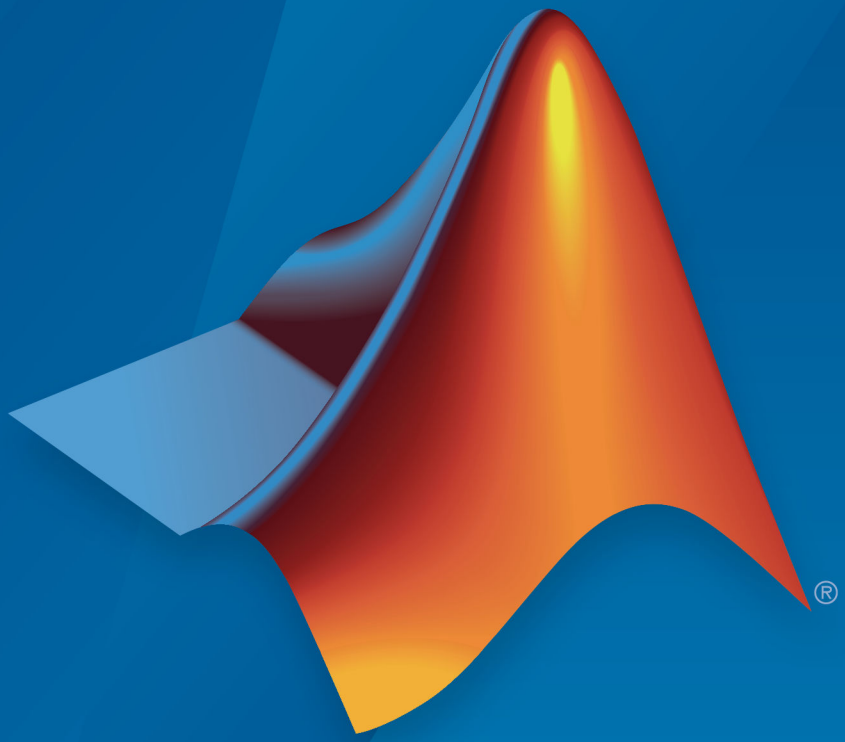
Vision HDL Toolbox™

User's Guide



MATLAB®

R2017b

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Vision HDL Toolbox™ User's Guide

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

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)

Contents

1 Streaming Pixel Interface

2 Algorithms

3 Best Practices

4 Prototyping

5 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.

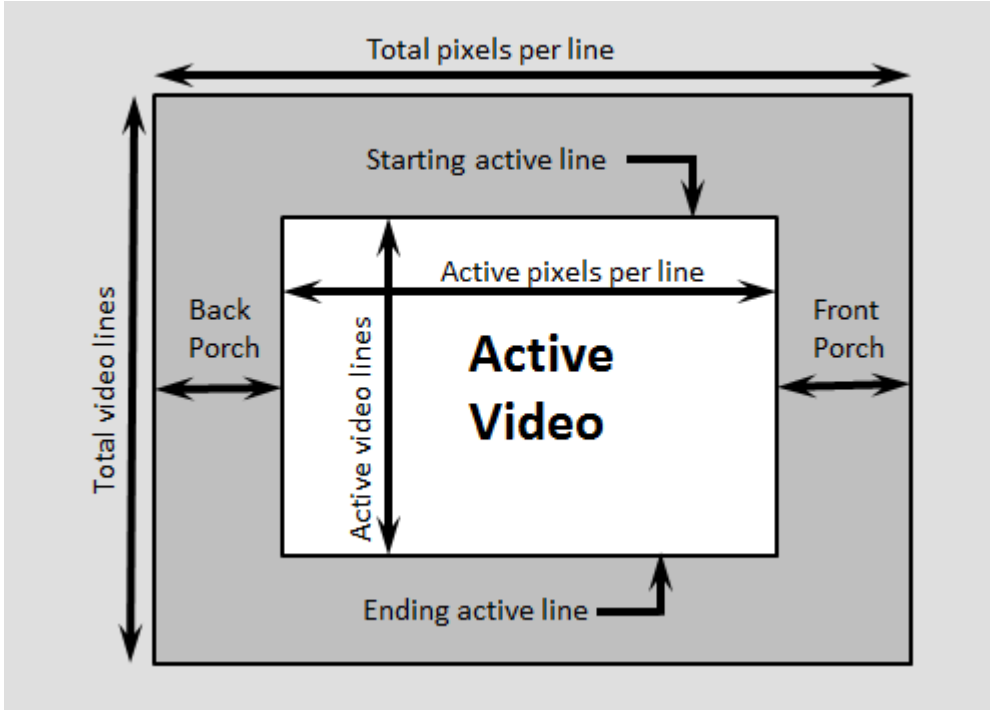
How Does a Streaming Pixel Interface Work?

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`.

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.

If your data is already in a serial format, design your own logic to generate these control signals from your existing serial control scheme.

Supported Pixel Data Types

Vision HDL Toolbox blocks and objects include ports or arguments for streaming pixel data. The blocks and objects capture one pixel at a time from the input, and produce one pixel at a time for output. 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 has multiple 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.

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 to the `step` method representing the pixel data. The following table describes the format of the pixel data.

Port or Argument	Description	Data Type
`pixel`	Scalar that represents binary or grayscale pixel value, or a vector of 2 to 4 values representing a color pixel.	Supported data types can include: • `boolean` or `logical` • `uint` or `int` • `fixdt()` `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.

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 1-8.

In MATLAB, the control signal argument is a structure. For details of the structure data type, see “Pixel Control Structure” on page 1-10.

