Vision HDL Toolbox™

User's Guide

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Revision History

Contents

Code Generation and Deployment

3

Block Reference Examples

4

Streaming Pixel Interface

Streaming Pixel Interface

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.
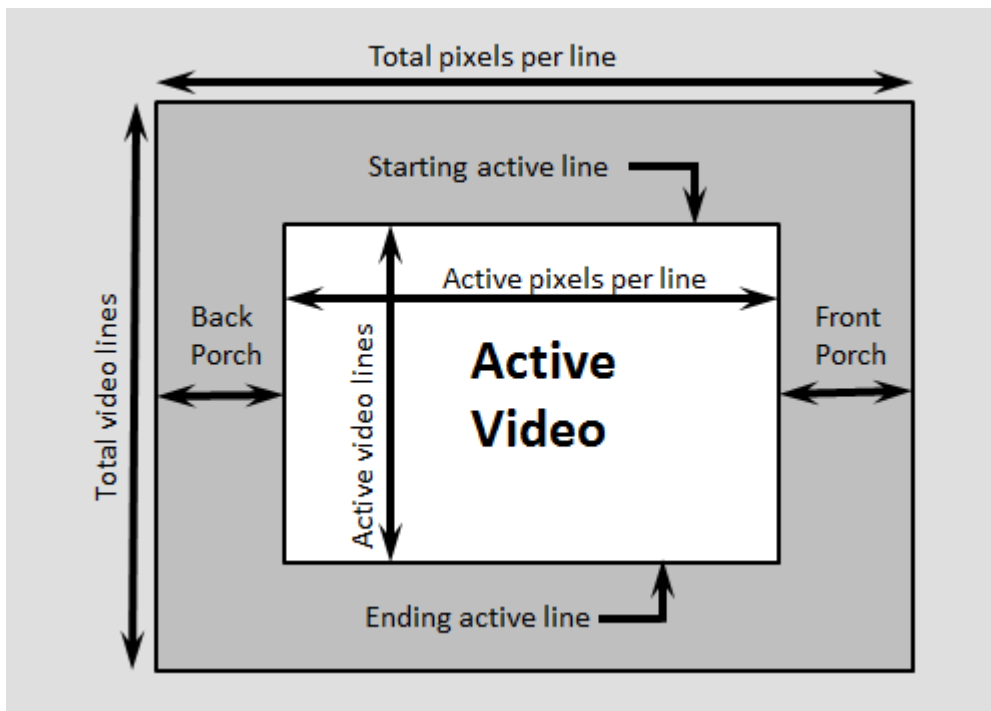
Why Use a Streaming Pixel Interface?

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 1-28.

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. 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 • Multipixel streaming — Column vector of four or eight pixel values 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: • `boolean` or `logical` • `uint` or `int` • `fixdt()` `double` and `single` data types are supported for simulation, but not for HDL code generation.

Note These blocks support multipixel streaming:

- Image Filter
- Edge Detector
- Median Filter
- Line Buffer
- Binary morphology: Closing, Dilation, Erosion, and Opening

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 1-21.

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.
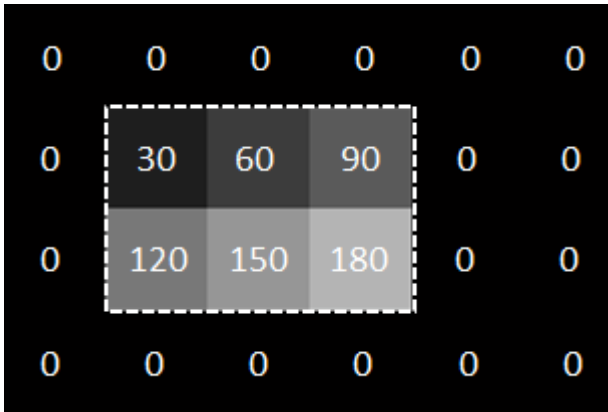
The block or object serializes the image from left to right, one line at a time. The timing diagram shows the control signals and pixel data that correspond to this image, which is the serial output of the Frame To Pixels block for this frame, configured for single-pixel streaming.
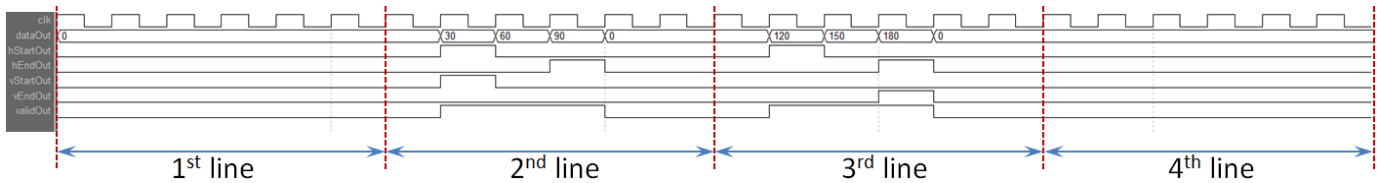
For an example using the Frame to Pixels block to serialize an image, see "Design Video Processing Algorithms for HDL in Simulink".

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.

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.
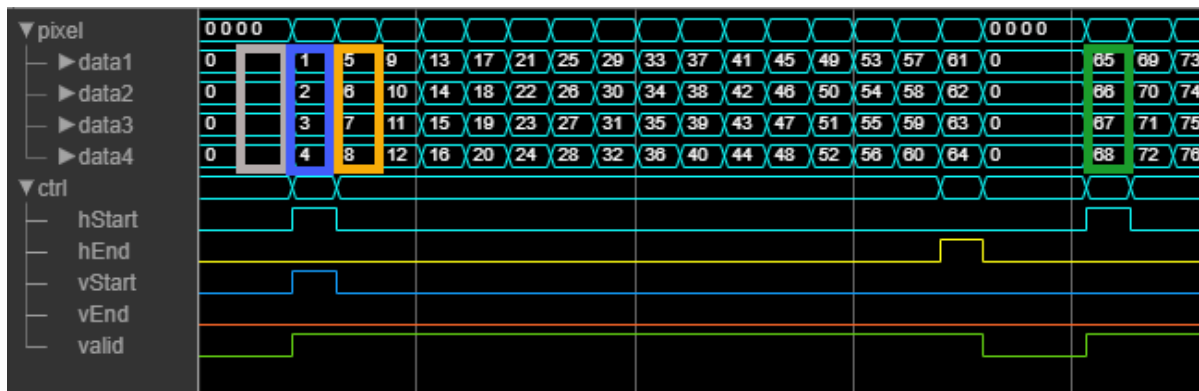
For an example model that uses multipixel streaming, see "Filter Multipixel Video Streams" on page 1-8.

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 1-8

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.
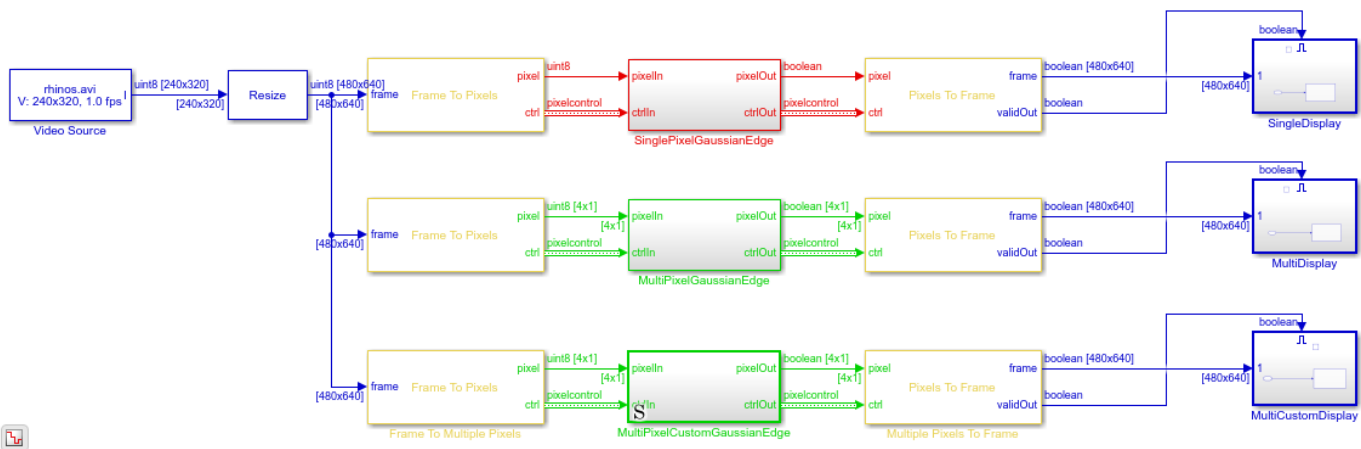
Multipixel Streaming Using Library Blocks

