicates each valid pixel. For this example, the input data and control signals are defined in the `InitFcn` callback. The vectors describe this 2-by-3, 8-bit grayscale frame. In the figure, the active image area is in the dashed rectangle, and the inactive pixels surround it. The pixels are labeled with their grayscale values.



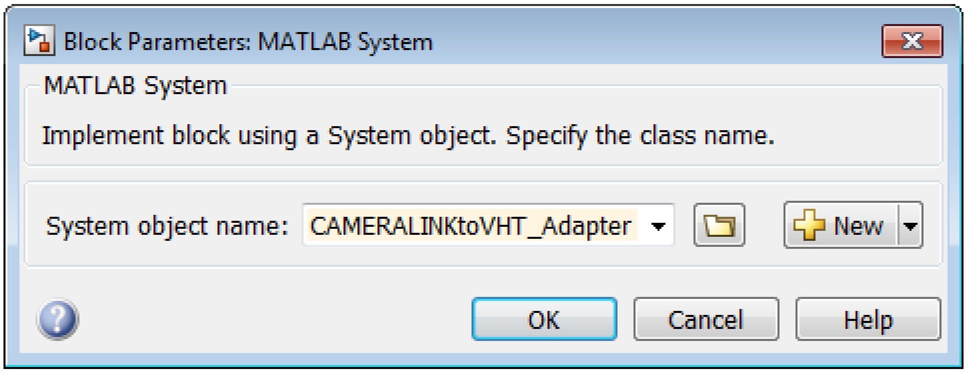
```
FIn = logical([0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0]);
LIn = logical([0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0]);
DIn = logical([0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0]);
pixIn = uint8([0,0,0,0,0,0,0,30,60,90,0,0,0,120,150,180,0,0,0,0,0,0,0,0]);
```

### Convert Camera Link Control Signals to pixelcontrol Format

Write a custom System object to convert Camera Link signals to the Vision HDL Toolbox format. This example uses the object designed in the "Convert Camera Control Signals to pixelcontrol Format" on page 1-23 example.

The object converts the control signals, and then creates a structure that contains the new control signals. When the object is included in a MATLAB System block, the block translates this structure into the bus format expected by Vision HDL Toolbox blocks. For the complete code for the System object, see `CAMERALINKtoVHT_Adapter.m`.

Create a MATLAB System block and point it to the System object.


Block Parameters: MATLAB System

MATLAB System

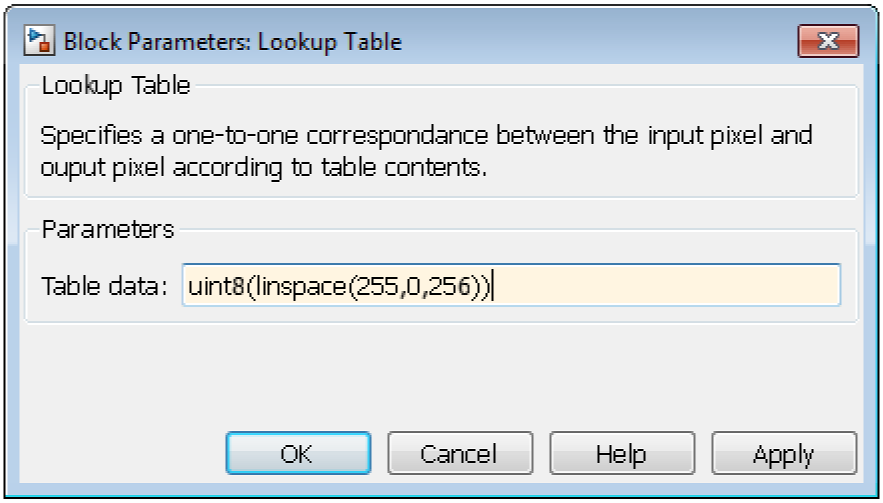
Implement block using a System object. Specify the class name.

System object name: CAMERALINKtoVHT_Adapter

### Design Vision HDL Toolbox Algorithm

Select Vision HDL Toolbox blocks to process the video stream. These blocks accept and return a scalar pixel value and a `pixelcontrol` bus that contains the associated control signals. This standard interface makes it easy to connect blocks from the Vision HDL Toolbox libraries together.

This example uses the Lookup Table block to invert each pixel in the test image. Set the table data to the reverse of the `uint8` grayscale color space.


Block Parameters: Lookup Table

Lookup Table

Specifies a one-to-one correspondance between the input pixel and ouput pixel according to table contents.

Parameters

Table data: uint8(linspace(255,0,256))

#### Convert pixelcontrol to Camera Link

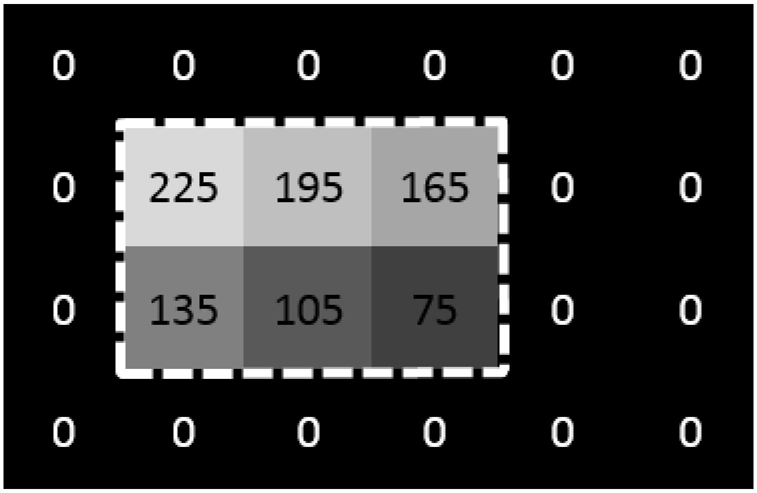
Write a custom System object to convert Vision HDL Toolbox signals back to the Camera Link format. This example uses the object designed in the “Convert Camera Control Signals to pixelcontrol Format” on page 1-23 example.

The object accepts a structure of control signals. When you include the object in a MATLAB System block, the block translates the input `pixelcontrol` bus into this structure. Then it computes the equivalent Camera Link signals. For the complete code for the System object, see `VHTtoCAMERALINKAdapter.m`.

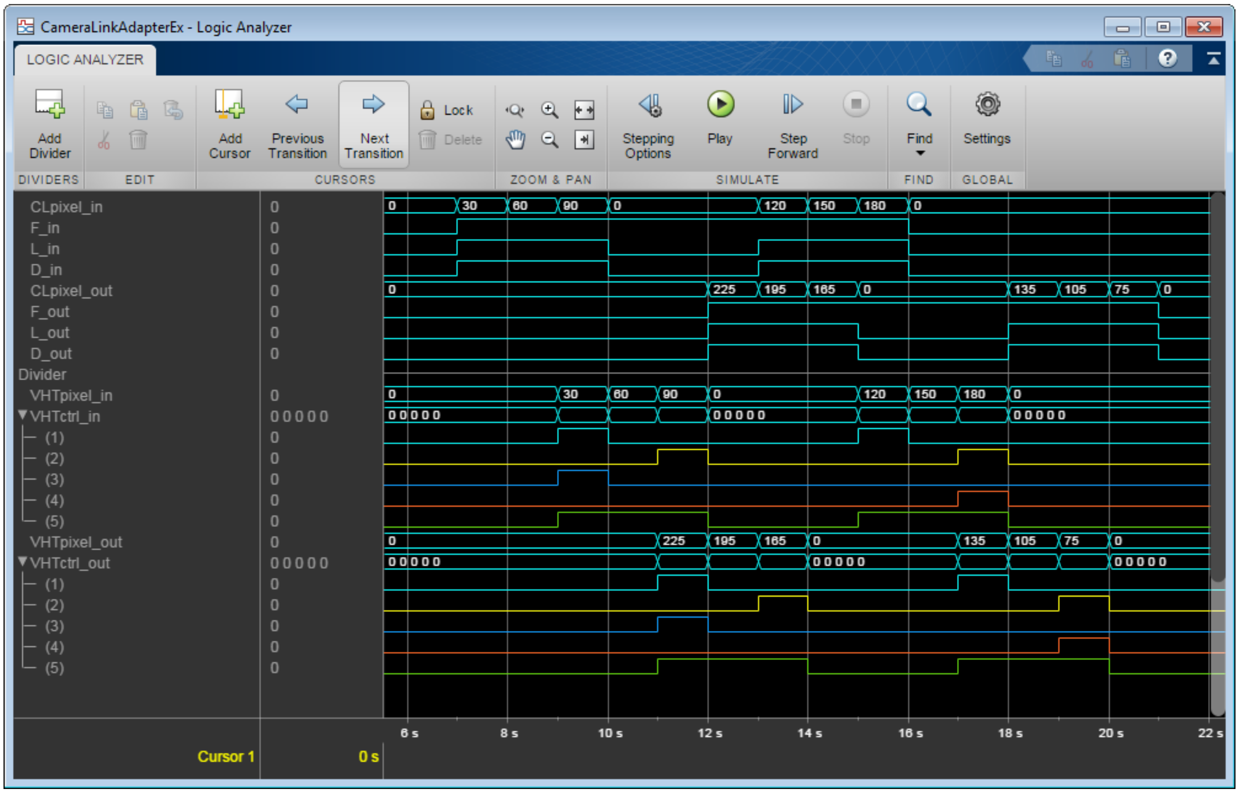
Create a second MATLAB System block and point it to the System object.

#### View Results

Run the simulation. The resulting vectors represent this inverted 2-by-3, 8-bit grayscale frame. In the figure, the active image area is in the dashed rectangle, and the inactive pixels surround it. The pixels are labeled with their grayscale values.



If you have a DSP System Toolbox™ license, you can view the signals over time using the Logic Analyzer. Select all the signals in the `CameraLink_InvertImage` subsystem for streaming, and open the Logic Analyzer. This waveform shows the input and output Camera Link control signals and pixel values at the top, and the input and output of the Lookup Table block in `pixelcontrol` format at the bottom. The `pixelcontrol` busses are expanded to observe the boolean control signals.



For more info on observing waveforms in Simulink, see “Inspect and Measure Transitions Using the Logic Analyzer” (DSP System Toolbox).

## Generate HDL Code for Subsystem

To generate HDL code you must have an HDL Coder™ license.

To generate the HDL code, use the following command.

```
makehdl('CameraLinkAdapterEx/CameraLink_InvertImage')
```

You can now simulate and synthesize these HDL files along with your existing Camera Link system.

## See Also

### More About

- “Streaming Pixel Interface” on page 1-2

# Algorithms

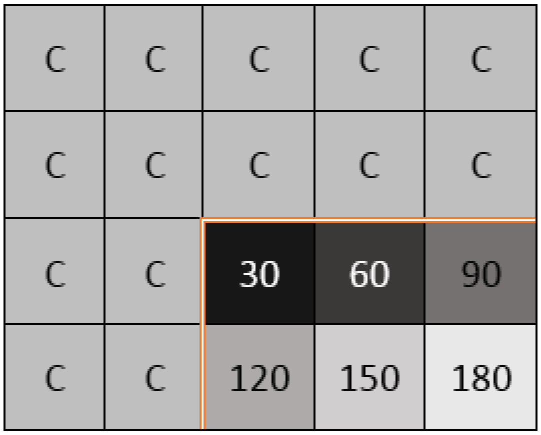
## Edge Padding

To perform a kernel-based operation such as filtering on a pixel at the edge of a frame, Vision HDL Toolbox algorithms pad the edges of the frame with extra pixels. These padding pixels are used for internal calculation only. The output frame has the same dimensions as the input frame. The padding operation assigns a pattern of pixel values to the inactive pixels around a frame. Vision HDL Toolbox algorithms provide padding by constant value, replication, or symmetry. Some blocks and System objects enable you to select from these padding methods.

The diagrams show the top-left corner of a frame, with padding added to accommodate a 5 × 5 filter kernel. When computing the filtered value for the top-left active pixel, the algorithm requires two rows and two columns of padding. The edge of the active image is indicated by the double line.

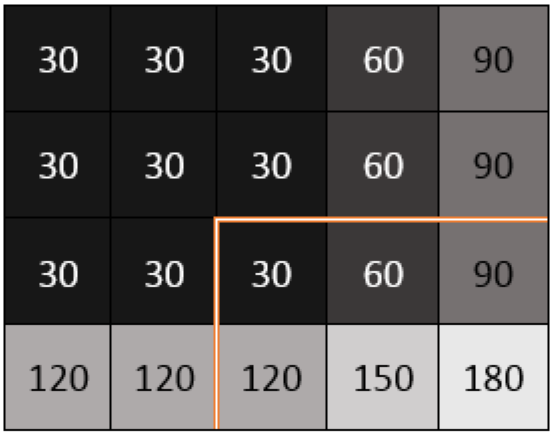
- `Constant` — Each added pixel is assigned the same value. On some blocks and System objects you can specify the constant value. The value `0`, representing black, is a reserved value in some video standards. It is common to choose a small number, such as `16`, as a near-black padding value.