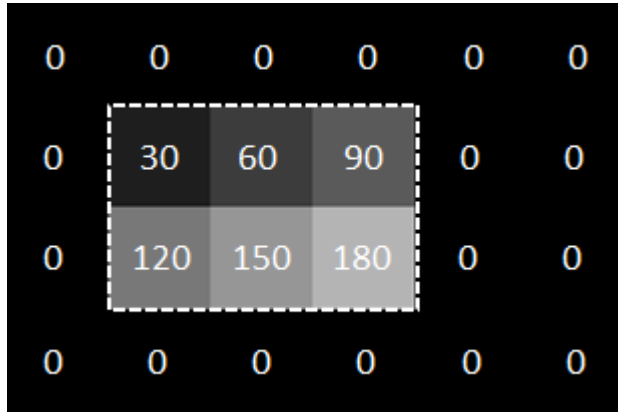
Timing Diagram of Serial Pixel Interface

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:

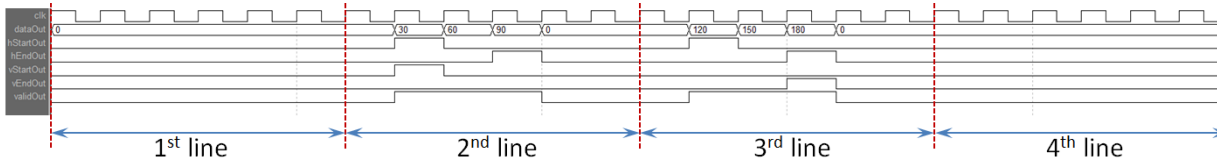
- 1-pixel-wide back porch
- 2-pixel-wide front porch
- 1 line before the first active line
- 1 line after the last active line

You can configure the dimensions of the active and inactive regions with the Frame To Pixels block or `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. The diagram shows the serial output of the Frame To Pixels block for this frame.



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 Video Processing Algorithms for HDL in MATLAB”.

See Also

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

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`

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 connect the data and control output of a `step` method to the input of another `step` method, because Vision HDL Toolbox objects use this structure for input and output. To convert an image into a pixel stream and control signals, use the `FrameToPixels` 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

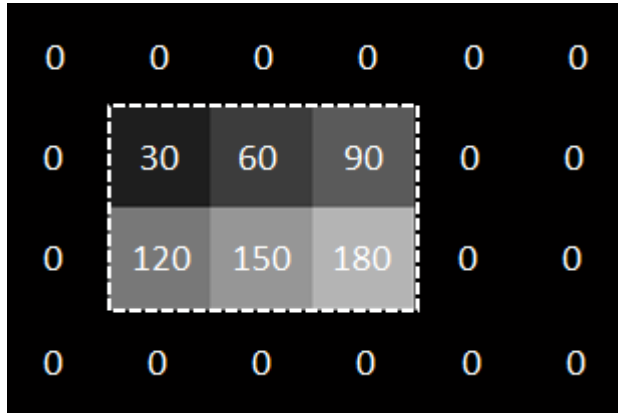
This example converts Camera Link® signals to the `pixelcontrol` structure, inverts the pixels with a Vision HDL Toolbox object, and converts the control signals back to the Camera Link format.

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.

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

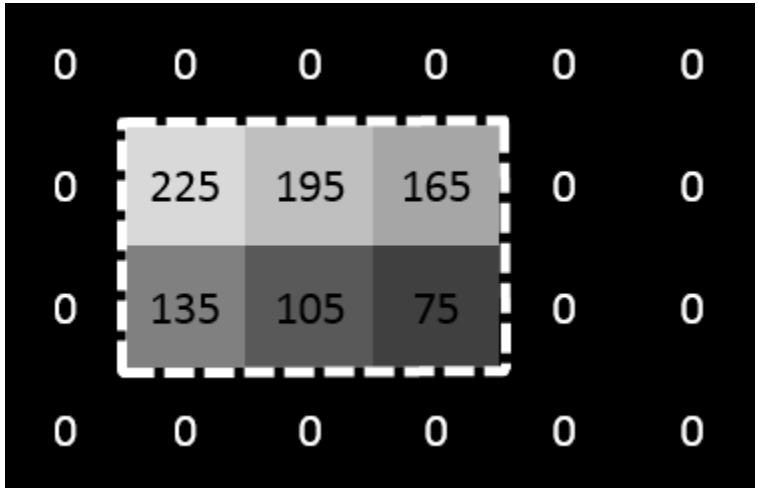
To invert a Camera Link pixel stream using these components, write a test bench script that:

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

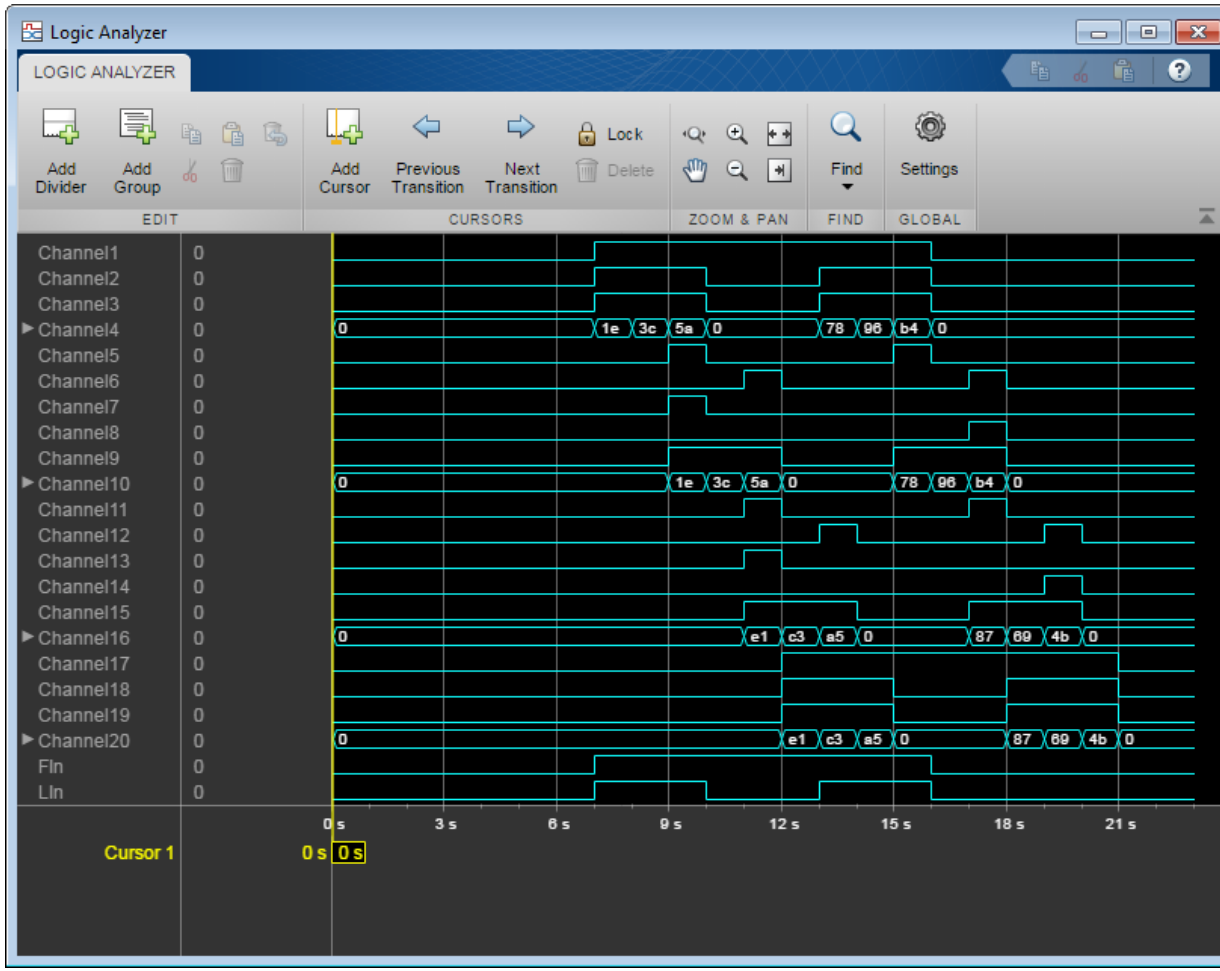
```
[~,numPixelsPerFrame] = size(pixel);
pixOut = zeros(numPixelsPerFrame,1,'uint8');
pixel_d = zeros(numPixelsPerFrame,1,'uint8');
pixOut_d = zeros(numPixelsPerFrame,1,'uint8');
DOut = false(numPixelsPerFrame,1);
FOut = false(numPixelsPerFrame,1);
LOut = false(numPixelsPerFrame,1);
ctrl = repmat(pixelcontrolstruct,numPixelsPerFrame,1);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

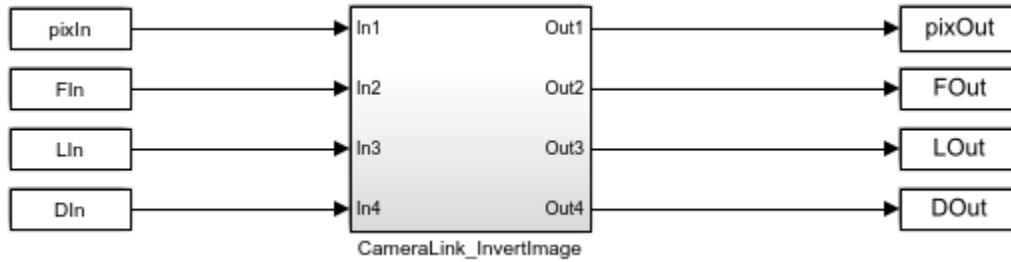
Integrate Vision HDL Blocks Into Camera Link System

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.

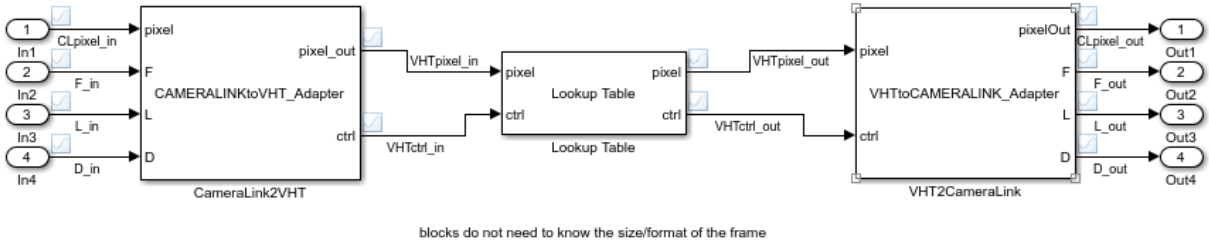


Structure of the Subsystem

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.

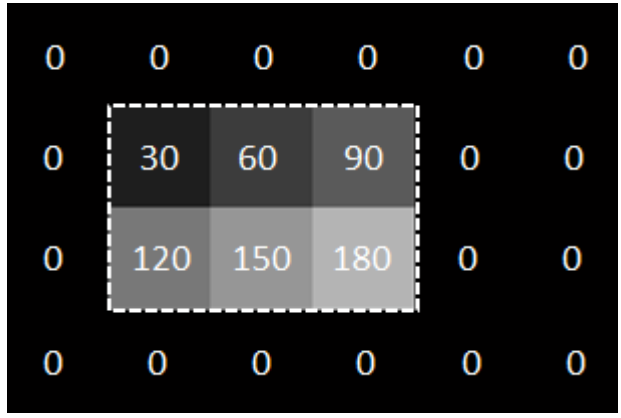
You can put any combination of Vision HDL Toolbox blocks into the middle of the subsystem. This example uses an inversion Lookup Table.

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.



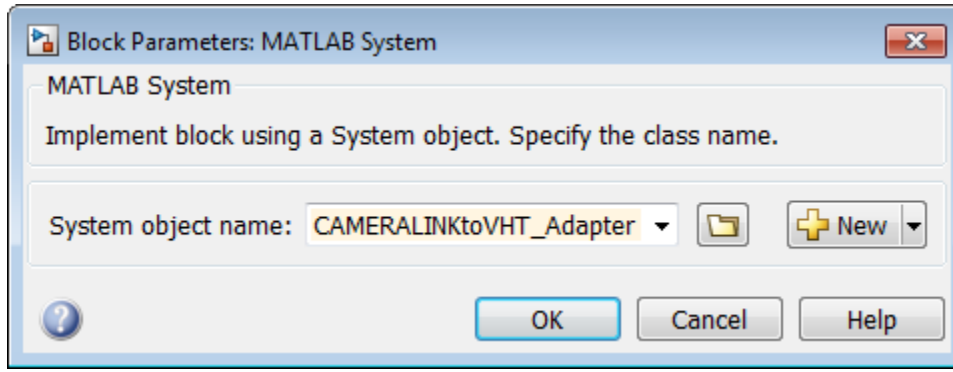