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.
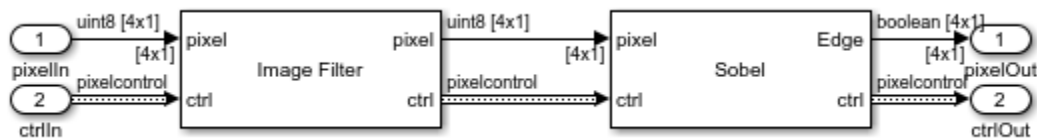
Custom Multipixel Algorithms

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.
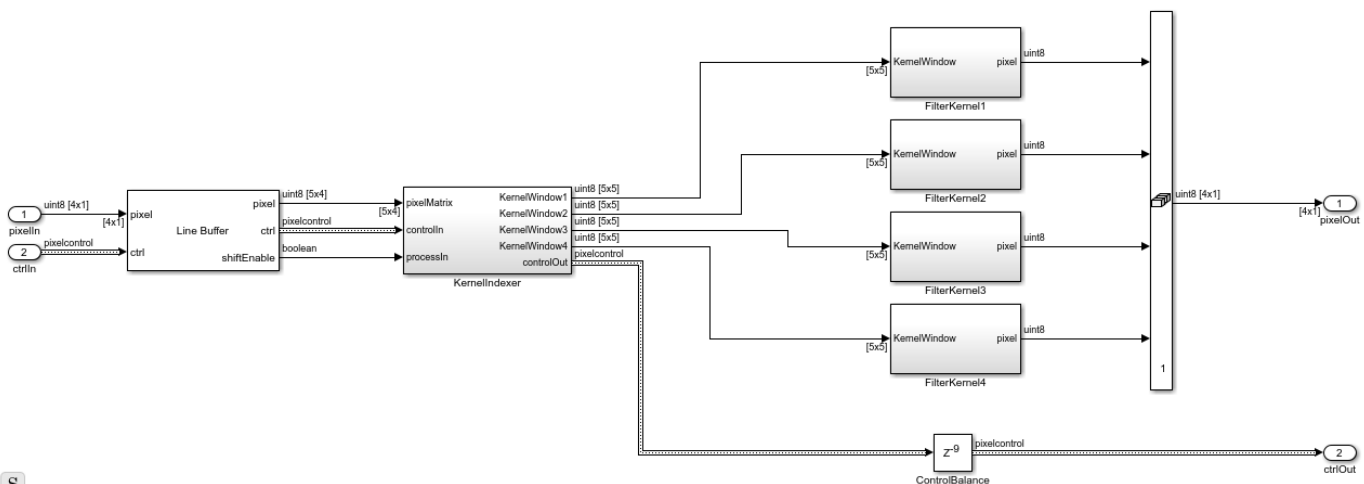
Form Kernels from Line Buffer Output

The KernelIndexer subsystem forms 4 [5x5] filter kernels from the 2-D output of the Line Buffer block.
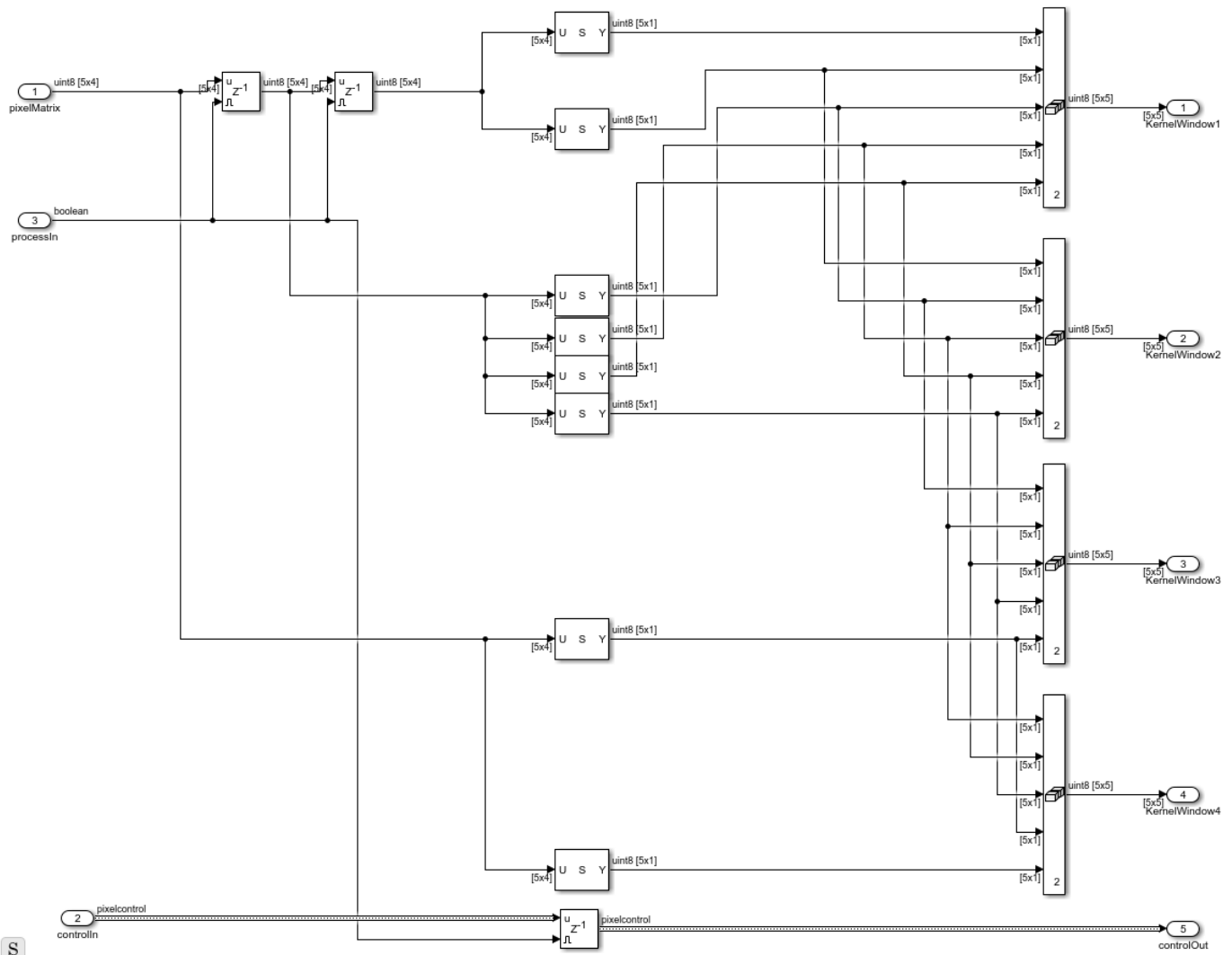
The diagram shows how the filter kernel is extracted from the [5x4] output stream, for the kernel that is centered on the first pixel in the [4x1] output. This first kernel includes pixels from 2 adjacent [5x4] Line Buffer outputs.