  In the diagram, *C* represents the constant value assigned to the inactive pixels around the active frame.



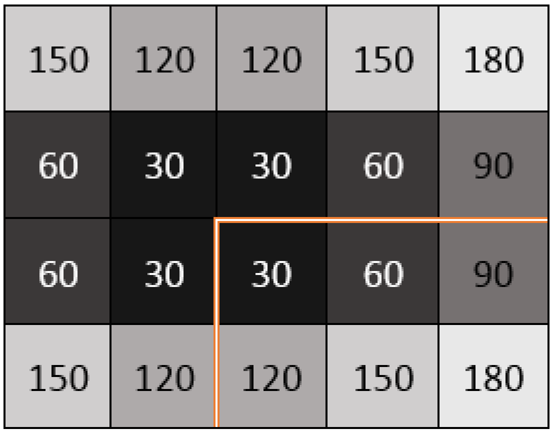
- `Replicate` — The pixel values at the edge of the active frame are repeated to make rows and columns of padding pixels.

  The diagram shows the pattern of replicated values assigned to the inactive pixels around the active frame.



- `Symmetric` — The padding pixels are added such that they mirror the edge of the image.

  The diagram shows the pattern of symmetric values assigned to the inactive pixels around the active frame. The pixel values are symmetric about the edge of the image in both dimensions.



| | | | | |
|---|---|---|---|---|
| 150 | 120 | 120 | 150 | 180 |
| 60 | 30 | 30 | 60 | 90 |
| 60 | 30 | 30 | 60 | 90 |
| 150 | 120 | 120 | 150 | 180 |

Padding requires minimum horizontal and vertical blanking periods. This interval gives the algorithm time to add and store the extra pixels. The blanking period, or inactive pixel region, must be at least *kernel size* pixels in each dimension.

## See Also

Image Filter | `visionhdl.ImageFilter`

### More About

- “Streaming Pixel Interface” on page 1-2

3

# Code Generation and Deployment

## Accelerate a MATLAB Design With MATLAB Coder

Vision HDL Toolbox designs in MATLAB must call one or more System objects for every pixel. This serial processing is efficient in hardware, but is slow in simulation. One way to accelerate simulations of these objects is to simulate using generated C code rather than the MATLAB interpreted language.

Code generation accelerates simulation by locking down the sizes and data types of variables inside the function. This process removes the overhead of the interpreted language checking for size and data type in every line of code. You can compile a video processing algorithm and test bench into MEX functions, and use the resulting MEX file to speed up the simulation.

To generate C code, you must have a MATLAB Coder™ license.

See “Accelerate a Pixel-Streaming Design Using MATLAB Coder”.

## HDL Code Generation from Vision HDL Toolbox

**In this section...**

### What Is HDL Code Generation?

You can use MATLAB and Simulink for rapid prototyping of hardware designs. Vision HDL Toolbox blocks and System objects, when used with HDL Coder™, provide support for HDL code generation. HDL Coder tools generate target-independent synthesizable Verilog® and VHDL® code for FPGA programming or ASIC prototyping and design.

### HDL Code Generation Support in Vision HDL Toolbox

Most blocks and objects in Vision HDL Toolbox support HDL code generation.

The following blocks and objects are for simulation only and are not supported for HDL code generation :

- Frame To Pixels (`visionhdl.FrameToPixels`)
- Pixels To Frame (`visionhdl.PixelsToFrame`)
- FIL Frame To Pixels (`visionhdl.FILFrameToPixels`)
- FIL Pixels To Frame (`visionhdl.FILPixelsToFrame`)
- Measure Timing (`visionhdl.MeasureTiming`)

### Streaming Pixel Interface in HDL

The streaming pixel bus and structure data type used by Vision HDL Toolbox blocks and System objects is flattened into separate signals in HDL.

In VHDL, the interface is declared as:

```
PORT( clk            :   IN    std_logic;
      reset          :   IN    std_logic;
      enb            :   IN    std_logic;
```

```
        in0             :   IN    std_logic_vector(7 DOWNTO 0); -- uint8
        in1_hStart      :   IN    std_logic;
        in1_hEnd        :   IN    std_logic;
        in1_vStart      :   IN    std_logic;
        in1_vEnd        :   IN    std_logic;
        in1_valid       :   IN    std_logic;
        out0            :   OUT   std_logic_vector(7 DOWNTO 0); -- uint8
        out1_hStart     :   OUT   std_logic;
        out1_hEnd       :   OUT   std_logic;
        out1_vStart     :   OUT   std_logic;
        out1_vEnd       :   OUT   std_logic;
        out1_valid      :   OUT   std_logic
        );
```

In Verilog, the interface is declared as:

```
  input   clk;
  input   reset;
  input   enb;
  input   [7:0] in0;  // uint8
  input   in1_hStart;
  input   in1_hEnd;
  input   in1_vStart;
  input   in1_vEnd;
  input   in1_valid;
  output  [7:0] out0;  // uint8
  output  out1_hStart;
  output  out1_hEnd;
  output  out1_vStart;
  output  out1_vEnd;
  output  out1_valid;
```

## Blocks and System Objects Supporting HDL Code Generation

Most blocks and objects in Vision HDL Toolbox are supported for HDL code generation. For exceptions, see “HDL Code Generation Support in Vision HDL Toolbox” on page 3-3. This page helps you find blocks and objects supported for HDL code generation in other MathWorks® products.

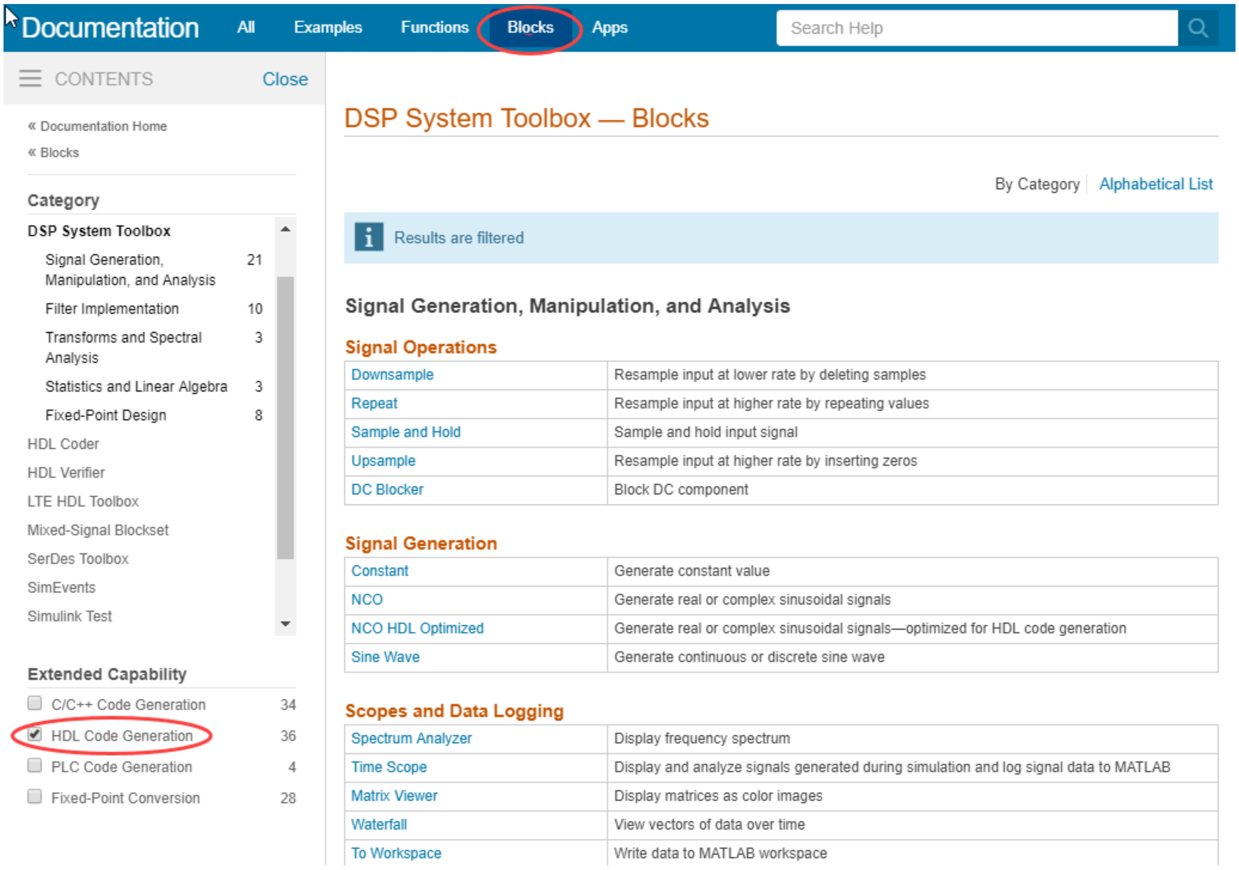
### Blocks

In the Simulink library browser, you can find libraries of blocks supported for HDL code generation in the **HDL Coder**, **Communications Toolbox HDL Support**, and **DSP System Toolbox HDL Support** block libraries.

To create a library of HDL-supported blocks from all your installed products, enter `hdllib` at the MATLAB command line. This command requires an HDL Coder license.

Refer to the "Extended Capabilities > HDL Code Generation" section of each block page for block implementations, properties, and restrictions for HDL code generation.

You can also view blocks that are supported for HDL code generation in documentation by filtering the block reference list. Click **Blocks** in the blue bar at the top of the Help window, then select the **HDL code generation** check box at the bottom of the left column. The blocks are listed in their respective products. You can use the table of contents in the left column to navigate between products and categories.



DSP System Toolbox — Blocks

By Category | Alphabetical List

Results are filtered

| Category | |
|---|---|
| **DSP System Toolbox** | |
| Signal Generation, Manipulation, and Analysis | 21 |
| Filter Implementation | 10 |
| Transforms and Spectral Analysis | 3 |
| Statistics and Linear Algebra | 3 |
| Fixed-Point Design | 8 |
| HDL Coder | |
| HDL Verifier | |
| LTE HDL Toolbox | |
| Mixed-Signal Blockset | |
| SerDes Toolbox | |
| SimEvents | |
| Simulink Test | |

| Extended Capability | |
|---|---|
| ☐ C/C++ Code Generation | 34 |
| ☑ HDL Code Generation | 36 |
| ☐ PLC Code Generation | 4 |
| ☐ Fixed-Point Conversion | 28 |

Signal Generation, Manipulation, and Analysis

Signal Operations

| | |
|---|---|
| Downsample | Resample input at lower rate by deleting samples |
| Repeat | Resample input at higher rate by repeating values |
| Sample and Hold | Sample and hold input signal |
| Upsample | Resample input at higher rate by inserting zeros |
| DC Blocker | Block DC component |

Signal Generation

| | |
|---|---|
| Constant | Generate constant value |
| NCO | Generate real or complex sinusoidal signals |
| NCO HDL Optimized | Generate real or complex sinusoidal signals—optimized for HDL code generation |
| Sine Wave | Generate continuous or discrete sine wave |

Scopes and Data Logging

| | |
|---|---|
| Spectrum Analyzer | Display frequency spectrum |
| Time Scope | Display and analyze signals generated during simulation and log signal data to MATLAB |
| Matrix Viewer | Display matrices as color images |
| Waterfall | View vectors of data over time |
| To Workspace | Write data to MATLAB workspace |

## System Objects

To find System objects supported for HDL code generation, see Predefined System Objects (HDL Coder).

# Generate HDL Code From Simulink

## Introduction

This page shows you how to generate HDL code from the design described in “Design Video Processing Algorithms for HDL in Simulink”. You can generate HDL code from the HDL Algorithm subsystem in the model.

To generate HDL code, you must have an HDL Coder license.

## Prepare Model

Run the `visionhdlsetup` function to configure the model for HDL code generation. If you started your design using the Vision HDL Toolbox Simulink model template, your model is already configured for HDL code generation.

## Generate HDL Code

Right-click the HDL Algorithm block, and select **HDL Code** > **Generate HDL from subsystem** to generate HDL using the default settings. The output log of this operation is shown in the MATLAB Command Window, along with the location of the generated files.