```
FIn = logical([0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0]);
LIn = logical([0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0]);
DIn = logical([0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0]);
pixIn = uint8([0,0,0,0,0,0,0,30,60,90,0,0,0,120,150,180,0,0,0,0,0,0,0,0]);
```

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 1-11 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.

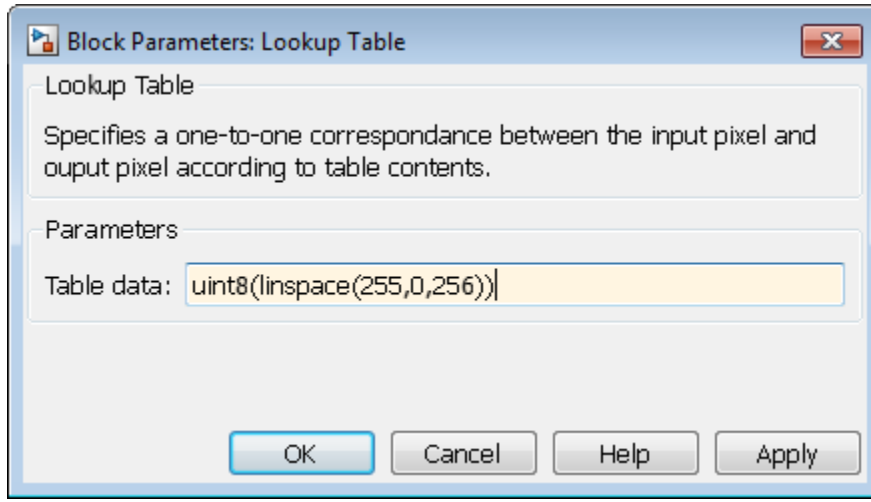
Create a MATLAB System block and point it to the System object.



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.



Convert pixelcontrol to Camera Link

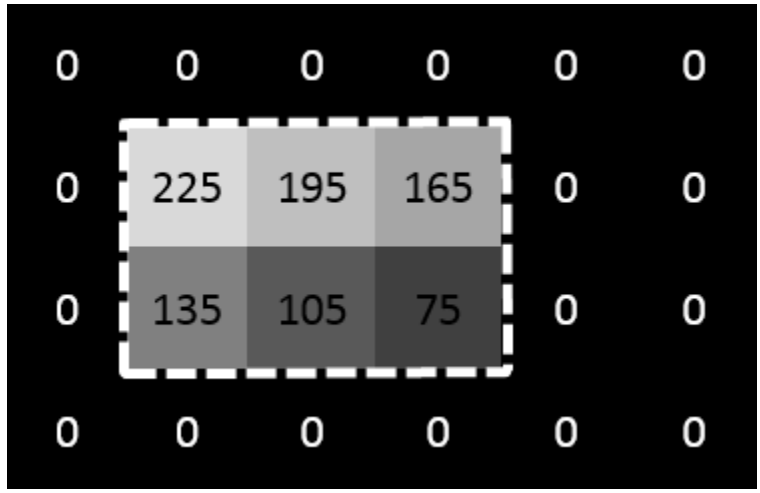
Write a custom System object to convert Vision HDL Toolbox signals back to the Camera Link format. This example uses the object designed in the “Convert Camera Control Signals to pixelcontrol Format” on page 1-11 example.

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.

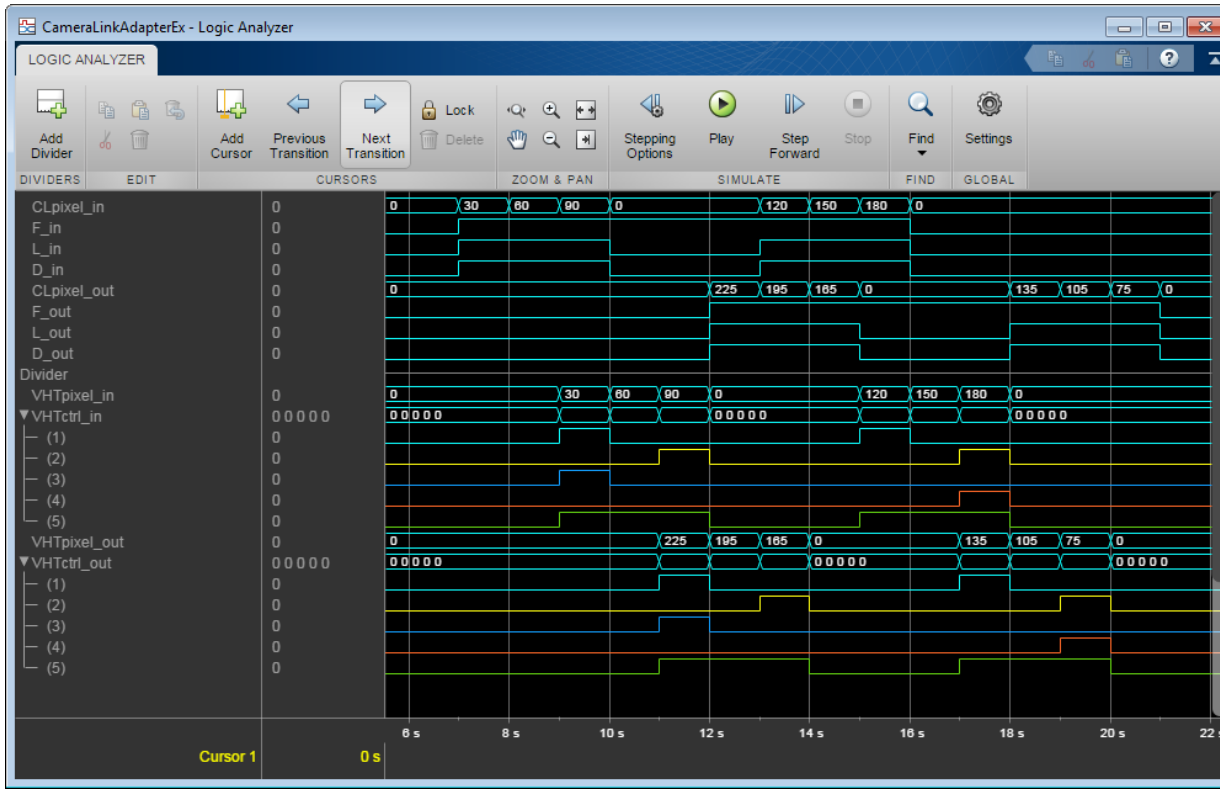
Create a second MATLAB System block and point it to the System object.

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

2

Algorithms

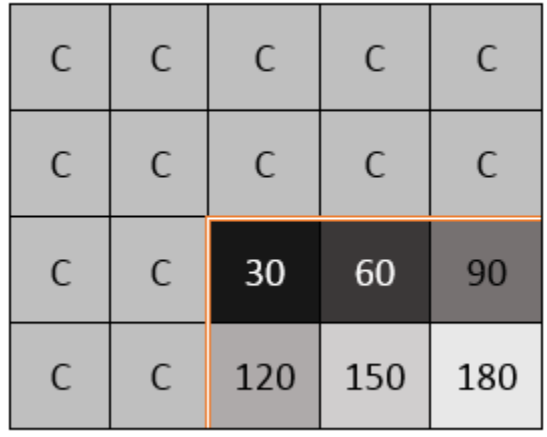
Edge Padding

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.

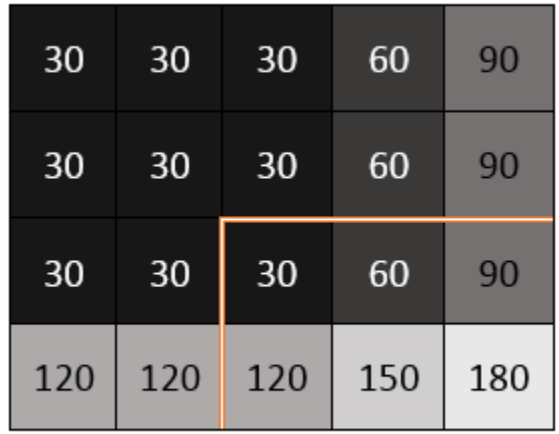
- `Constant` — Each added pixel is assigned the same value. On some blocks and System objects you can specify the constant value. The value `0`, representing black, is a reserved value in some video standards. It is common to choose a small number, such as `16`, as a near-black padding value.

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



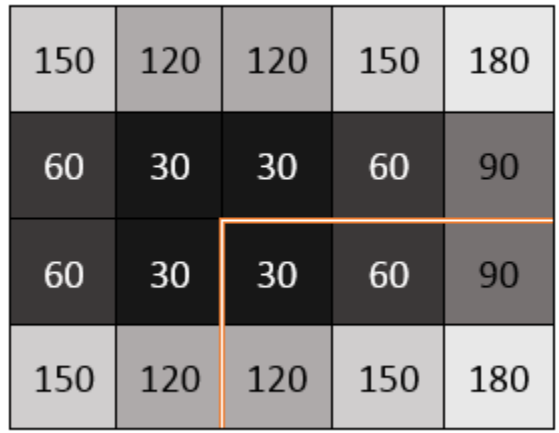