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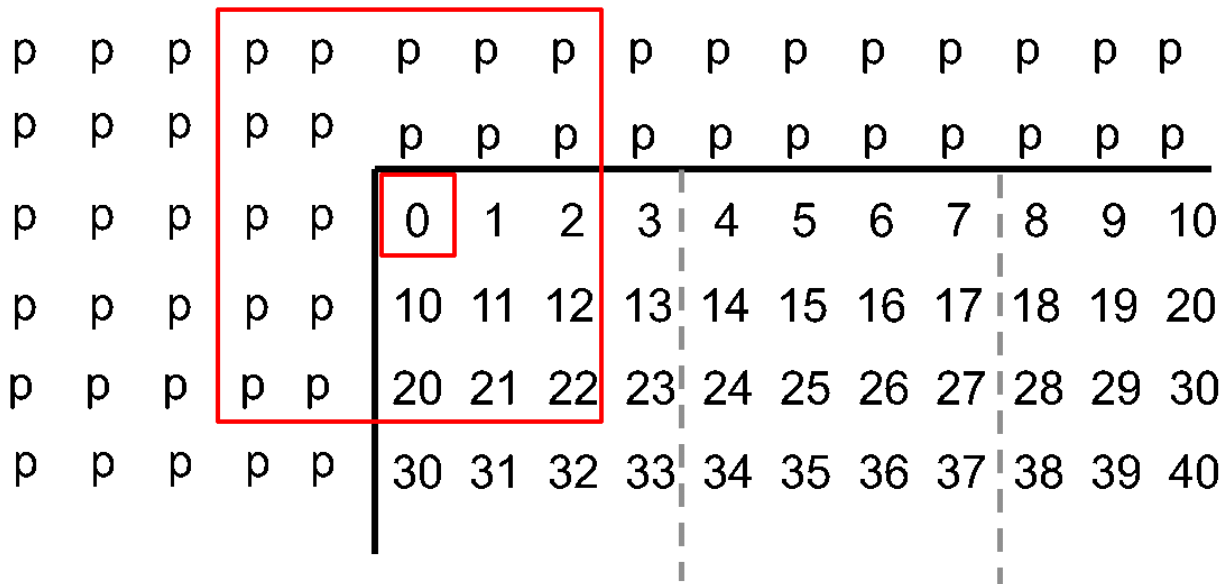
The kernel centered on the last pixel in the [4x1] output also includes the third adjacent [5x4] output. So, to form four [5x5] kernels, the subsystem must access columns from three [5x4] matrices.

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
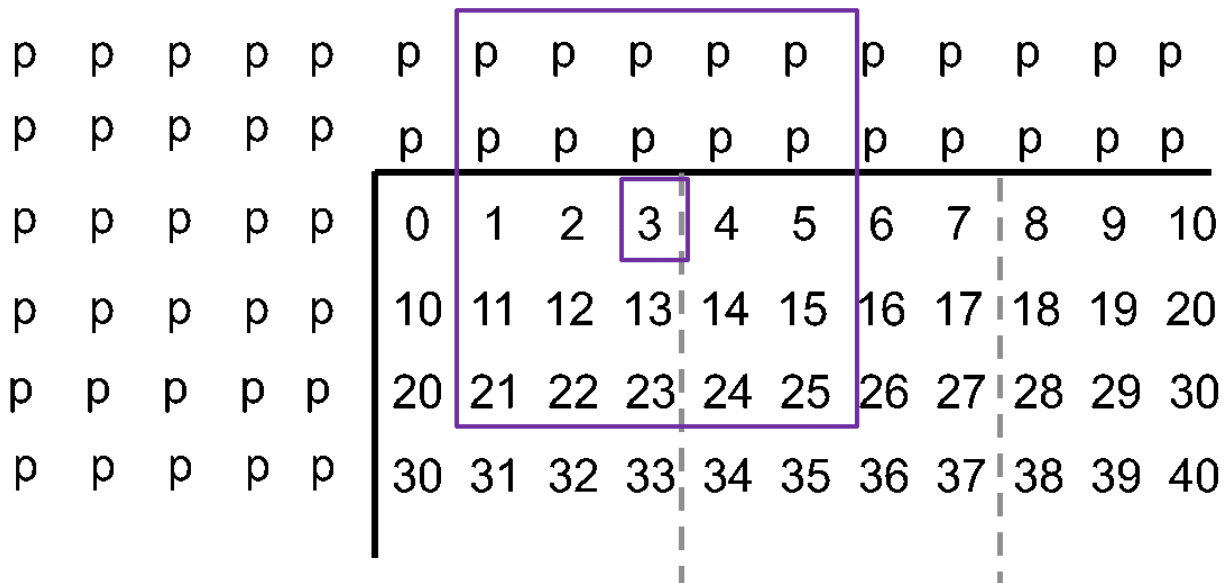
The KernelIndexer subsystem uses the current [5x4] input, and stores two more [5x4] matrices using registers enabled by `shiftEnable`. This design is similar to the tapped delay line used with a Line Buffer using single pixel streaming. The subsystem then accesses pixel data across the columns to form 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.
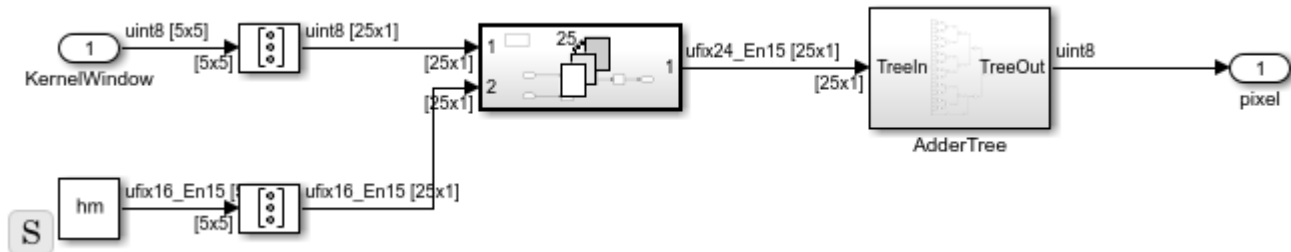
Implement Edge Detectors