To change code generation options, use the **HDL Code Generation** section of Simulink Configuration Parameters. For guidance through the HDL code generation process, or to select a target device or synthesis tool, right-click on the HDL Algorithm block, and select **HDL Code** > **HDL Workflow Advisor**.

Alternatively, from the MATLAB Command Window, you can call:

```
makehdl([modelname '/HDL Algorithm'])
```

## Generate HDL Test Bench

You can select options to generate a test bench in Simulink Configuration Parameters or in **HDL Workflow Advisor**.

Alternatively, to generate an HDL test bench from the command line, call:

```
makehdltb([modelname '/HDL Algorithm'])
```

## See Also

**Functions**
`makehdl` | `makehdltb`

### Related Examples

- “HDL Code Generation and FPGA Synthesis Using Simulink HDL Workflow Advisor” (HDL Coder)
- “Choose a Test Bench for Generated HDL Code” (HDL Coder)

## Generate HDL Code From MATLAB

This example show you how to generate HDL code from the design in “Design a Hardware-Targeted Image Filter in MATLAB”.

To generate HDL code, you must have an HDL Coder license.

### Create an HDL Coder Project

Copy the relevant files to a temporary folder.

```
functionName = 'HDLTargetedDesign';
tbName = 'VisionHDLMATLABTutorialExample';
vhtExampleDir = fullfile(matlabroot,'examples','visionhdl');
workDir = [tempdir 'vht_matlabhdl_ex'];

cd(tempdir)
[~, ~, ~] = rmdir(workDir, 's');
mkdir(workDir)
cd(workDir)

copyfile(fullfile(vhtExampleDir, [functionName,'.m*']), workDir)
copyfile(fullfile(vhtExampleDir, [tbName,'.m*']), workDir)
```

Open the HDL Coder app and create a new project.

```
coder -hdlcoder -new vht_matlabhdl_ex
```

In the **HDL Code Generation** pane, add the function file `HDLTargetedDesign.m` and the test bench file `VisionHDLMATLABTutorialExample.m` to the project.

Click next to the signal names under **MATLAB Function** to define the data types for the input and output signals of the function. The control signals are `logical` scalars. The pixel data type is `uint8`. The pixel input is a scalar.

### Generate HDL Code

1. Click **Workflow Advisor** to open the advisor.
2. Click **HDL Code Generation** to view the HDL code generation options.
3. On the **Target** tab, set **Language** to `Verilog` or `VHDL`.

4 Also on the **Target** tab, select **Generate HDL** and **Generate HDL test bench**.

5 On the **Coding Style** tab, select **Include MATLAB source code as comments** and **Generate report** to generate a code generation report with comments and traceability links.

6 Click **Run** to generate the HDL design and the test bench with reports.

Examine the log window and click the links to view the generated code and the reports.

## See Also

### Related Examples

- “Getting Started with MATLAB to HDL Workflow” (HDL Coder)
- “Generate HDL Code from MATLAB Code Using the Command Line Interface” (HDL Coder)
- “HDL Code Generation for System Objects” (HDL Coder)
- “Pixel-Streaming Design in MATLAB”

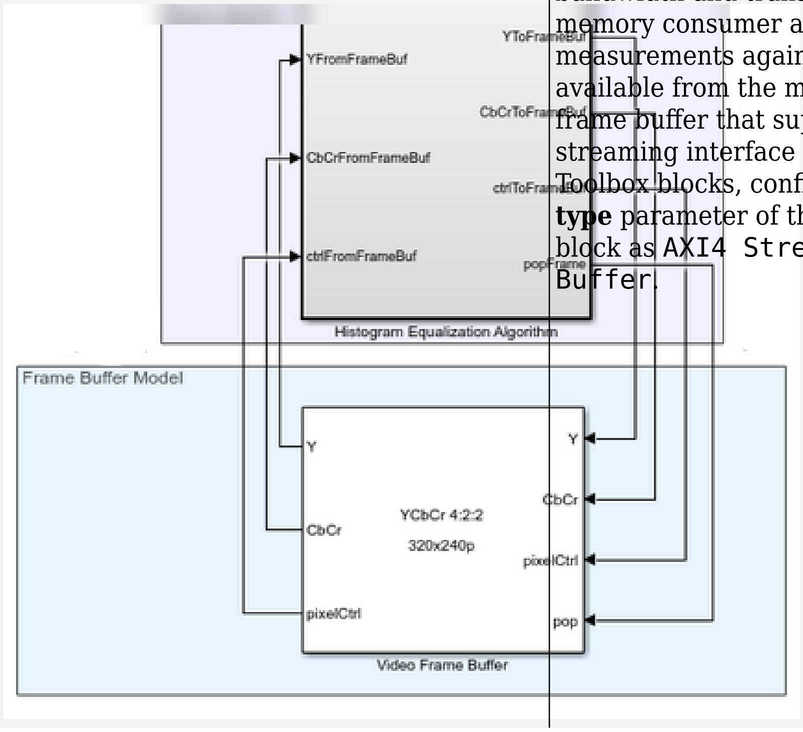
## Modeling External Memory

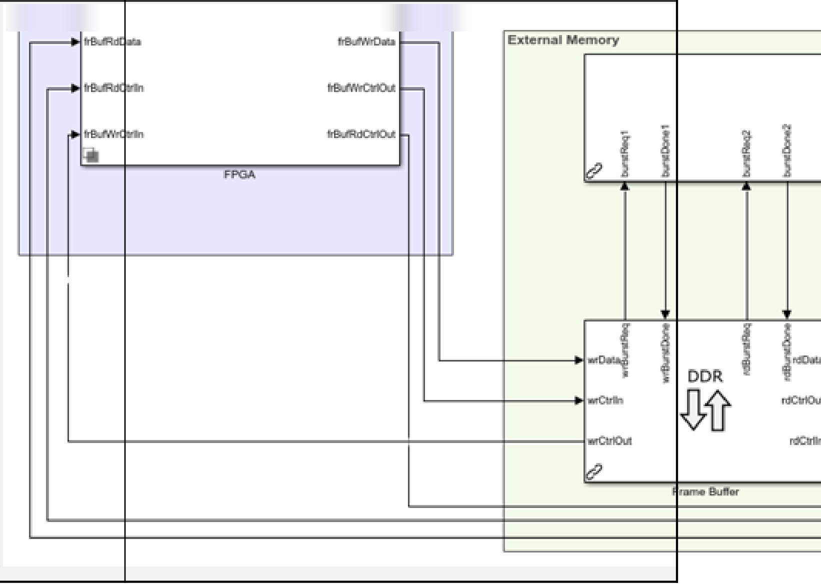
You can model external memory using features from Computer Vision Toolbox™ Support Package for Xilinx® Zynq®-Based Hardware or SoC Blockset™. Both products provide models for a frame buffer or a random access interface. They both also map your subsystem ports to physical AXI memory interfaces when you generate HDL code and target a prototype board.

Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware provides a simple model of the memory interface. It does not model the timing of the interface. This level of modeling assists with targeting a memory interface on hardware, but behavior can differ between the simulation and the hardware. For more information, see “Model External Memory Interfaces” (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

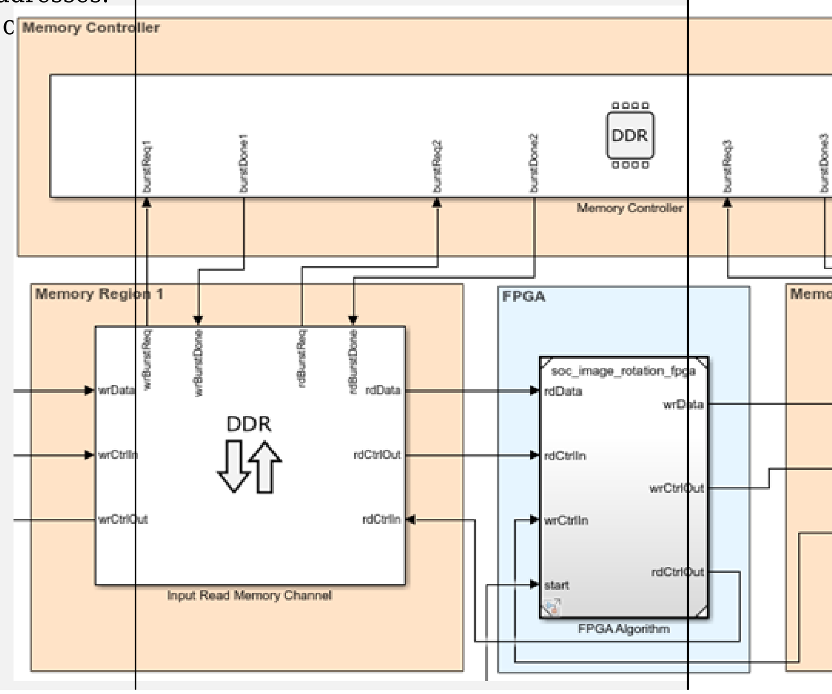
SoC Blockset provides library blocks to model a memory controller and multiple memory channels. This model calculates and visualizes memory bandwidth, burst counts, and transaction latencies in simulation. You can also model memory accesses from a processor as part of hardware-software co-design. Use the **SoC Builder** app to generate code for FPGA and processor designs and load and run the design on a board. You can also deploy an AXI memory interconnect monitor on your FPGA, which can return memory transaction information for debugging and visualization in Simulink. This level of modeling helps you verify throughput and latency requirements and enables modeling of multiple memory consumers, including processor memory access. For more information, see “Memory Transactions” (SoC Blockset).

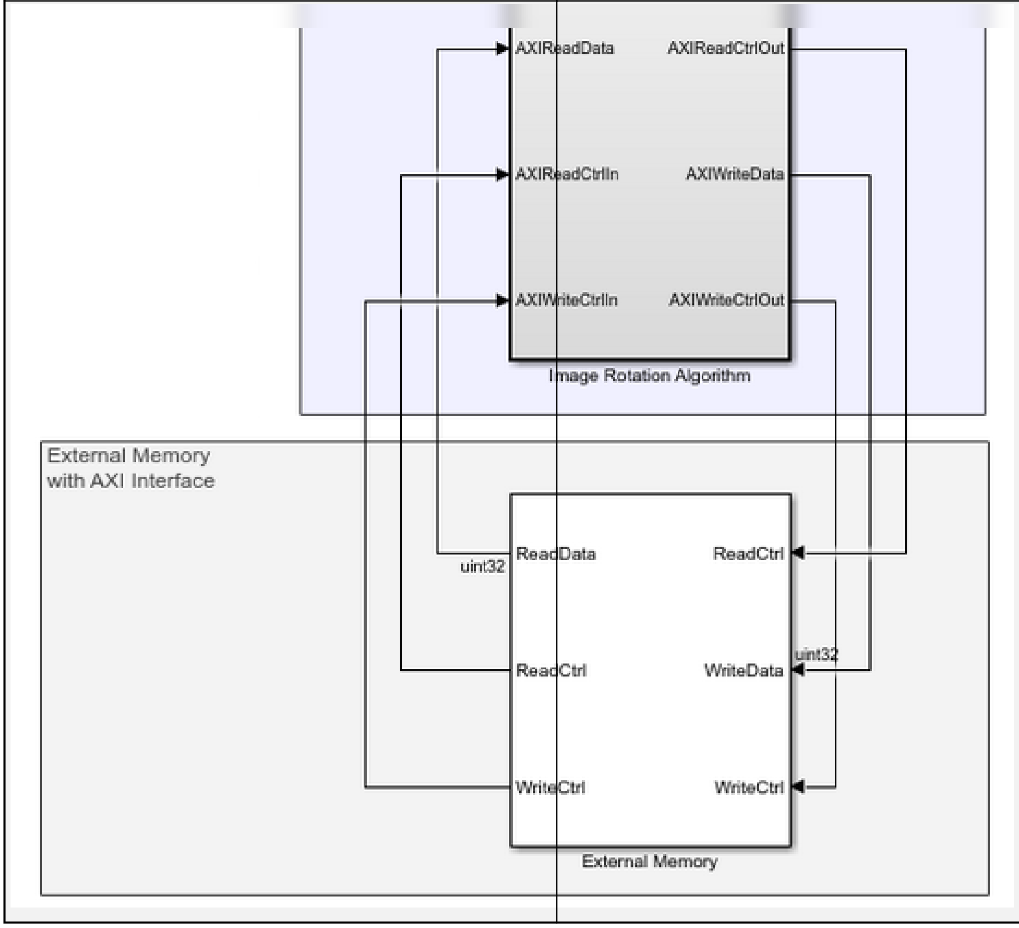
## Frame Buffer

| Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware | SoC Blockset |
|---|---|
| This figure shows part of the "Histogram Equalization with Zynq-Based Hardware" (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware) example. The Video Frame Buffer block accepts and returns the pixel streaming interface used by Vision HDL Toolbox blocks. It reads and returns an entire frame when you set the **pop** signal to 1. To use this block in your designs, copy it from the example model.  | This figure shows part of the "Histogram Equalization Using Video Frame Buffer" (SoC Blockset) example. The example shows how to use the Memory Channel and Memory Controller library blocks to model a frame buffer and additional memory consumers. You can use this model to confirm that the memory interface meets the throughput and latency requirements of your design. You can measure the bandwidth and transaction latency for each memory consumer and check the measurements against the total bandwidth available from the memory. To model a frame buffer that supports the pixel streaming interface used by Vision HDL Toolbox blocks, configure the **Channel type** parameter of the Memory Channel block as `AXI4 Stream Video Frame Buffer`. |

| Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware | SoC Blockset |
|---|---|
|  |   frBufRdData<br>frBufRdCtrlIn<br>frBufWrCtrlIn<br>frBufWrData<br>frBufWrCtrlOut<br>frBufRdCtrlOut<br>FPGA<br>External Memory<br>burstReq1<br>burstDone1<br>burstReq2<br>burstDone2<br>wrData<br>wrCtrlIn<br>wrCtrlOut<br>wrBurstReq<br>wrBurstDone<br>rdBurstReq<br>rdBurstDone<br>rdData<br>rdCtrlOut<br>rdCtrlIn<br>DDR<br>Frame Buffer |