- `Replicate` — The pixel values at the edge of the active frame are repeated to make rows and columns of padding pixels.

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.



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

Best Practices

Accelerate a MATLAB Design With MATLAB Coder

Vision HDL Toolbox designs in MATLAB must call the `step` method of 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”.

4

Prototyping

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 4-2. 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 System 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 “Supported Blocks” (HDL Coder) pages for block implementations, properties, and restrictions for HDL code generation.

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 `hdlsetup` 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 CodeGenerate 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 CodeHDL 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

- “Generate Code Using the HDL Workflow Advisor” (HDL Coder)
- “Generate HDL Code Using the Command Line” (HDL Coder)
- “Test Bench Overview” (HDL Coder)

Generate HDL Code From MATLAB

This example show you how to generate HDL code from the design in “Design Video Processing Algorithms for HDL 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"

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 Altera® or Xilinx® 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 on the FPGA.

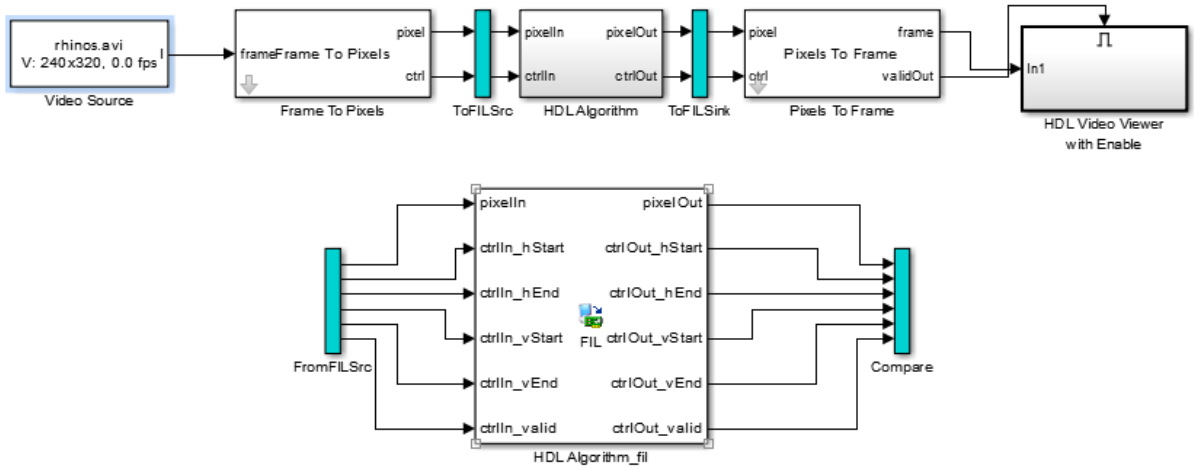
Vision HDL Toolbox provides the FIL Frame To Pixels and FIL Pixels To Frame blocks to accelerate communication between Simulink and the FPGA board. In MATLAB, you can modify the generated code to speed up communication with the FPGA board.

In this section...