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.
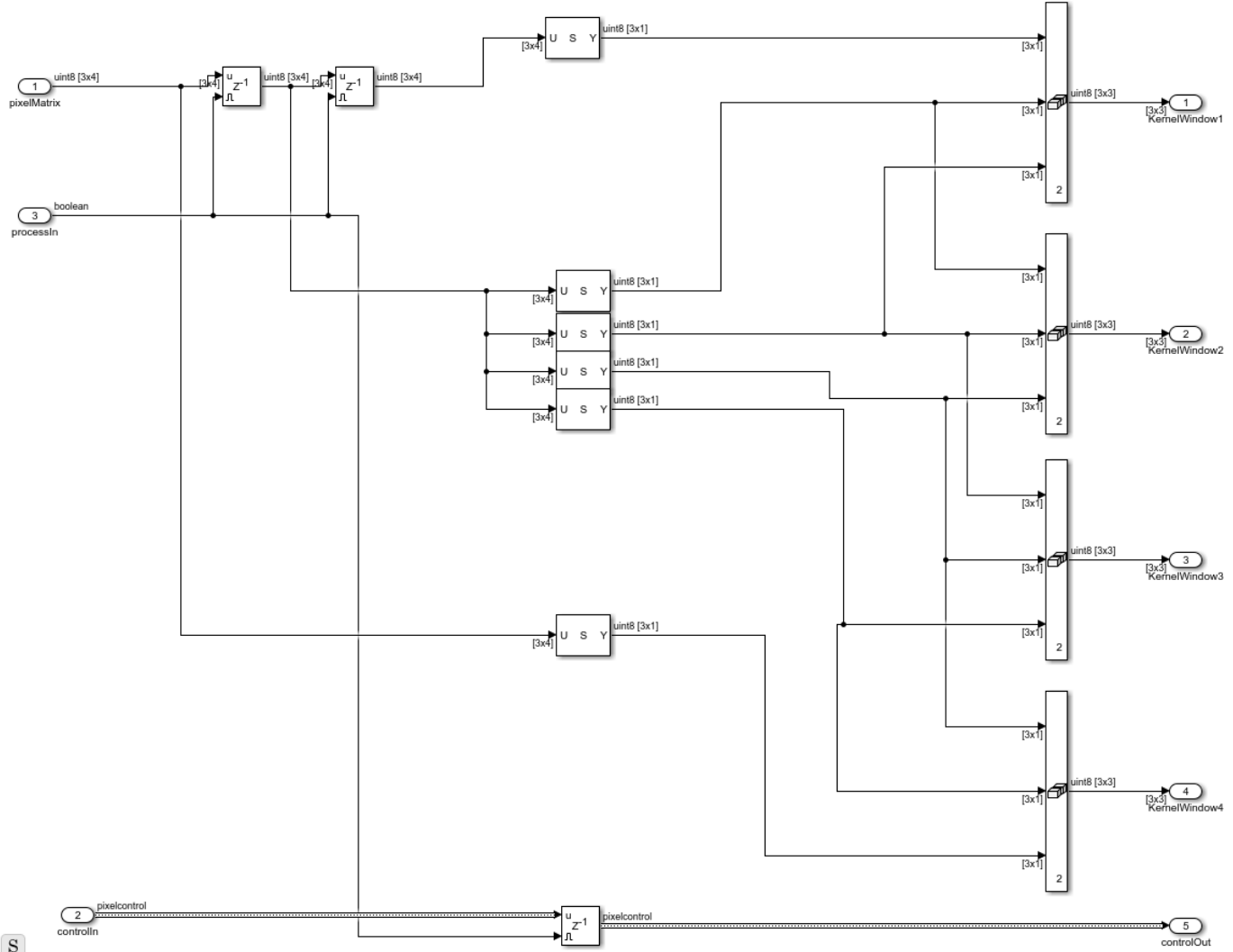
Extending to Larger Kernel Sizes

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.
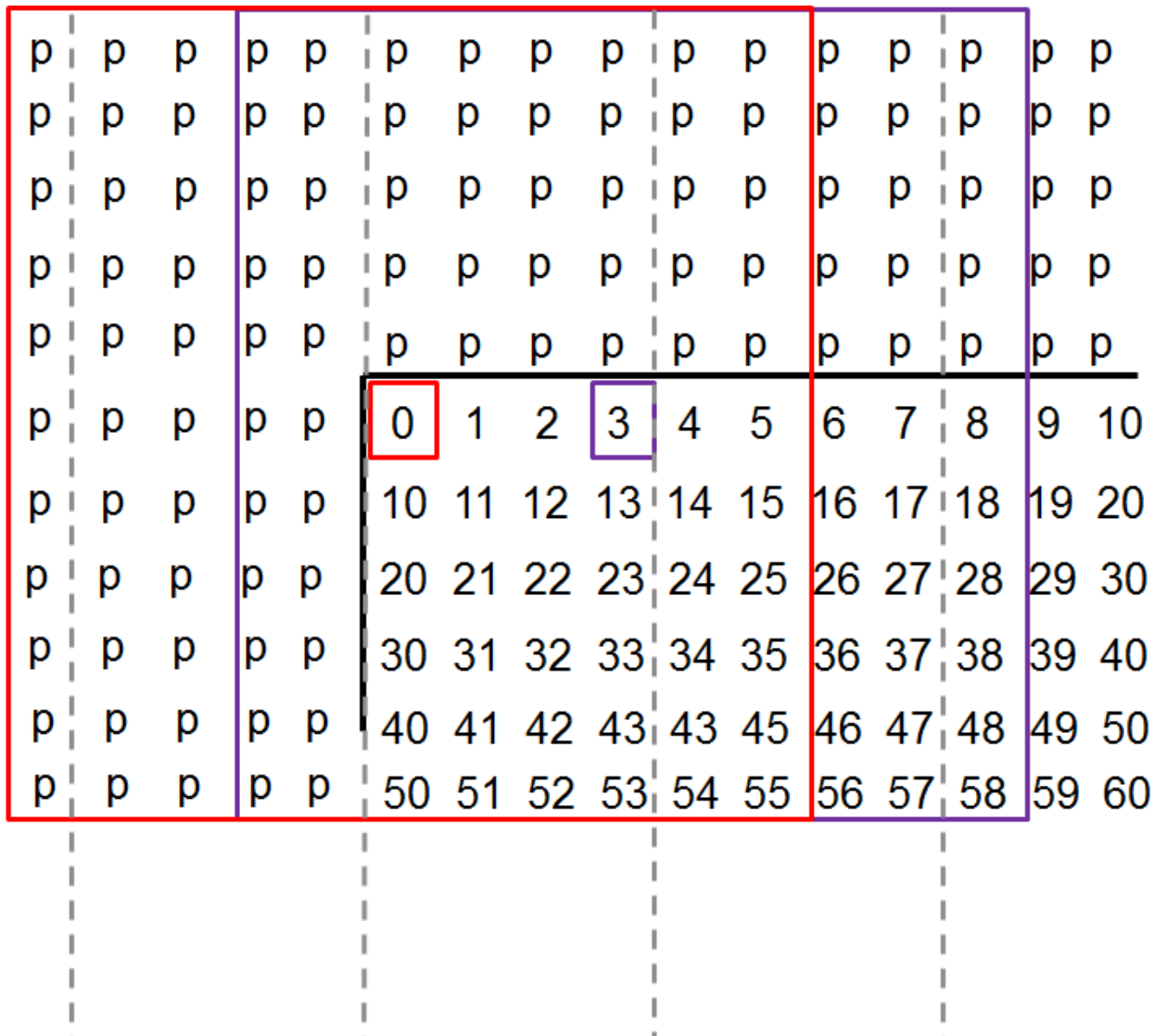
Improving Simulation Time

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

MultiPixel-MultiComponent Video Streaming

This example shows how to work with a multipixel-multicomponent pixel stream. Multipixel-multicomponent streaming enables real-time processing of high-resolution or high-frame-rate color video streams.

To demonstrate working with such a video stream, this example implements the well-known *bloom effect* image post-processing technique. The bloom effect introduces or enhances the glow of light sources in an image.

Top Level I/0

Each pixel of a high-resolution or high-frame-rate pixel stream is modeled as a NumPixel-by-NumComponent matrix. Matrix data types are supported for HDL code generation within a design, but not for the ports of the top-level susbsystem. In this case, the input pixel stream is split into three 4-by-1 vectors at the input of the DUT, and then recombined at the output into a 4-by-3 matrix for the Pixels To Frame block.
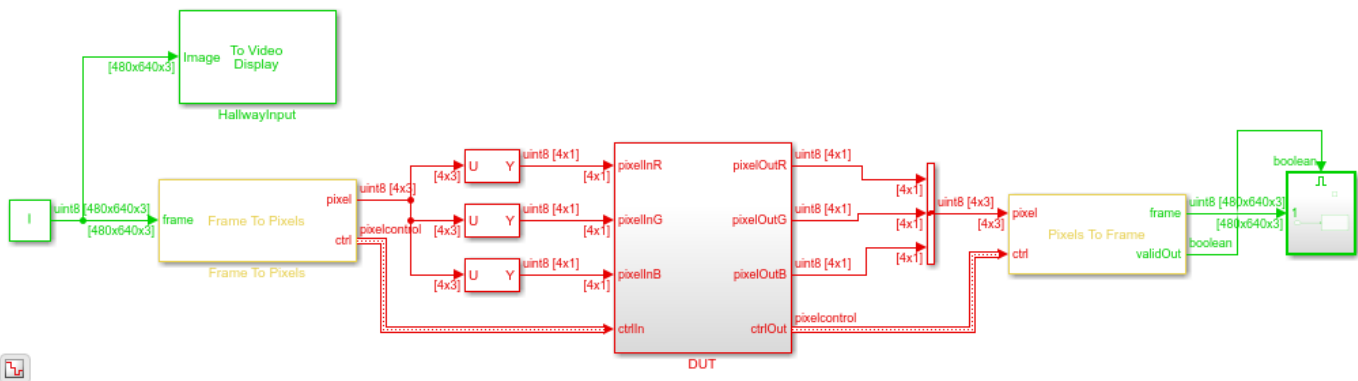
Bloom Effect

The example model follows these three steps to add a bloom effect to the input image.