## Random Access

| Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware | SoC Blockset |
|---|---|
| This figure shows part of the “Image Rotation with Zynq-Based Hardware” (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware) example. The External Memory block reads and writes to any address in the memory. In this case, rather than connecting the pixel stream to the memory interface, your custom FPGA logic must generate read and write transactions with specific addresses. To use this block in your designs, c from the example model. | This figure shows part of the “Random Access of External Memory” (SoC Blockset) example. This design uses a Memory Controller and two Memory Channel blocks to implement a random-access interface. In this case, rather than connecting the pixel stream to the memory interface, your custom FPGA logic must generate read and write transactions with specific addresses.   Memory Controller<br>DDR<br>burstReq1<br>burstDone1<br>burstReq2<br>burstDone2<br>burstReq3<br>burstDone3<br>Memory Controller<br>Memory Region 1<br>wrBurstReq<br>wrBurstDone<br>rdBurstReq<br>rdBurstDone<br>wrData<br>wrCtrlIn<br>wrCtrlOut<br>rdData<br>rdCtrlOut<br>rdCtrlIn<br>DDR<br>Input Read Memory Channel<br>FPGA<br>soc_image_rotation_fpga<br>rdData<br>rdCtrlIn<br>wrCtrlIn<br>start<br>wrData<br>wrCtrlOut<br>rdCtrlOut<br>FPGA Algorithm |

| Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware | SoC Blockset |
|---|---|
|   AXIReadData<br>AXIReadCtrlOut<br>AXIReadCtrlIn<br>AXIWriteData<br>AXIWriteCtrlIn<br>AXIWriteCtrlOut<br>Image Rotation Algorithm<br>External Memory with AXI Interface<br>uint32<br>ReadData<br>ReadCtrl<br>ReadCtrl<br>uint32<br>WriteData<br>WriteCtrl<br>WriteCtrl<br>External Memory | |

## See Also

"Model External Memory Interfaces" (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware) | "Memory Transactions" (SoC Blockset)

## HDL Cosimulation

HDL cosimulation links an HDL simulator with MATLAB or Simulink. This communication link enables integrated verification of the HDL implementation against the design. To perform this integration, you need an HDL Verifier™ license. HDL Verifier cosimulation tools enable you to:

- Use MATLAB or Simulink to create test signals and software test benches for HDL code
- Use MATLAB or Simulink to provide a behavioral model for an HDL simulation
- Use MATLAB analysis and visualization capabilities for real-time insight into an HDL implementation
- Use Simulink to translate legacy HDL descriptions into system-level views

## See Also

### More About

- "HDL Cosimulation" (HDL Verifier)

## FPGA-in-the-Loop

FPGA-in-the-loop (FIL) enables you to run a Simulink or MATLAB simulation that is synchronized with an HDL design running on an FPGA board. This link between the simulator and the board enables you to verify HDL implementations directly against Simulink or MATLAB algorithms. You can apply real-world data and test scenarios from these algorithms to the HDL design that is running on the FPGA.

In Simulink, you can use the FIL Frame To Pixels and FIL Pixels To Frame blocks to accelerate communication between Simulink and the FPGA board. In MATLAB, you can modify the generated code to speed up communication with the FPGA board.

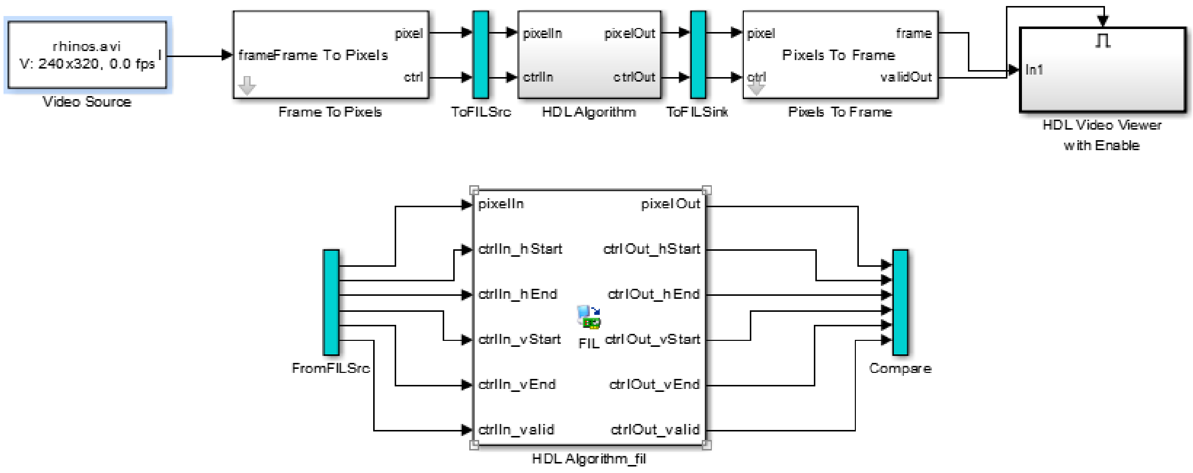
### FPGA-in-the-Loop Simulation with Vision HDL Toolbox Blocks

This example shows how to modify the generated FPGA-in-the-loop (FIL) model for more efficient simulation of the Vision HDL Toolbox™ streaming video protocol.

#### Autogenerated FIL Model

When you generate a programming file for a FIL target in Simulink, the HDL Workflow Advisor creates a model to compare the FIL simulation with your Simulink design. For details of how to generate FIL artifacts for a Simulink model, see “FIL Simulation with HDL Workflow Advisor for Simulink” (HDL Verifier).

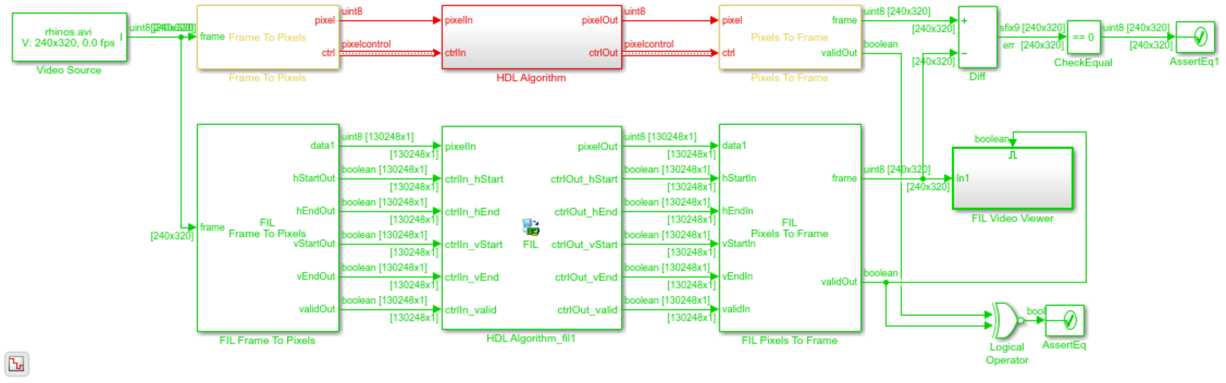
For Vision HDL Toolbox designs, the FIL block in the generated model replicates the pixel-streaming interface and sends one pixel at a time to the FPGA. The model shown was generated from the example model in “Design Video Processing Algorithms for HDL in Simulink”.



The top part of the model replicates your Simulink design. The generated FIL block at the bottom communicates with the FPGA. ToFILSrc subsystem copies the pixel-stream input of the HDL Algorithm block to the FromFILSrc subsystem. The ToFILSink subsystem copies the pixel-stream output of the HDL Algorithm block into the Compare subsystem, where it is compared with the output of the HDL Algorithm_fil block. For image and video processing, this setup is slow because the model sends only a single pixel, and its associated control signals, in each packet to and from the FPGA board.

### Modified FIL Model for Pixel Streaming

To improve the communication bandwidth with the FPGA board, you can use the generated FIL block with vector input rather than streaming. This example includes a model, FILSimulinkWithVHTExample.slx, created by modifying the generated FIL model. The modified model uses the FIL Frame To Pixels and FIL Pixels To Frame blocks to send one frame at a time to the generated FIL block. You cannot run this model as is. You must generate your own FIL block and bitstream file that use your board and connection settings.



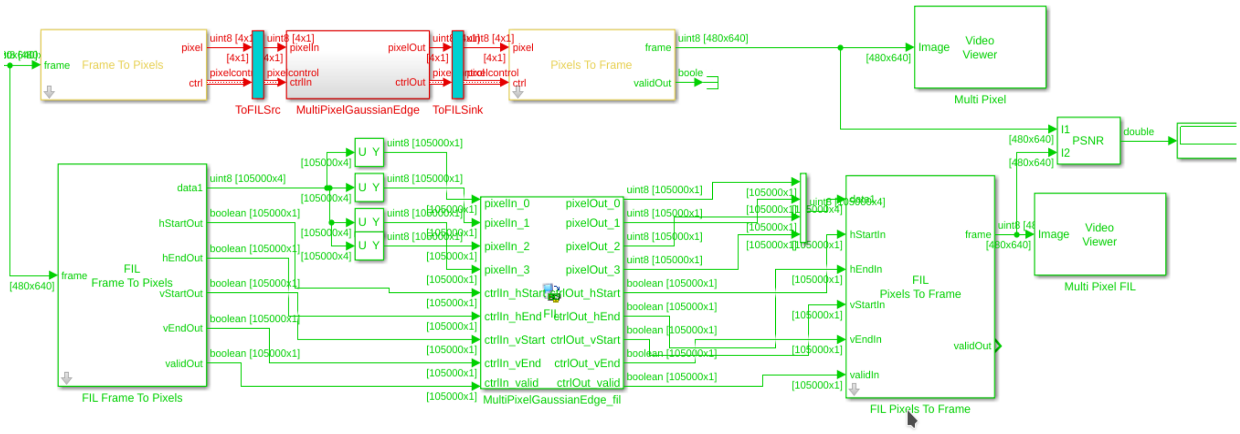
To convert from the generated model to the modified model:

1. Remove the ToFILSrc, FromFILSrc, ToFILSink, and Compare subsystems, and create a branch at the frame input of the Frame To Pixels block.
2. Insert the FIL Frame To Pixels block before the HDL Algorithm_fil block. Insert the FIL Pixels To Frame block after the HDL Algorithm_fil block.
3. Branch the frame output of the Pixels To Frame block for comparison. You can compare the entire frame at once with a Diff block. Compare the `validOut` signals using an XOR block.
4. In the FIL Frame To Pixels and FIL Pixels To Frame blocks, set the Video format parameter to match the video format of the Frame To Pixels and Pixels To Frame blocks.
5. Set the **Vector size** in the FIL Frame To Pixels and FIL Pixels To Frame blocks to `Frame` or `Line`. The size of the FIL Frame To Pixels vector output must match the size of the FIL Pixels To Frame vector input. The vector size of the FIL block interfaces does not modify the generated HDL code. It affects only the packet size of the communication between the simulator and the FPGA board.