“FIL In Simulink” on page 4-10
“FIL In MATLAB” on page 4-12

FIL In Simulink

Autogenerated FIL Model

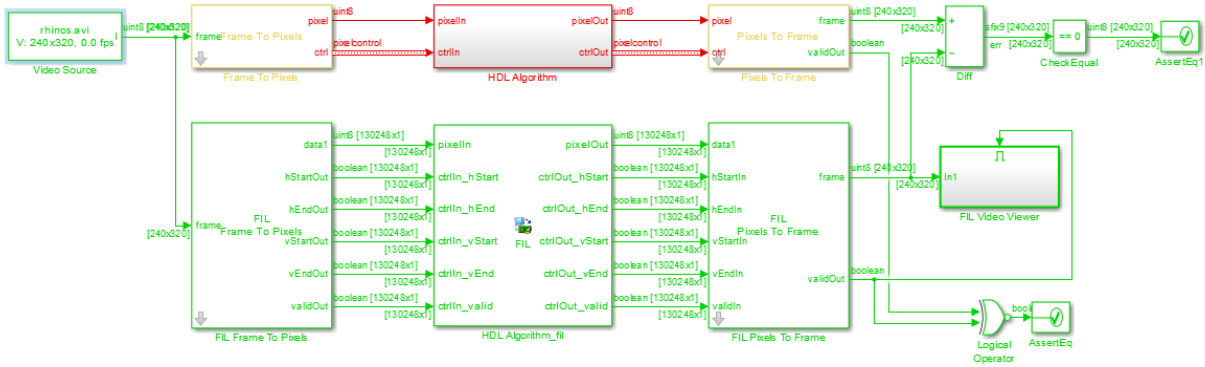
When you generate a programming file for a FIL target in Simulink, the tool creates a model to compare the FIL simulation with your Simulink design. For Vision HDL Toolbox designs, the FIL block in that model replicates the pixel-streaming interface and sends one pixel at a time to the FPGA.



The blue ToFILSrc subsystem branches the pixel-stream input of the HDL Algorithm block to the FromFILSrc subsystem. The blue ToFILSink subsystem branches 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 modify the autogenerated model. The modified model uses the FIL Frame To Pixels and FIL Pixels To Frame blocks to send one frame at a time.



1. Remove the blue 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. Select the **Vector size** in the FIL Frame To Pixels and FIL Pixels To Frame blocks as `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 interfaces of the FIL block does not modify the generated HDL code. It affects only the packet size of the communication between the simulator and the FPGA board.

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

FIL In MATLAB

Autogenerated FIL Function

When you generate a programming file for a FIL target in MATLAB, the tool creates a test bench to compare the FIL simulation with your MATLAB design. For Vision HDL

Toolbox designs, the *name*_fil function in the test bench replicates the pixel-streaming interface and sends one pixel at a time to the FPGA.

This code snippet is from the generated test bench *name*_tb_fil.m. The code calls the generated *name*_fil function once for each pixel in a frame.