1 The MultiPixelColorSpaceConverter and BrightSpotFilter subsystems find bright spots in the intensity image by checking pixel values against a threshold.

2 The 15x15 Blur subsystem spreads out the bright spots by applying a Gaussian filter.

3 The BlendBloom subsystem adds the Gaussian-enhanced bright spots back to the original image.

The model uses For Each subsystems for repeated operations in parallel, which results in a cleaner model and reduces the number of HDL files that are generated.
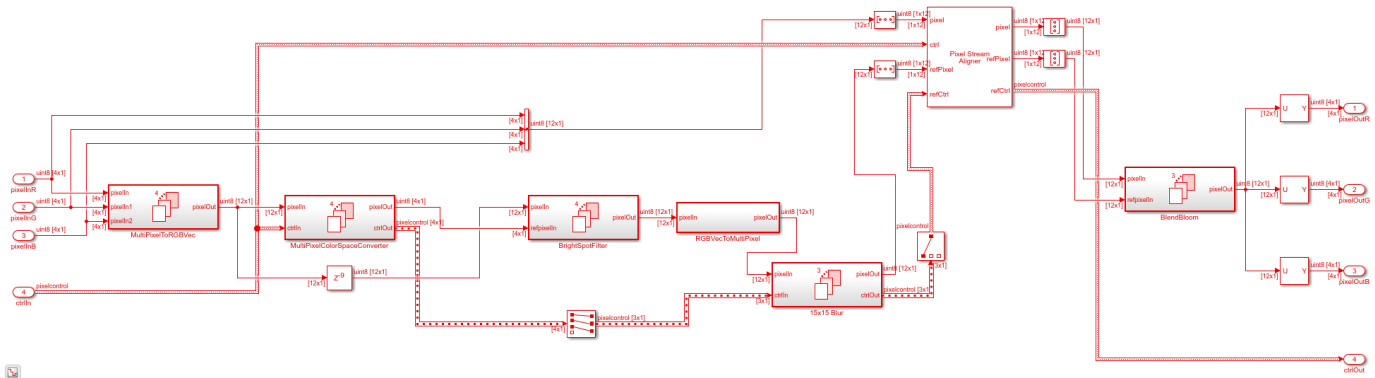
Matrix Flattening

Vision HDL Toolbox™ neighborhood-processing blocks can operate on vector inputs, but do not support matrix inputs. The line buffer used inside the blocks returns a NumPixels-by-KernelHeight matrix. Using multicomponent inputs would result in a NumPixels-by-KernelHeight-by-NumComponents output matrix, however, 3-D matrices are not supported for HDL code generation. For Each subsystems support HDL code generation with scalar and vector inputs but not with matrix inputs. To work around this issue, the model flattens the NumPixels-by-NumComponents matrix into a NumPixels*NumComponents-by-1 vector, and then uses the partition settings on For Each subsystems to repeat the operation across each slice of the vector.

The model encodes the pixel vector in two different ways. The first encoding represents the pixels as four RGB values concatenated together into a 12-by-1 vector, as shown in this diagram. The model sets the **Partition width** parameter of the MultiPixelColorSpaceConverter subsystem to 3. The subsystem implements four RGB-to-Intensity operations in parallel.
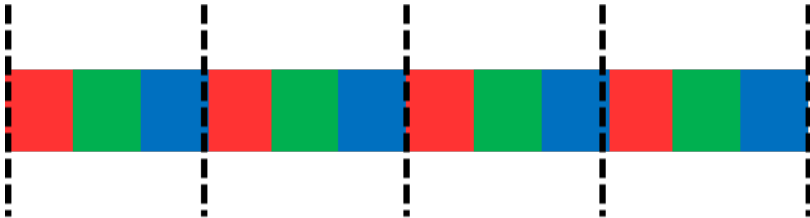
The second pixel encoding concatenates four R, G, or B pixels together, as shown in this diagram. The 15x15 Blur subsystem accepts a 12-by-1 input vector. The model sets the **Partition width** parameter of the 15x15 Blur subsystem to 4, so it operates on three 4-by-1 multipixel vectors. The subsystem implements three Image Filter blocks in parallel. Each filter operates on a 4-by-15 window and returns a 4-by-1 vector.