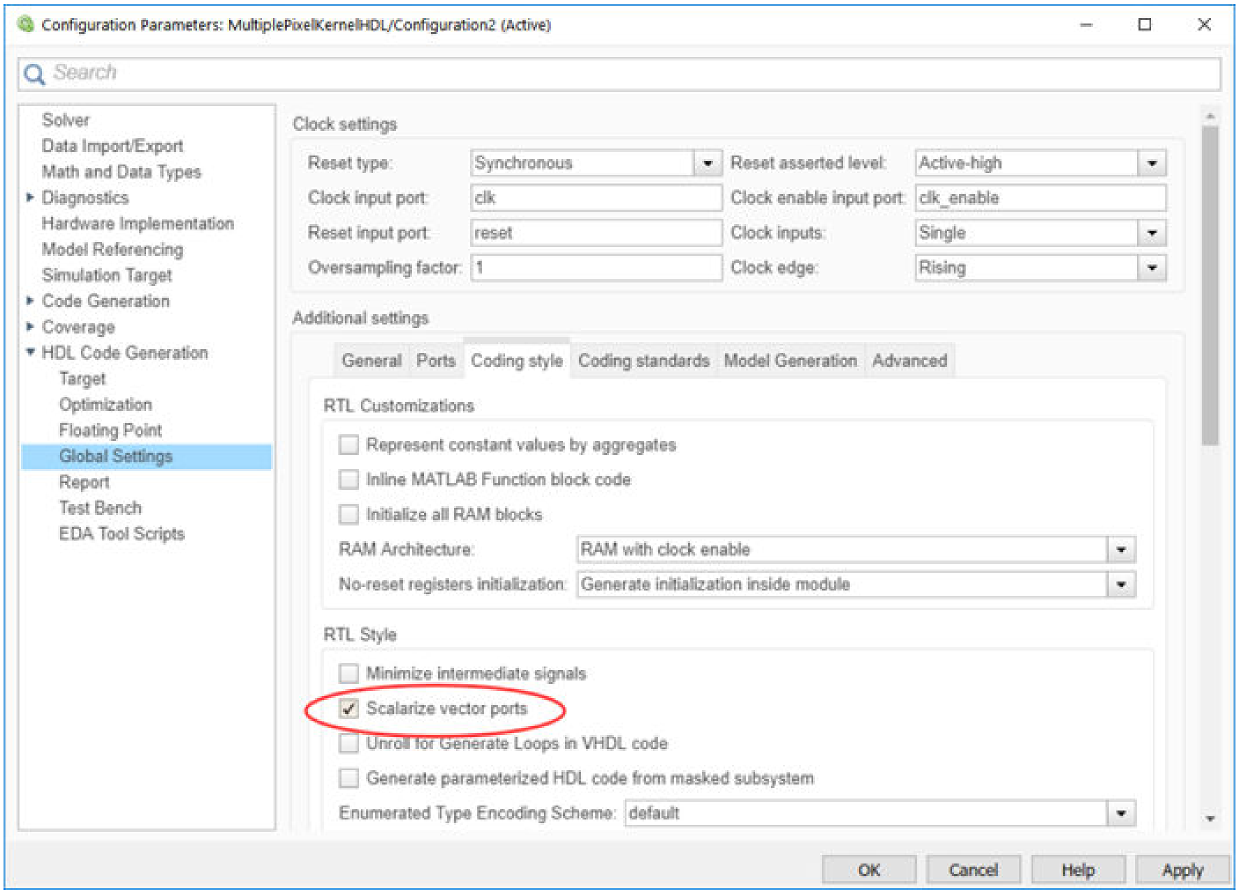
The modified model sends an entire frame to the FPGA board in each packet, significantly improving the efficiency of the communication link.

## FPGA-in-the-Loop Simulation with Multipixel Streaming

When using FPGA-in-the-Loop with a multipixel streaming design, you must flatten the pixel ports to vectors for input and output of the FIL block. Use Selector blocks to separate the input pixel streams into *NumPixels* vectors, and use a Vector Concatenate block to recombine the output vectors.



Also, in **Configuration Parameters > HDL Code Generation > Global Settings > Coding style**, select the **Scalarize vector ports** checkbox.


Configuration Parameters: MultiplePixelKernelHDL/Configuration2 (Active)

Clock settings

Reset type: Synchronous
Reset asserted level: Active-high
Clock input port: clk
Clock enable input port: clk_enable
Reset input port: reset
Clock inputs: Single
Oversampling factor: 1
Clock edge: Rising

Additional settings

RTL Customizations

- [ ] Represent constant values by aggregates
- [ ] Inline MATLAB Function block code
- [ ] Initialize all RAM blocks

RAM Architecture: RAM with clock enable
No-reset registers initialization: Generate initialization inside module

RTL Style

- [ ] Minimize intermediate signals
- [✓] Scalarize vector ports
- [ ] Unroll for Generate Loops in VHDL code
- [ ] Generate parameterized HDL code from masked subsystem

Enumerated Type Encoding Scheme: default

## FPGA-in-the-Loop Simulation with Vision HDL Toolbox System Objects

This example shows how to modify the generated FPGA-in-the-loop (FIL) script for more efficient simulation of the Vision HDL Toolbox™ streaming video protocol. For details of how to generate FIL artifacts for a MATLAB® System object™, see “FIL Simulation with HDL Workflow Advisor for MATLAB” (HDL Verifier).

### Autogenerated FIL Function

When you generate a programming file for a FIL target in MATLAB, the HDL Workflow Advisor creates a test bench to compare the FIL simulation with your MATLAB design. For Vision HDL Toolbox designs, the *DUTname*`_fil` function in the test bench replicates the pixel-streaming interface and sends one pixel at a time to the FPGA. *DUTname* is the name of the function that you generated HDL code from.

This code snippet is from the generated test bench *TBname*`_fil.m`, generated from the example script in “Pixel-Streaming Design in MATLAB”. The code calls the generated *DUTname*`_fil` function once for each pixel in a frame.

```
for p = 1:numPixPerFrm
    [pixOutVec( p ),ctrlOutVec( p )] = PixelStreamingDesignHDLDesign_fil( pixInVec( p 
end
```

The generated *DUTname*`_fil` function calls your HDL-targeted function. It also calls the *DUTname*`_sysobj_fil` function, which contains a System object that connects to the FPGA. *DUTname*`_fil` compares the output of the two functions to verify that the FPGA implementation matches the original MATLAB results. This snippet is from the file *DUTname*`_fil.m`.

```
% Call the original MATLAB function to get reference signal
[ref_pixOut,tmp_ctrlOut] = PixelStreamingDesignHDLDesign(pixIn,ctrlIn);

  ...

% Run FPGA-in-the-Loop
[pixOut,ctrlOut_hStart,ctrlOut_hEnd,ctrlOut_vStart,ctrlOut_vEnd,ctrlOut_valid] ...
  = PixelStreamingDesignHDLDesign_sysobj_fil(pixIn,ctrlIn_hStart,ctrlIn_hEnd,ctrlIn_vS

  ...

% Verify the FPGA-in-the-Loop output
hdlverifier.assert(pixOut,ref_pixOut,'pixOut');
```

For image and video processing, this setup is slow because the function sends only one pixel, and its associated control signals, in each packet to and from the FPGA board.

### Modified FIL Test Bench for Pixel Streaming

To improve the communication bandwidth with the FPGA board, you can modify the autogenerated test bench, *TBname*`_fil.m`. The modified test bench calls the FIL System object directly, with one frame at a time. These snippets are from the

PixelStreamingDesignHDLTestBench_fil_frame.m script, modified from FIL artifacts generated from the example script in “Pixel-Streaming Design in MATLAB”. You cannot run this script as is. You must generate your own FIL System object, function, and bitstream file that use your board and connection settings. Then, either modify your version of the generated test bench, or modify this script to use your generated FIL object.

Declare an instance of the generated FIL System object.

```
fil = class_PixelStreamingDesignHDLDesign_sysobj;
```

Comment out the loop over the pixels in the frame.

```
%         for p = 1:numPixPerFrm
%            [pixOutVec( p ),ctrlOutVec( p )] = PixelStreamingDesignHDLDesign_fil( pixI
%         end
```

Replace the commented loop with the code below. Call the `step` method of the `fil` object with vectors containing the whole frame of data pixels and control signals. Pass each control signal to the object separately, as a vector of logical values. Then, recombine the control signal vectors into a vector of structures.

```
[pixOutVec,hStartOut,hEndOut,vStartOut,vEndOut,validOut] = ...
    step(fil,pixInVec,[ctrlInVec.hStart]',[ctrlInVec.hEnd]',[ctrlInVec.vStart]',[ctrlIn
ctrlOutVec = arrayfun(@(hStart,hEnd,vStart,vEnd,valid) ...
    struct('hStart',hStart,'hEnd',hEnd,'vStart',vStart,'vEnd',vEnd,'valid',valid),...
    hStartOut,hEndOut,vStartOut,vEndOut,validOut);
```

These code changes remove the pixel-by-pixel verification of the FIL results against the MATLAB results. Optionally, you can add a pixel loop to call the reference function, and a frame-by-frame comparison of the results. However, calling the original function for a reference slows down the simulation.

```
for p = 1:numPixPerFrm
     [ref_pixOutVec(p),ref_ctrlOutVec(p)] = PixelStreamingDesignHDLDesign(pixInVec(p),c
end
```

After the call to the `fil` object, compare the output vectors.

```
hdlverifier.assert(pixOutVec',ref_pixOutVec,'pixOut')
hdlverifier.assert([ctrlOutVec.hStart],[ref_ctrlOutVec.hStart],'hStart')
hdlverifier.assert([ctrlOutVec.hEnd],[ref_ctrlOutVec.hEnd],'hEnd')
hdlverifier.assert([ctrlOutVec.vStart],[ref_ctrlOutVec.vStart],'vStart')
hdlverifier.assert([ctrlOutVec.vEnd],[ref_ctrlOutVec.vEnd],'vEnc')
hdlverifier.assert([ctrlOutVec.valid],[ref_ctrlOutVec.valid],'valid')
```

This modified test bench sends an entire frame to the FPGA board in each packet, significantly improving the efficiency of the communication link.

## See Also

**Blocks**
FIL Frame To Pixels | FIL Pixels To Frame | Image Filter

**Objects**
`visionhdl.ImageFilter`

### More About

- “FPGA Verification” (HDL Verifier)

# Prototype Vision Algorithms on Zynq-Based Hardware

You can use the Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware to prototype your vision algorithms on Zynq-based hardware that is connected to real input and output video devices. Use the support package to:

- Capture input or output video from the board and import it into Simulink for algorithm development and verification.
- Generate and deploy vision IP cores to the FPGA on the board. (requires HDL Coder)
- Generate and deploy C code to the ARM® processor on the board. You can route the video data from the FPGA into the ARM® processor to develop video processing algorithms targeted to the ARM processor. (requires Embedded Coder®)
- View the output of your algorithm on an HDMI device.

## Video Capture

Using this support package, you can capture live video from your Zynq device and import it into Simulink. The video source can be an HDMI video input to the board, an on-chip test pattern generator included with the reference design, or the output of your custom algorithm on the board. You can select the color space and resolution of the input frames. The capture resolution must match that of your input camera.

Once you have video frames in Simulink, you can:

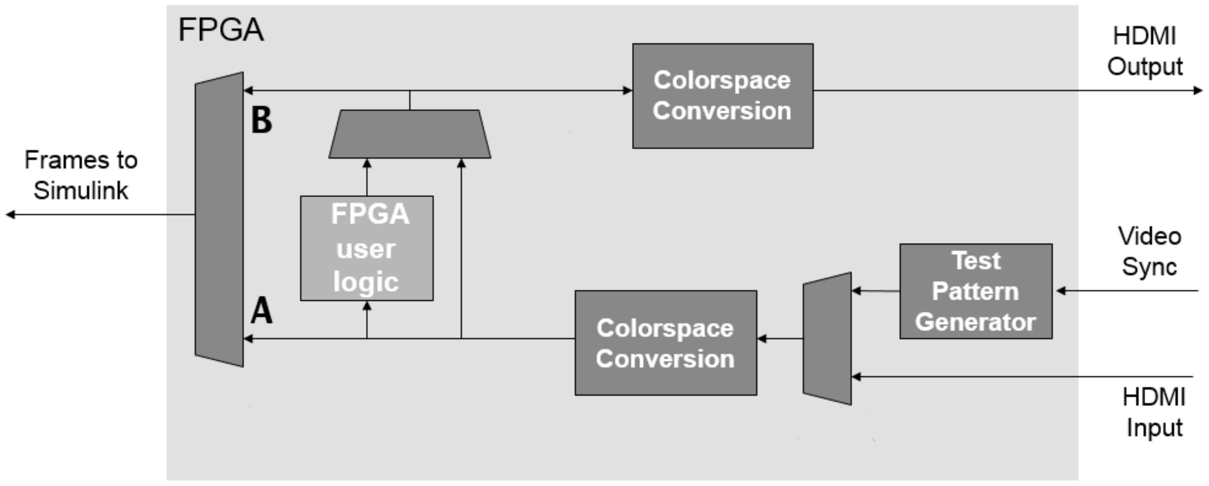
- Design frame-based video processing algorithms that operate on the live data input. Use blocks from the Computer Vision Toolbox libraries to quickly develop frame-based, floating-point algorithms.
- Use the Frame To Pixels block from Vision HDL Toolbox to convert the input to a pixel stream. Design and verify pixel-streaming algorithms using other blocks from the Vision HDL Toolbox libraries.