```
for p = 1:numPixPerFrm
  [hStartIn,hEndIn,vStartIn,vEndIn,validIn] = pixelcontrolsignals(ctrlIn(p));
  [pixOut(p),hStartOut,hEndOut,vStartOut,vEndOut,validOut] = .../
      visionhdlgamma_design_fil(pixIn(p),hStartIn,hEndIn,vStartIn,...
                                         vEndIn,validIn);
  ctrlOut(p) = pixelcontrolstruct(hStartOut,hEndOut,vStartOut,...
                                  vEndOut,validOut);
end
```

The generated *name*_fil function calls your HDL-targeted function. It also calls the *name*_sysobj_fil function, which contains a System object™ that connects to the FPGA. *name*_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 *name*_fil.m.

```
% Call the original MATLAB function to get reference signal
[ref_pixOut,ref_hStartOut,ref_hEndOut,ref_vStartOut,ref_vEndOut,ref_validOut] = ...
   visionhdlgamma_design(pixIn,hStartIn,hEndIn,vStartIn,vEndIn,validIn);

% Run FPGA-in-the-loop
[pixOut,hStartOut,hEndOut,vStartOut,vEndOut,validOut] = ...
  visionhdlgamma_design_sysobj_fil(pixIn,hStartIn,hEndIn,vStartIn,...
     vEndIn,validIn);

% Convert output signals
hStartOut = logical(hStartOut);
hEndOut = logical(hEndOut);
vStartOut = logical(vStartOut);
vEndOut = logical(vEndOut);
validOut = logical(validOut);

% Verify the FPGA-in-the-loop output against the reference
hdlverifier.assert(pixOut,ref_pixOut,'pixOut')
hdlverifier.assert(hStartOut,ref_hStartOut,'hStartOut')
hdlverifier.assert(hEndOut,ref_hEndOut,'hEndOut')
hdlverifier.assert(vStartOut,ref_vStartOut,'vStartOut')
hdlverifier.assert(vEndOut,ref_vEndOut,'vEndOut')
hdlverifier.assert(validOut,ref_validOut,'validOut')
```

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

Modified FIL Test Bench for Pixel Streaming

To improve the communication bandwidth with the FPGA board, you can modify the autogenerated test bench, *name*`_tb_fil.m`. The modified test bench calls the FIL System object directly, with one frame at a time.

Declare an instance of the `class_`*name*`_sysobj` System object. Preallocate vectors for the input pixel data and control signals.

```
fil = class_visionhdlgamma_design_sysobj;
hStartIn = true(numPixPerFrm,1);
hEndIn   = true(numPixPerFrm,1);
vStartIn = true(numPixPerFrm,1);
vEndIn   = true(numPixPerFrm,1);
validIn  = true(numPixPerFrm,1);
```

Comment out the loop over the pixels in the frame, as well as the call to the `pix2frm` object.

```
%     for p = 1:numPixPerFrm
%       [hStartIn(p),hEndIn(p),vStartIn(p),vEndIn(p),validIn(p)] = ...\
%       pixelcontrolsignals(ctrlIn(p));
%       [pixOut(p),hStartOut,hEndOut,vStartOut,vEndOut,validOut] =...
%          visionhdlgamma_design_fil(pixIn(p),hStartIn(p),hEndIn(p),...
%                                    vStartIn(p),vEndIn(p),validIn(p));
%       ctrlOut(p) = pixelcontrolstruct(hStartOut,hEndOut,vStartOut,...
%                                       vEndOut,validOut);
%     end
%     frmOut = step(pix2frm,pixOut,ctrlOut);
```

Call the `step` method of the `class_`*name*`_sysobj` object with vectors containing the whole frame of data pixels and control signals. Pass each control signal to `step` separately, as a vector of `logical` values. After this `step` call, recombine the control signal vectors into a vector of structures. Convert the vector data back to a matrix representing the active frame using the `pix2frm` object.

```
[fil_pixOut,fil_hStartOut,fil_hEndOut,fil_vStartOut,fil_vEndOut,fil_validOut] = ...
    step(fil,pixIn,[ctrlIn.hStart]',[ctrlIn.hEnd]',[ctrlIn.vStart]',...
           [ctrlIn.vEnd]',[ctrlIn.valid]');
fil_ctrlOut = arrayfun(@(hStart,hEnd,vStart,vEnd,valid) ...
```

```
    struct('hStart',hStart,'hEnd',hEnd,'vStart',vStart,'vEnd',vEnd,...
          'valid',valid),fil_hStartOut,fil_hEndOut,fil_vStartOut,...
           fil_vEndOut,fil_validOut);
frmOut_fil = step(pix2frm,fil_pixOut,fil_ctrlOut);
```

These code changes remove the pixel-by-pixel verification of the FIL results against the MATLAB results. Optionally, you can add a frame-by-frame comparison of the results. Calling the original pixel-by-pixel function for a reference slows down the FIL simulation. Leave the pixel loop intact, but instead of calling *name*`_fil`, call the original *name* function.

```
for p = 1:numPixPerFrm
   [hStartIn,hEndIn,vStartIn,vEndIn,validIn] = pixelcontrolsignals(ctrlIn(p));
   [ref_pixOut(p),hStartOut,hEndOut,vStartOut,vEndOut,validOut] = ...\
      visionhdlgamma_design(pixIn(p),hStartIn,hEndIn,vStartIn,vEndIn,validIn);
   ref_ctrlOut(p) = pixelcontrolstruct(hStartOut,hEndOut,vStartOut,vEndOut,...
      validOut);
end
```

After the call to the `class_`*name*`_sysobj` object, compare the two output vectors.