Simulation Results

Simulating the model displays these input and output images. The bloom effect makes the lighted areas of the scene look brighter and shows a halo effect.
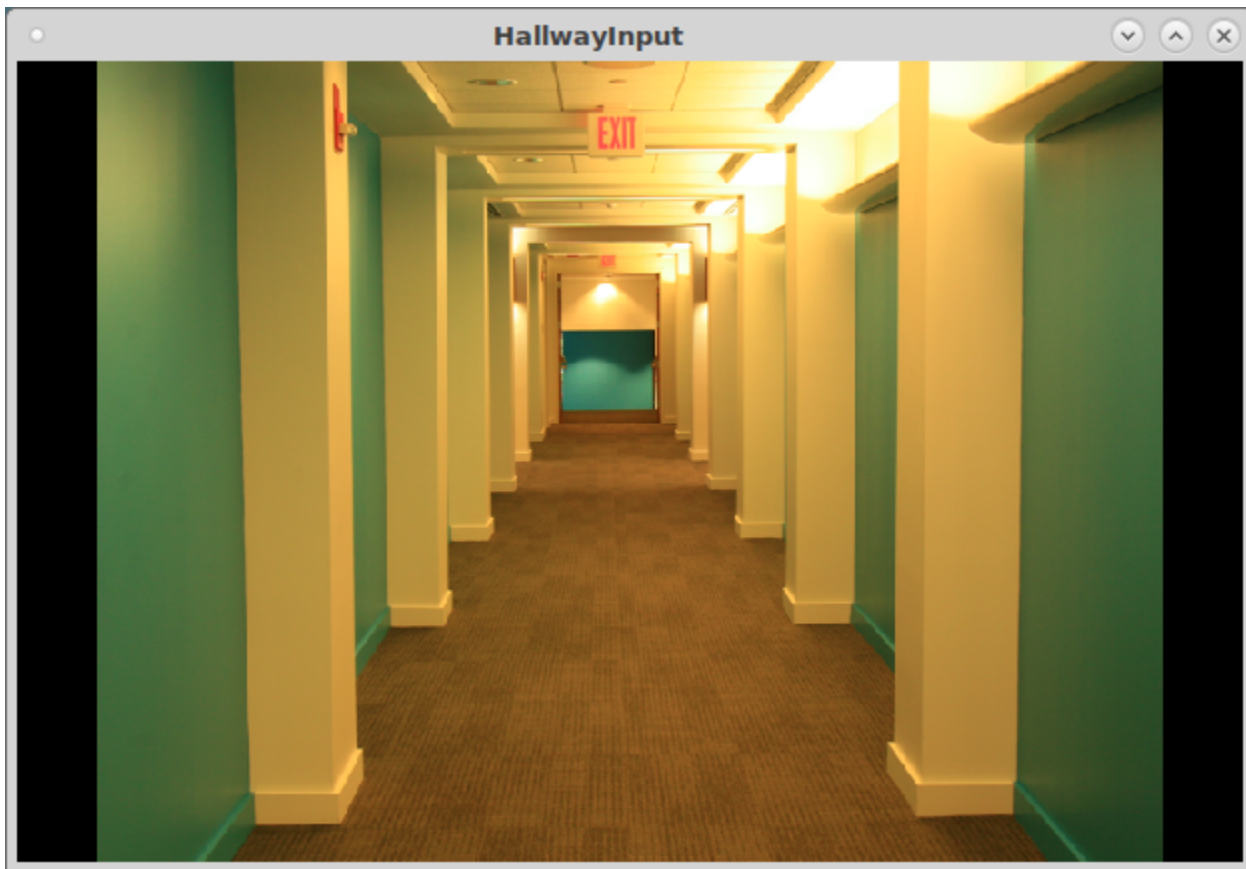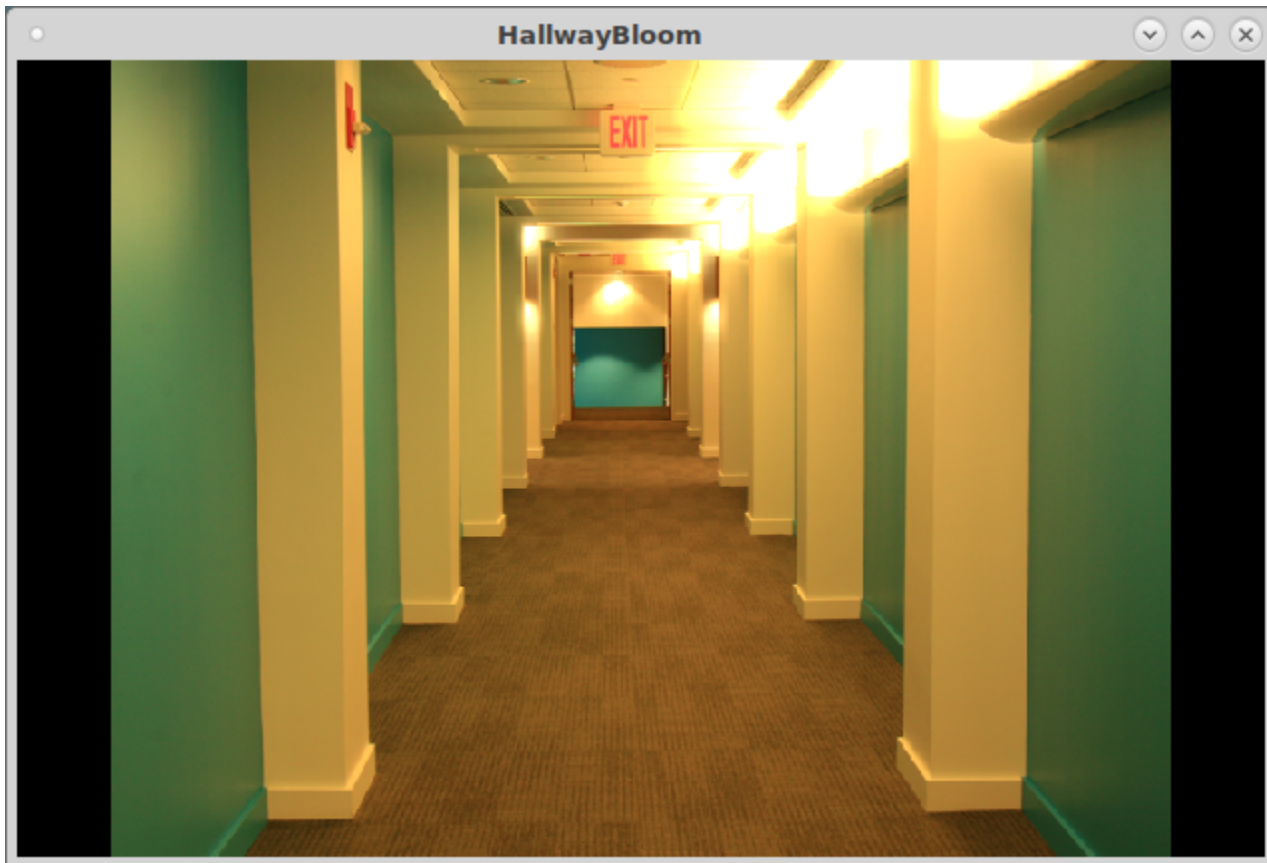
Implementation Results

This table shows the synthesis results of HDL code generated from the DUT subsystem and synthesized for a Xilinx™ Zynq™ ZC706 board. Because none of the resources exceed 25% of their respective category, the design has a relatively small footprint.

```
T =

  4x2 table

    Resource      Usage
    _____      _____

    DSP48         84
    Flip Flop     62724
    LUT           36710
    BRAM          132
```

See Also
Frame To Pixels | 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.
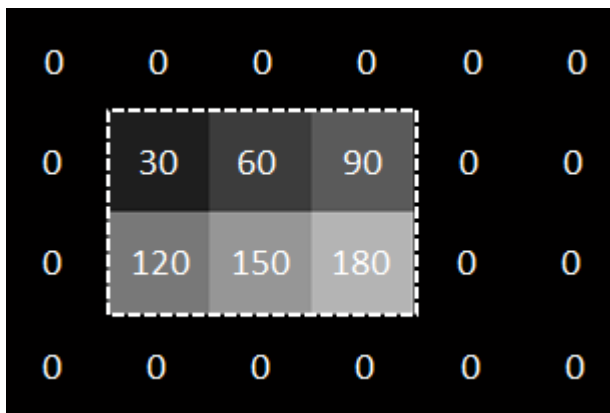

```
F = logical([0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0]);
L = logical([0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0]);
D = logical([0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0]);
pixel = uint8([0,0,0,0,0,0,0,30,60,90,0,0,0,120,150,180,0,0,0,0,0,0,0,0,0]);
```