## Reference Design

The Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware provides a reference design for prototyping video algorithms on the Zynq boards.

When you generate an HDL IP core for your pixel-streaming design using HDL Workflow Advisor, the core is included in this reference design as the FPGA user logic section. Points **A** and **B** in the diagram show the options for capturing video into Simulink.

The FPGA user logic can also contain an optional interface to external frame buffer memory, which is not shown in the diagram.



---

**Note** The reference design on the Zynq device requires the same video resolution and color format for the entire data path. The resolution you select must match that of your camera input. The design you target to the user logic section of the FPGA must not modify the frame size or color space of the video stream.

The reference design does not support multipixel streaming.

---

## Deployment and Generated Models

By running all or part of your pixel-streaming design on the hardware, you speed up simulation of your video processing system and can verify its behavior on real hardware. To generate HDL code and deploy your design to the FPGA, you must have HDL Coder and the HDL Coder Support Package for Xilinx Zynq Platform, as well as Xilinx Vivado® and the Xilinx SDK.

After FPGA targeting, you can capture the live output frames from the FPGA user logic back to Simulink for further processing and analysis. You can also view the output on an HDMI output connected to your board. Using the generated hardware interface model, you can control the video capture options and read and write AXI-Lite ports on the FPGA user logic from Simulink during simulation.

The FPGA targeting step also generates a software interface model. This model supports software targeting to the Zynq hardware, including external mode, processor-in-the-loop, and full deployment. It provides data path control, and an interface to any AXI-Lite ports you defined on your FPGA targeted subsystem. From this model, you can generate ARM code that drives or responds to the AXI-Lite ports on the FPGA user logic. You can then deploy the code on the board to run along with the FPGA user logic. To deploy software to the ARM processor, you must have Embedded Coder and the Embedded Coder Support Package for Xilinx Zynq Platform.

## See Also

### More About

- “Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware”

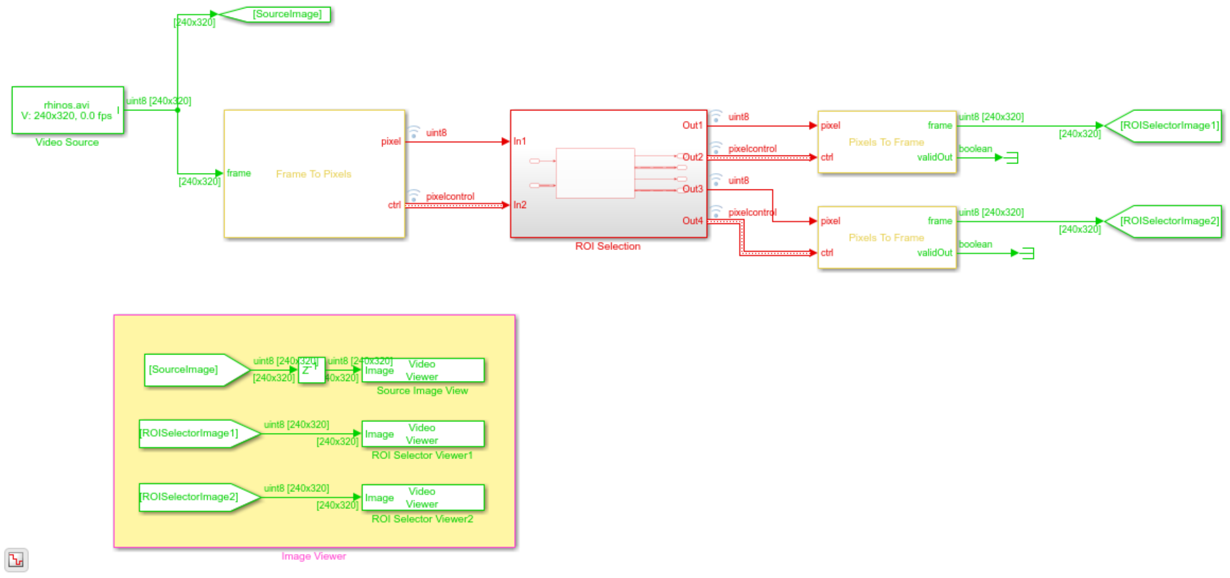
# Examples

## Select Region of Interest

This example shows how to select a region of active frame from a video stream by using the ROI Selector block from the Vision HDL Toolbox™.

There are numerous applications where the input video is divided into several zones. In medical imaging, the boundaries of a tumor may be defined on an image or in a volume for the purpose of measuring its size. In geographical information systems (GIS), an ROI can be taken as a polygonal selection from a 2-D map.

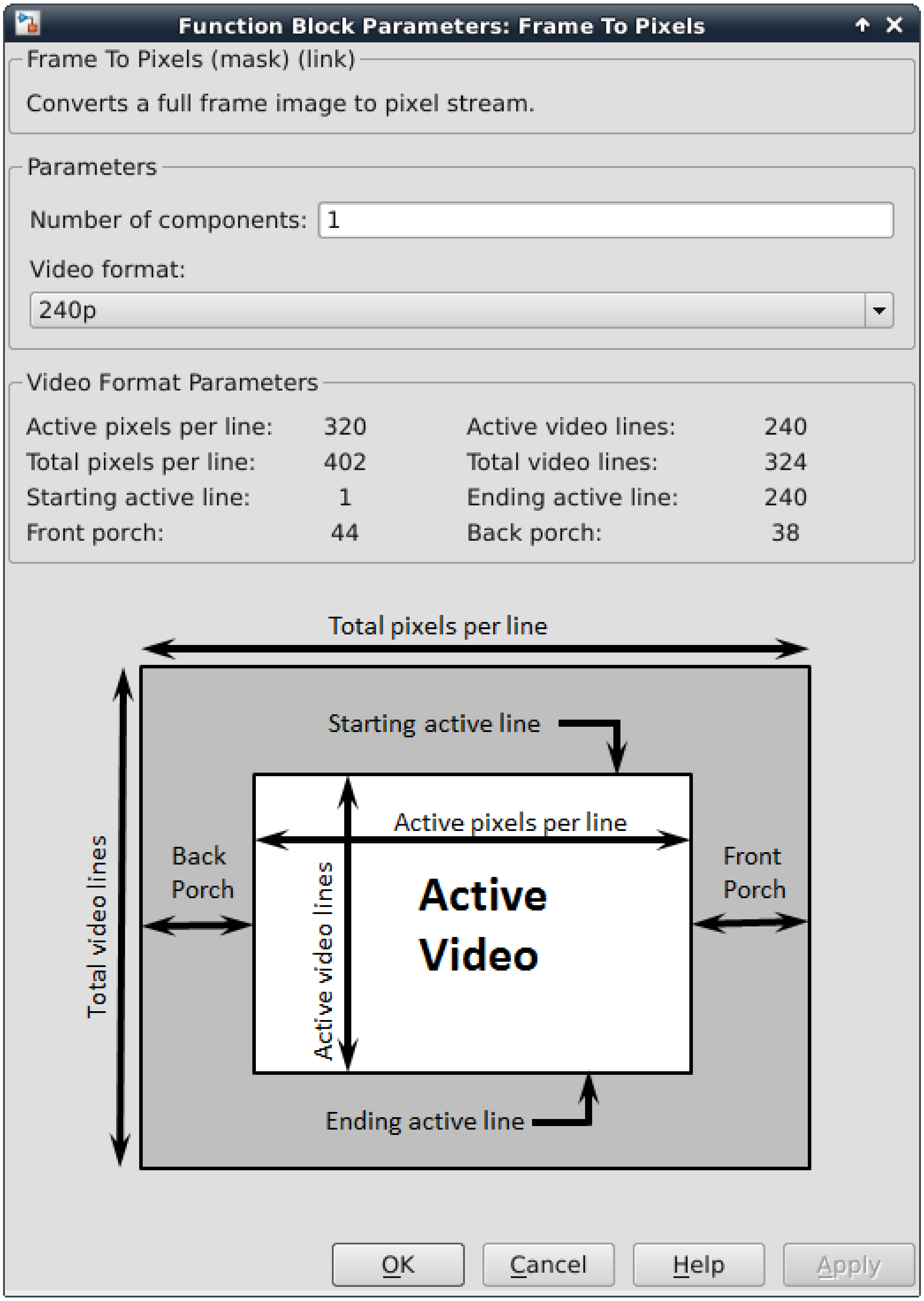
### Example Model



The example model includes a Video Source block that contains a 240p video sample. Each pixel is a scalar `uint8` value that represents intensity. The green and red lines represent full-frame processing and pixel-stream processing, respectively.

### Serialize the Image

Use Frame To Pixels block to convert a full-frame image into pixel stream. To simulate the effect of horizontal and vertical blanking periods found in real life hardware video systems, the active image is augmented with non-image data. For more information on the streaming pixel protocol, see “Streaming Pixel Interface” on page 1-2. The Frame To Pixels block is configured as shown:



Function Block Parameters: Frame To Pixels

Frame To Pixels (mask) (link)

Converts a full frame image to pixel stream.

Parameters

Number of components: 1

Video format: 240p

Video Format Parameters

| | | | |
|---|---|---|---|
| Active pixels per line: | 320 | Active video lines: | 240 |
| Total pixels per line: | 402 | Total video lines: | 324 |
| Starting active line: | 1 | Ending active line: | 240 |
| Front porch: | 44 | Back porch: | 38 |

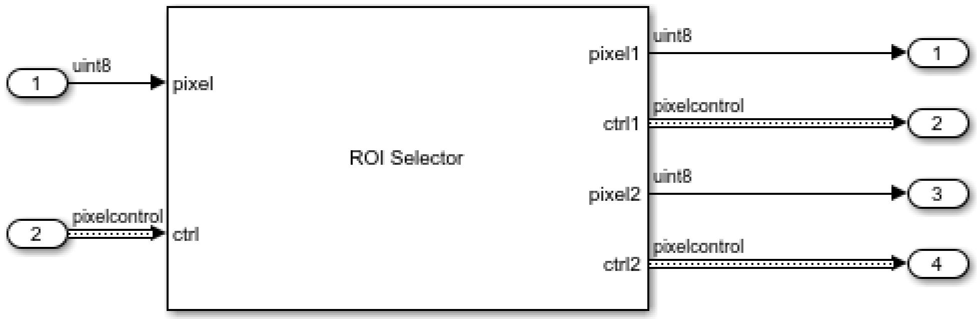
The **Number of components** parameter is set to 1 for grayscale image input, and the **Video format** parameter is `240p` to match the video source.

In this example, the Active Video region corresponds to the 240x320 matrix of the source image. Six other parameters, namely, **Total pixels per line**, **Total video lines**, **Starting active line**, **Ending active line**, **Front porch**, and **Back porch**, specify how many non-image data will be augmented on the four sides of the Active Video. For more information, see the Frame To Pixels block reference page.

Note that the sample time of the Video Source block is determined by the product of **Total pixels per line** and **Total video lines**.

### Select Regions of Interest

The ROI Selection subsystem contains only an ROI Selector block.



Use the ROI Selector block to select regions of interest. You can use the **Regions** parameter to experiment with different region sizes and examine their effect on the output frames. In this model, the **Regions** parameter is set to `[100 100 50 50;220 170 100 70]` which represents two regions, each specified by `[hPos vPos hSize vSize]`. The first region is 50-by-50 pixels and located 100 pixels to the right and 100 pixels down from the top-left corner of the active frame. The second region is 100 pixels wide and 70 pixels tall, and is located in the bottom-right corner of the active frame.