```
hdlverifier.assert(fil_pixOut,ref_pixOut,'pixOut')
hdlverifier.assert(fil_ctrlOut,ref_ctrlOut,'ctrlOut')
```

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

More About

- “FPGA Verification” (HDL Verifier)

Prototype Vision Algorithms on Zynq-Based Hardware

You can use the Computer Vision System 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. (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.

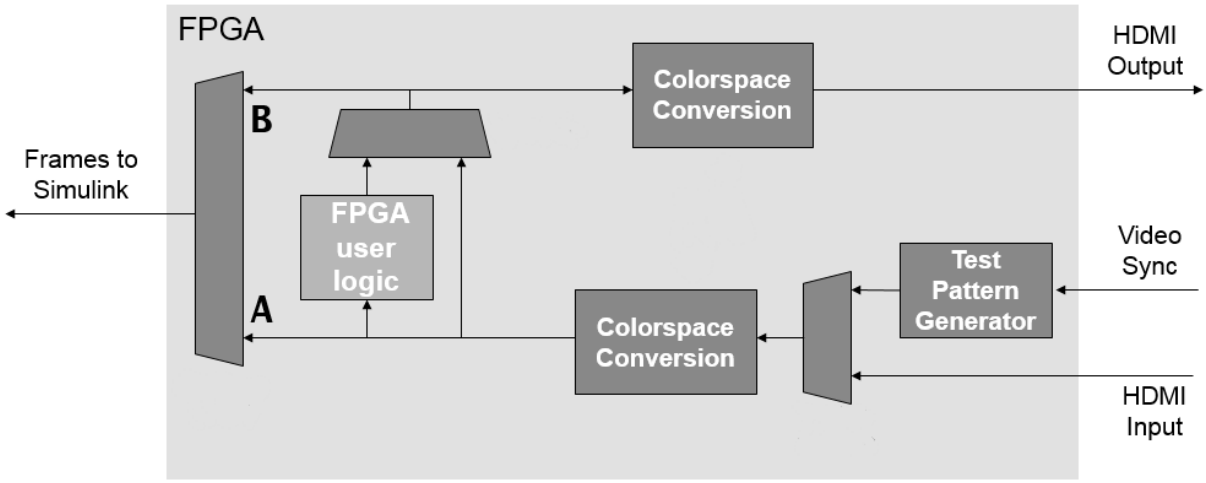
Once you have video frames in Simulink, you can:

- Design frame-based video processing algorithms that operate on the live data input. Use blocks from the Computer Vision System 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 System 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.



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.

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

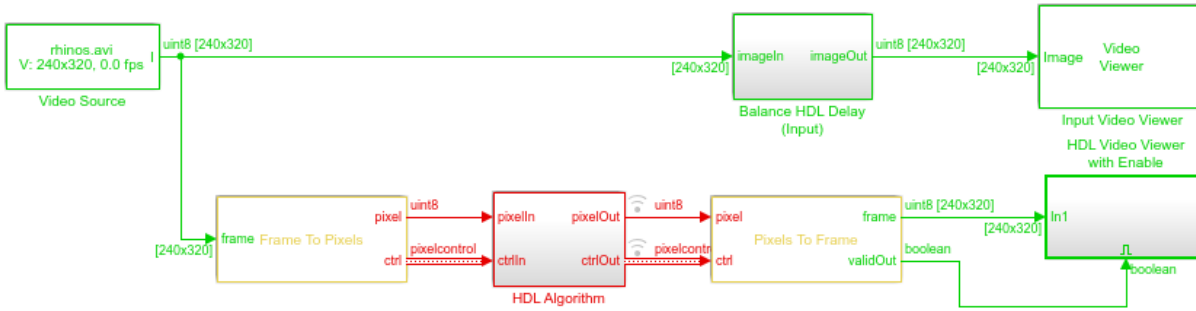
More About

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

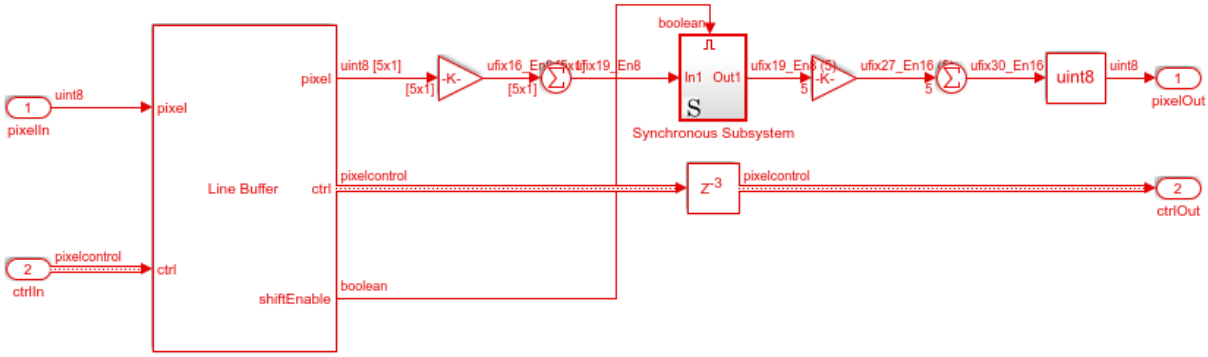
Examples

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

System Objects

`visionhdl.LineBuffer`