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.

```matlab
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

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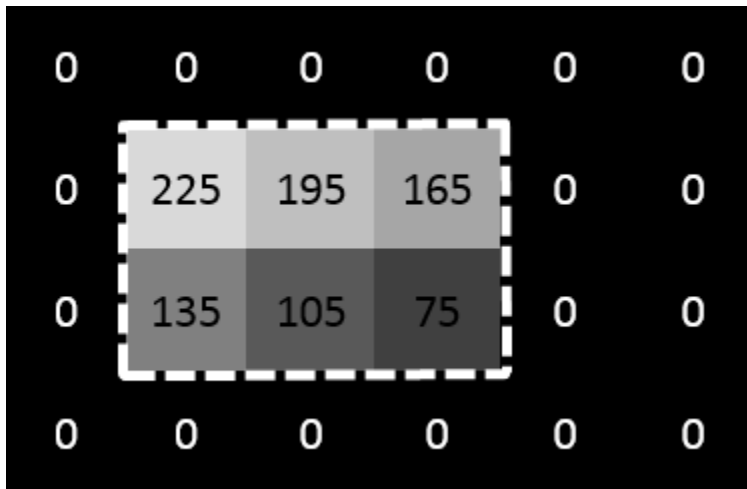
If you have a DSP System Toolbox™ license, you can view the vectors as signals over time using the Logic Analyzer. This waveform shows the `pixelcontrol` and Camera Link control signals, the starting pixel values, and the delayed pixel values after each operation.
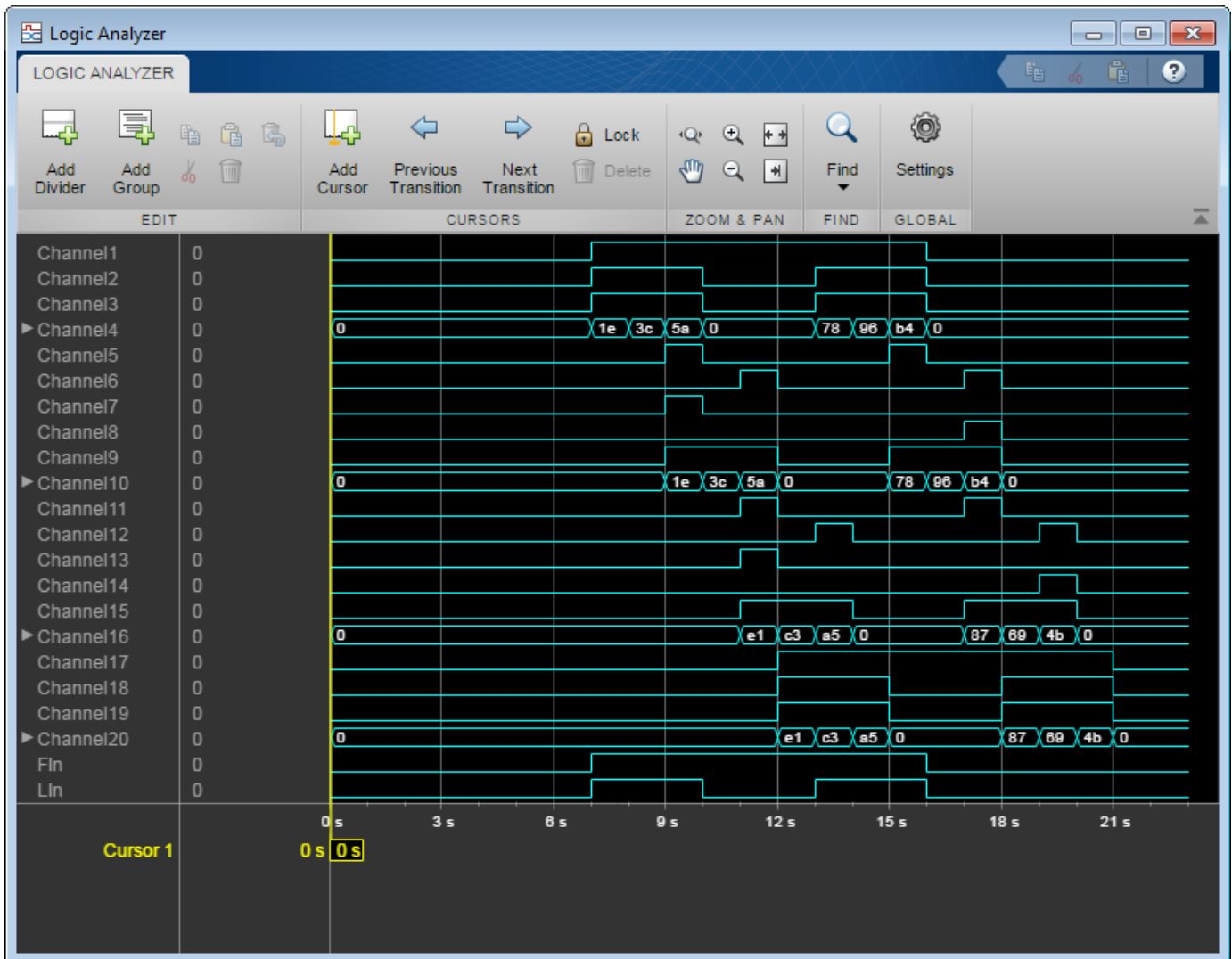
See Also

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.
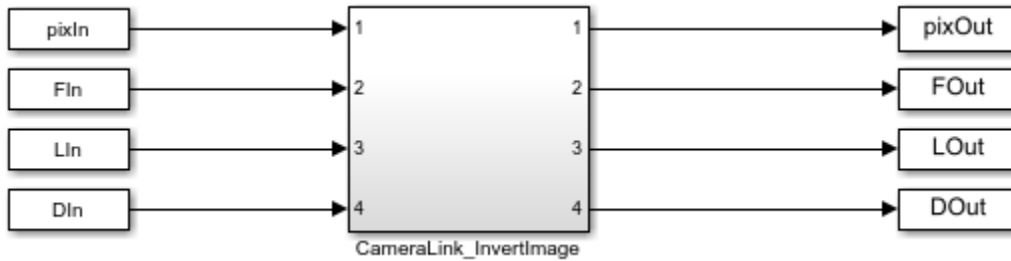
Structure of the Subsystem

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.
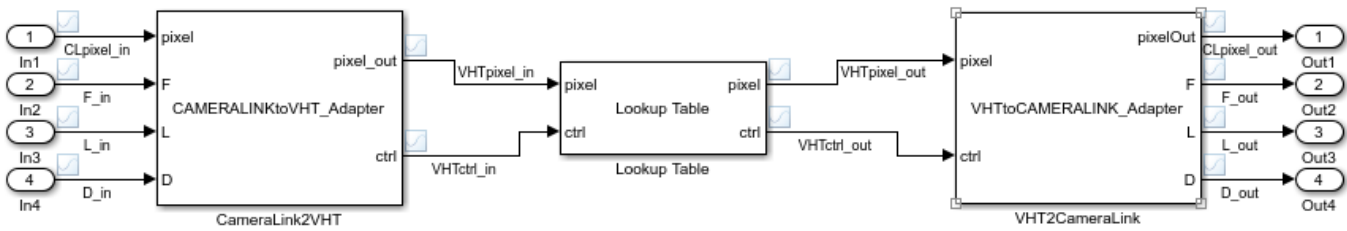
blocks do not need to know the size/format of the frame

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.
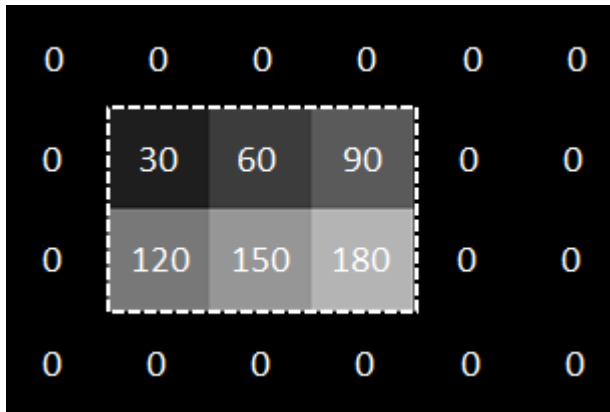