The ROI Selector block accepts a pixel stream and a bus that contains five control signals from the Frame To Pixels block. It returns each region as a pixel stream that uses the same protocol, by manipulating the control signals. Each region is selected by setting the `valid` signal in the output `pixelcontrol` bus to `false` for any pixels not included in the requested region.

## Display Regions of Interest

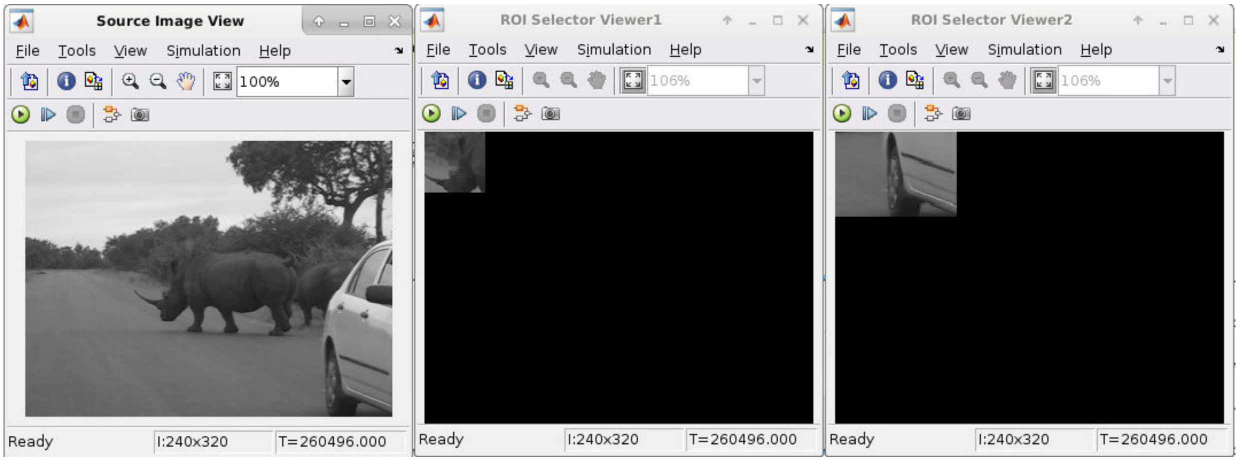
Use the Pixels To Frame block to convert the pixel stream back into a full frame. Since the output of the Pixels To Frame block is a 2-D matrix of a full image, there is no further need for the `pixelcontrol` bus.

The **Number of components** and **Video format** parameters of both Frame To Pixels and Pixels To Frame are set to `1` and `240p`, respectively, to match the format of the video source. The size of each active frame is smaller than 240p after the ROI selection. The Pixels to Frame block returns a 240-by-320 matrix with the active portion of the frame in the top-left corner.

Run the model to display the results. The model displays the output video streams by using three Video Viewer blocks.

- Source Image View -- The input video from the Video Source block
- ROI Selector Viewer1 -- The 50-by-50 pixel region
- ROI Selector Viewer2 -- The 100-by-70 pixel region

One frame of the source video and the two regions are shown from left to right.



The Unit Delay block on the top level of the model is to time-align the matrices for a fair comparison.

### Generate HDL Code

To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command:

```
makehdl('ROISelectionHDL/ROI Selection')
```

To generate a test bench, use the following command:

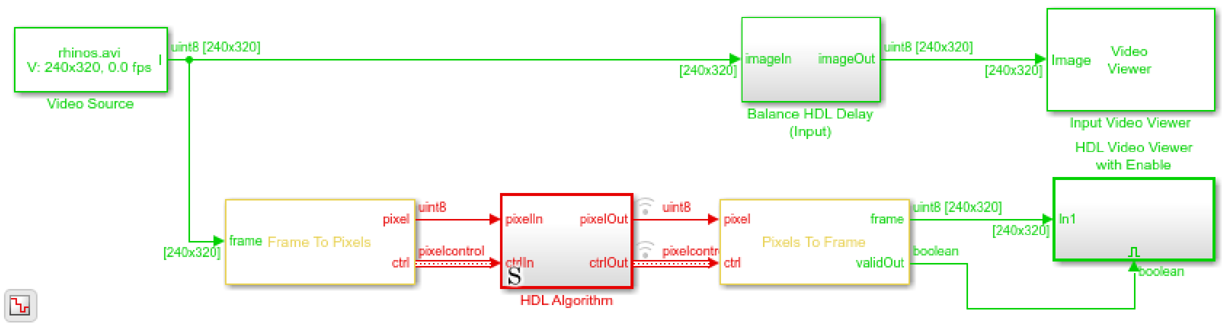
```
makehdltb('ROISelectionHDL/ROI Selection')
```

## See Also

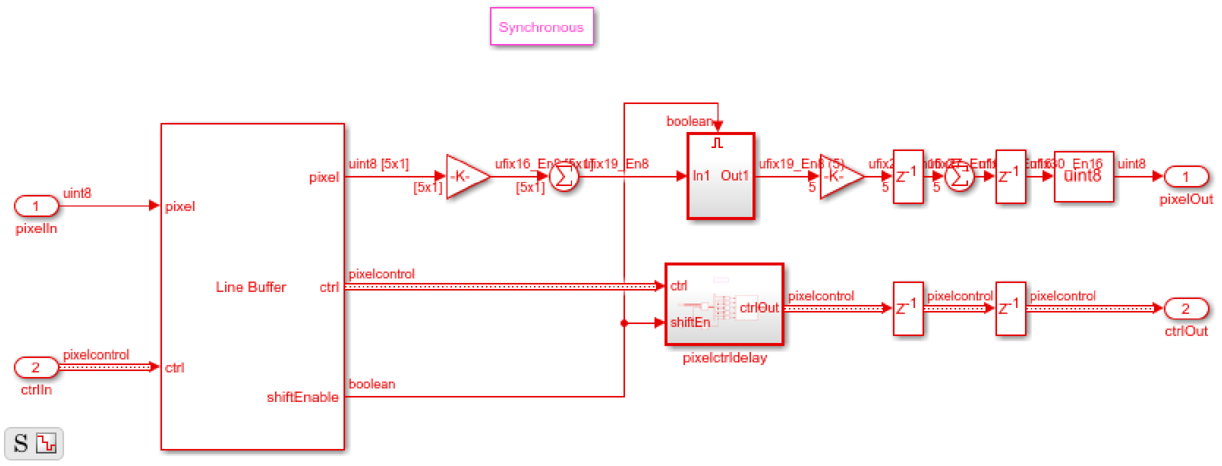
**Blocks**
Frame To Pixels | Pixels To Frame

## Construct a Filter Using Line Buffer

This example shows how to use the Line Buffer block to extract neighborhoods from an image for further processing. The model constructs a separable Gaussian filter.



Inside the HDL Algorithm subsystem, the Line Buffer block is configured for a 5-by-5 neighborhood. The output is a 5-by-1 column vector. The Gain and Sum blocks implement separate horizontal and vertical components of a 5-by-5 Gaussian filter with a 0.75 standard deviation. After vertical filtering, the model stores the column sums in a shift register that creates a 1-by-5 row vector. The row values are filtered again to calculate the new central pixel value of each neighborhood.



You can generate HDL code from the HDL Algorithm subsystem. You must have the HDL Coder™ software installed to run this command.

```
makehdl('SeparableFilterSimpleHDL/HDL Algorithm')
```

To generate an HDL test bench, use this command.

```
makehdltb('SeparableFilterSimpleHDL/HDL Algorithm')
```

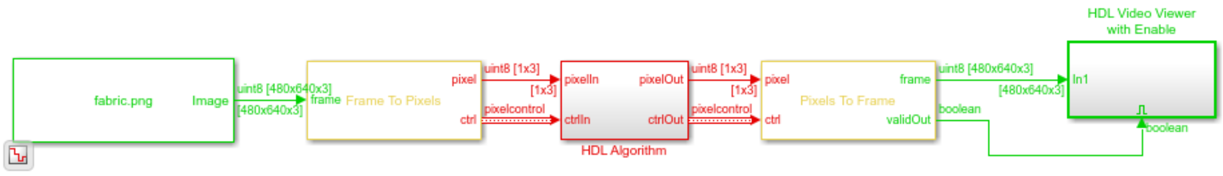
## See Also

**Blocks**
Frame To Pixels

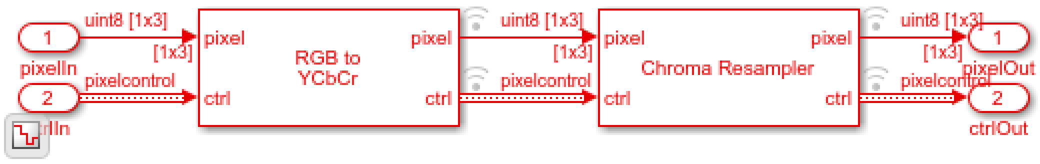
**Objects**
`visionhdl.LineBuffer`

## Convert RGB Image to YCbCr 4:2:2 Color Space

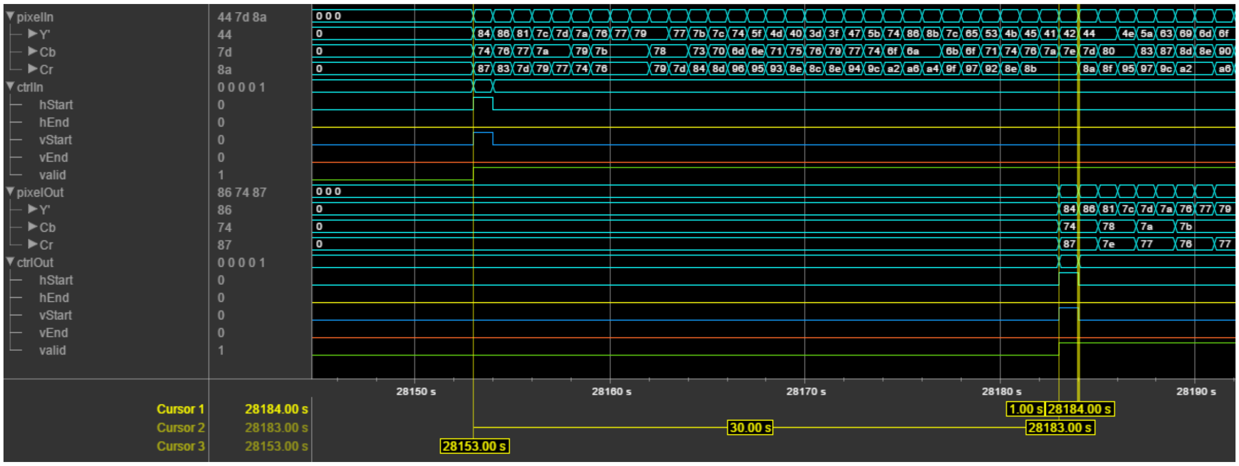
This example shows how to convert a pixel stream from R'G'B' color space to Y'CbCr 4:2:2 color space.



The model imports a 480p RGB image, then the Frame to Pixels block converts it to a pixel stream. Inside the HDL Algorithm subsystem, the Color Space Converter and Chroma Resampler blocks convert the pixel stream to YCbCr 4:2:2 format.



The waveform of the input and output pixel stream of the Chroma Resampler block shows the downsampling of the CbCr component values. The latency of the Chroma Resampler block depends on the size of the antialiasing filter. This example uses the default filter, which has 29 taps.



To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command.

```
makehdl('ChromaResampleExample/HDL Algorithm')
```

To generate the test bench, use the following command. Note that test bench generation takes a long time due to the large data size. Consider reducing the simulation time before generating the test bench.

```
makehdltb('ChromaResampleExample/HDL Algorithm')
```

The part of the model between the Frame to Pixels and Pixels to Frame blocks can be implemented on an FPGA.

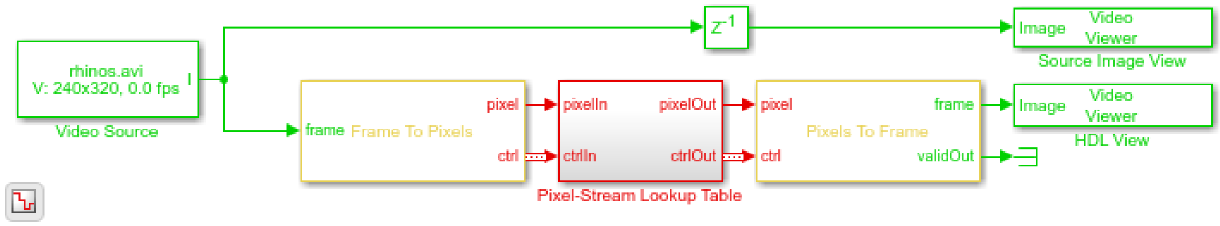
## See Also

**Blocks**

Chroma Resampler | Color Space Converter | Frame To Pixels

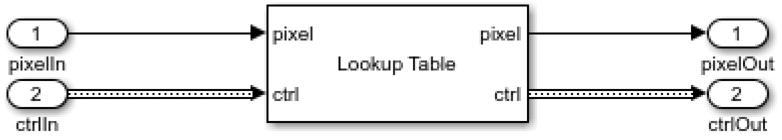
## Compute Negative Image

This example creates the negative of an image by looking up the opposite pixel values in a table.



For a hardware-compatible design, the model converts the input video to a stream of pixel values. The Frame to Pixels and Pixels to Frame blocks are configured to match the format of the video source.

The Pixel-Stream Lookup Table subsystem contains a Lookup Table block, configured with inversion data. The input pixel data is `uint8` type, so the negative value is `255 - pixel`, or `linspace(255,0,256)`. The output pixel data type is the same as the data type of the table contents, in this case, `uint8`.



To generate and check the HDL code referenced in this example, you must have an HDL Coder™ license.

To generate the HDL code, use the following command:

```
makehdl('LookupTableHDL/Pixel-Stream Lookup Table')
```

To infer a RAM to implement the lookup table, the `LUTRegisterResetType` property is set to none. To access this property, right-click the **Lookup Table** block inside the subsystem, and navigate to **HDL Coder > HDL Block Properties**.

To generate a test bench for the generated HDL code, use the following command:

```
makehdltb('LookupTableHDL/Pixel-Stream Lookup Table')
```

## See Also

**Blocks**
Frame To Pixels | Lookup Table

## Adapt Image Filter Coefficients from Frame to Frame

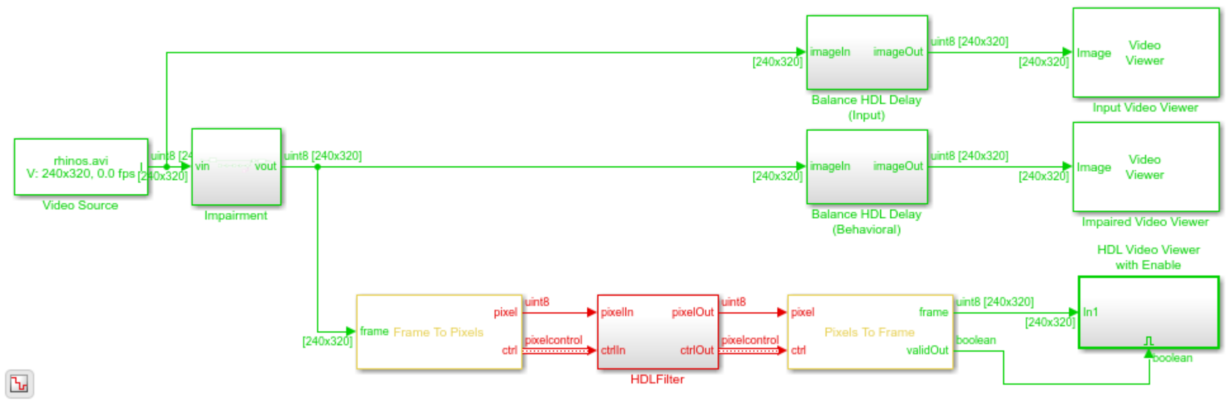
This example shows how to use programmable coefficients to correct a time-varying impairment on the input video.

There are many different techniques for filtering image and video signals that require filter coefficients that vary from frame to frame. To dynamically change the coefficients of the Image Filter block, set the **Filter coefficients source** parameter to `Input port`. The Image Filter block samples the input coefficient port at the beginning of each frame.

### The Example Model

The example model applies a brightness impairment to the input video, and the **HDL Filter** subsystem calculates filter coefficients for each frame and corrects the impairment. The model includes three video viewers: one for the original input video, another for the impaired video, and the third for the result of the filter that counteracts the impairment.

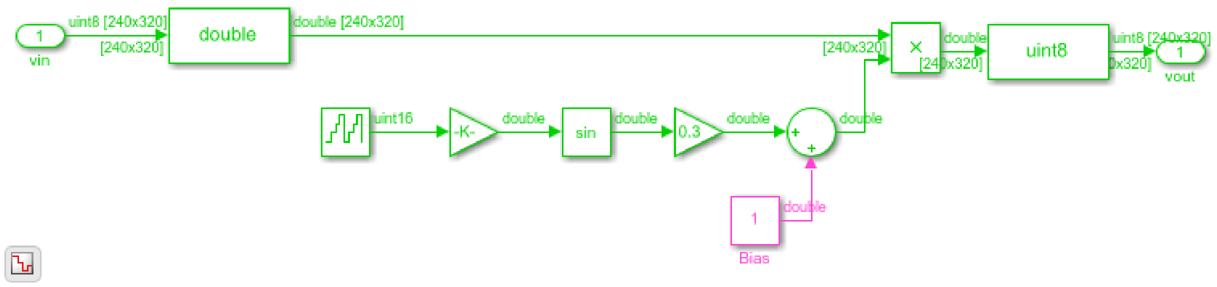
The model also includes Frame to Pixels and Pixels to Frame blocks to convert the matrix format video to streaming format suitable for HDL modeling.



### The Impairment

The impairment in this model is brightness modulation using a slow sine wave. Since the impairment is modeled purely behaviorally, the first step is to convert the image to double-precision values. The 16-bit counter counts up at the frame rate and the counter value is multiplied by 2*pi/40. The sine wave output is scaled down by 0.3 and a bias of

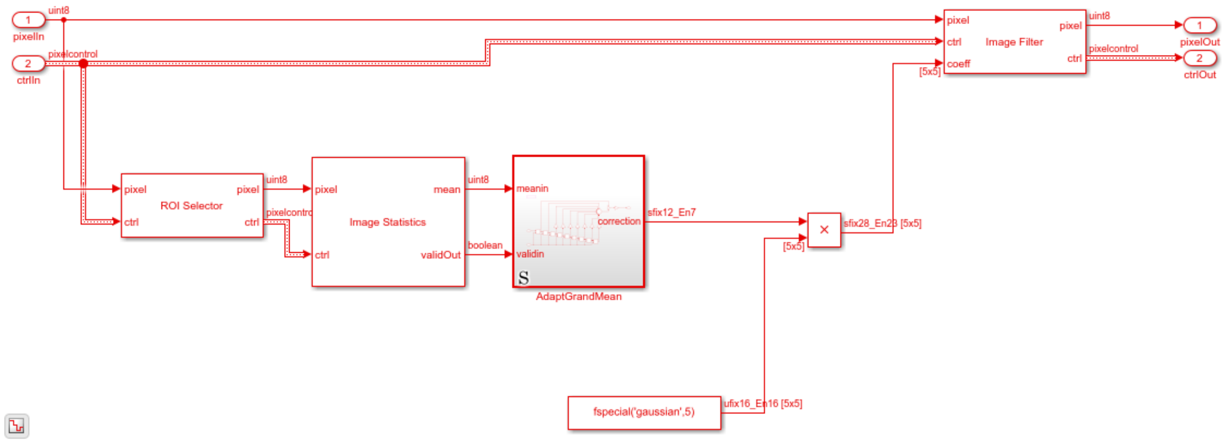
1.0 is then added. These calculations result in a +/-30% change in brightness over a period of 40 frames. After applying the impairment, the model converts back to `uint8` by using rounding with saturation.



## The Filter Algorithm

The HDL Algorithm subsystem starts by extracting a region of interest in the center of the image. Since this model is configured for a `320x240` video source, it uses a `100x100` region in the center of the video stream.

The Image Statistics block finds the mean of that central region. A new mean is computed for each `100x100` frame. The block sets the validOut port to `true` to indicate when the new mean is valid.



### Compute the Scaled Grand Mean

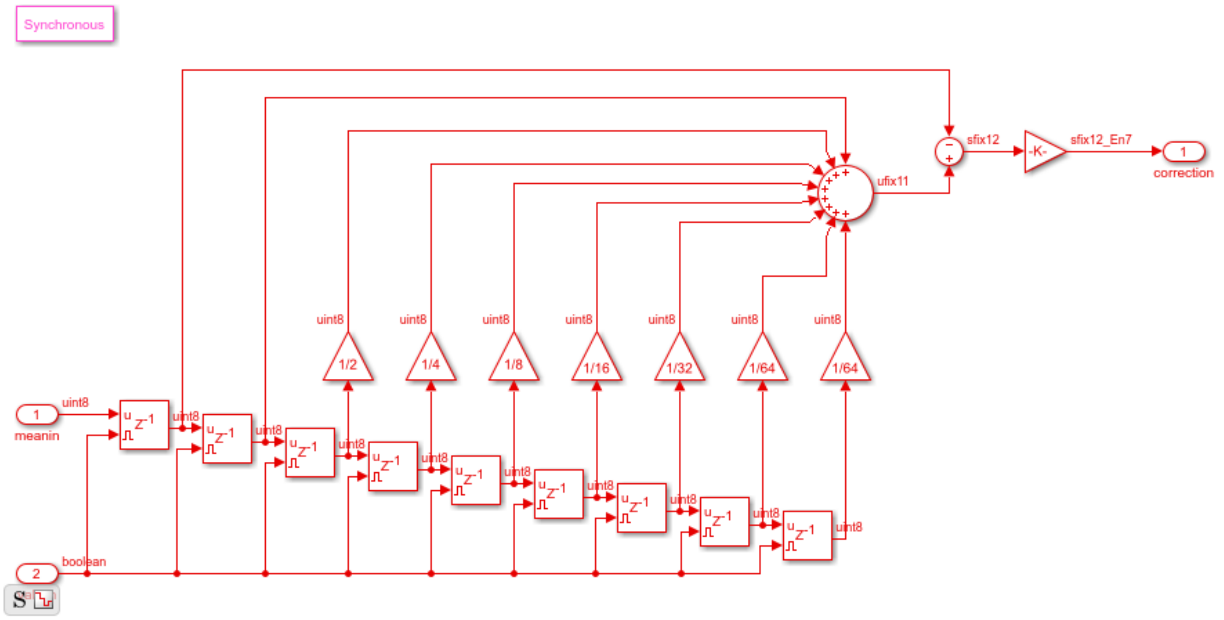
The **Adapt Grand Mean** subsystem computes the correction factor required to counteract the impairment.

You could use central-region-mean brightness directly, with a "gray-world" assumption that the average brightness is mid-scale (128 in this case). But, a more accurate approach is to use the previous brightness means, with the assumption that the average brightness does not change quickly frame to frame.

Forming a mean of means is known as a grand mean, but that calculation would give equal weight to the past frames. Instead, the subsystem weights the past frames with an exponential fractional decay with the coefficients `[1 1/2 1/4 1/8 1/16 1/32 1/64 1/64]`. The last coefficient would normally be `1/128` but by adjusting that value, the sum of the weights becomes exactly 2, making the normalizing factor a simple shift operation. Note that the initial value of all the delay line registers is mid-scale (128) to avoid large start-up transients in the correction.

The subsystem finds the correction factor using the current mean and the weighted grand mean. Since the grand mean scaled up by 2, if you subtract the current mean from it, the resulting value is the weighted grand mean plus or minus the error term in the direction of correcting the error.

The correction is then scaled by 2^-7 and sent to the output port. A normalization could be applied here by dividing by the grand mean, but in practice, simple scaling works well enough.



### Apply the Correction

The correction output from the **Adapt Grand Mean** subsystem is then used to scale the filter coefficients, in this case a Gaussian filter of size `5x5` with a standard deviation of the default 0.5. In the actual FPGA this filter uses 25 multipliers. Pipelining is of no concern here since these values are computed well before they are needed. The block samples the coefficient port when the `vStart` signal in the input ctrl bus is `true`.

### Going Further

In this simple example, you could alternatively apply the correction factor to the scalar pixel stream and then filter. The architecture shown can expand for more complex adaptive changes in the filter coefficients.

The `5x5` multiply of the correction factor with the gaussian coefficients could be implemented as a single serial multiplier rather than 25 parallel multipliers. To achieve this HDL implementation, include the Product block in a Subsystem, and right-click the Subsystem to open the HDL Block Properties. Set the **SharingFactor** property to 25 to implement a single time-multiplexed multiplier. With this setting, the multiply operation uses a 25-times faster clock than the rest of the design. Consider your required pixel clock speed and whether your device can accommodate the faster rate.

## See Also

**Blocks**
Image Filter | Image Statistics | ROI Selector