```
FIn = logical([0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0]);
LIn = logical([0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0]);
DIn = logical([0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0]);
pixIn = uint8([0,0,0,0,0,0,0,30,60,90,0,0,0,120,150,180,0,0,0,0,0,0,0,0,0]);
```

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.
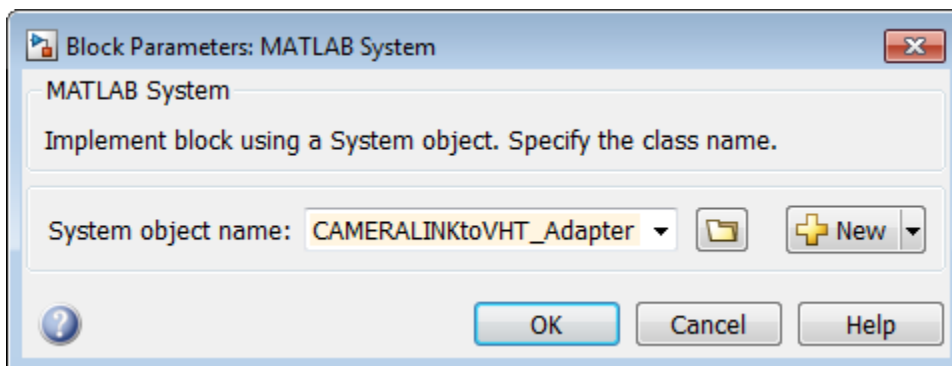
Design Vision HDL Toolbox Algorithm

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.
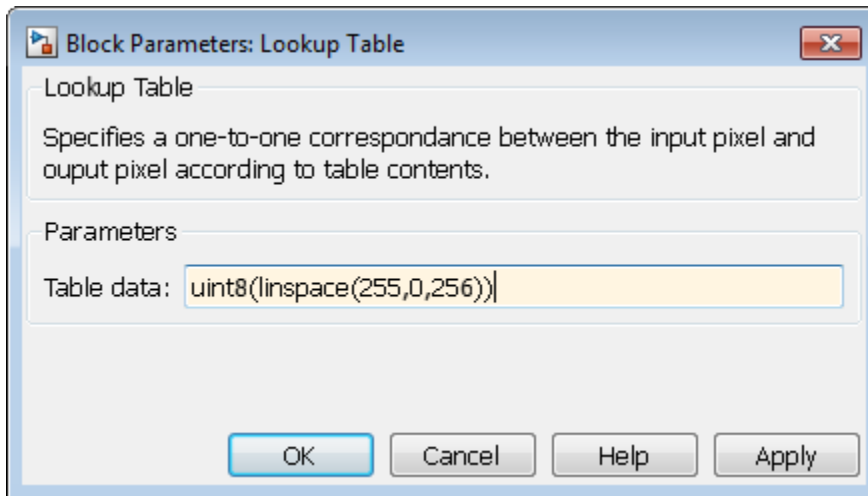
Convert pixelcontrol to Camera Link

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 1-23 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.
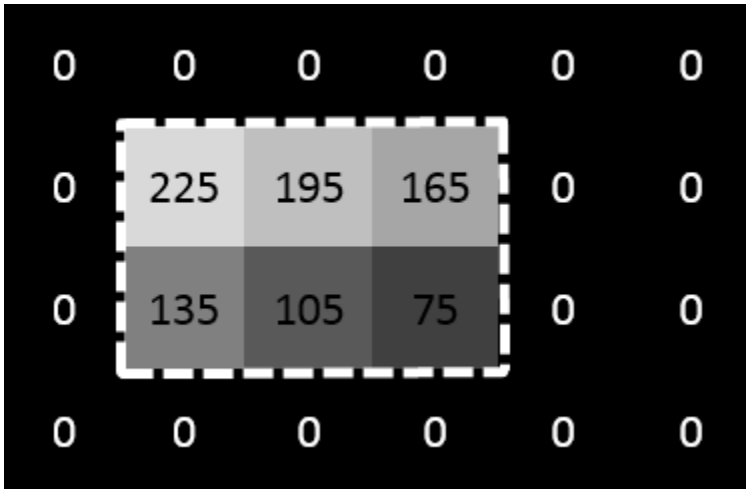
If you have a DSP System Toolbox™ license, you can view the signals over time using the Logic Analyzer. Select all the signals in the `CameraLink_InvertImage` subsystem for streaming, and open the Logic Analyzer. This waveform shows the input and output Camera Link control signals and pixel values at 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.
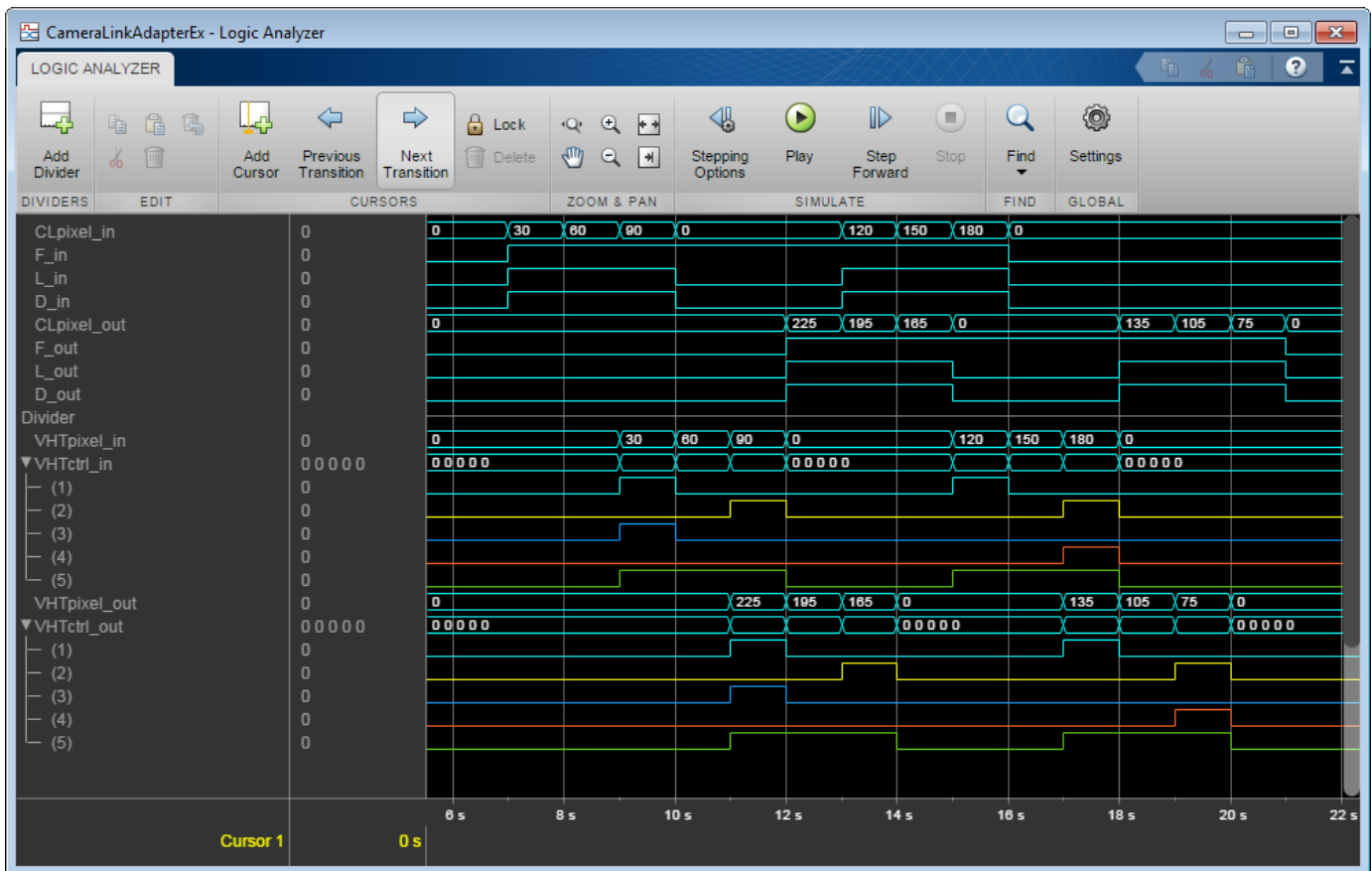
For more info on observing waveforms in Simulink, see "Inspect and Measure Transitions Using the Logic Analyzer" (DSP System Toolbox).

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

HDL-Optimized Algorithm Design

Edge Padding
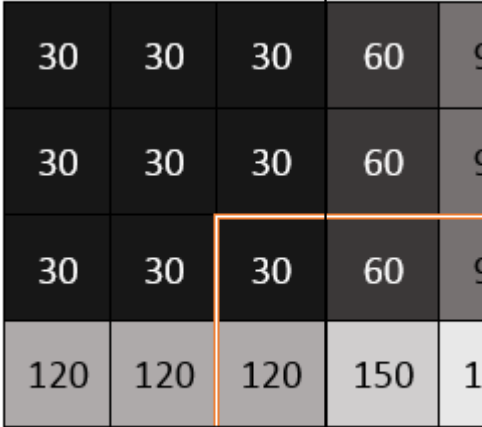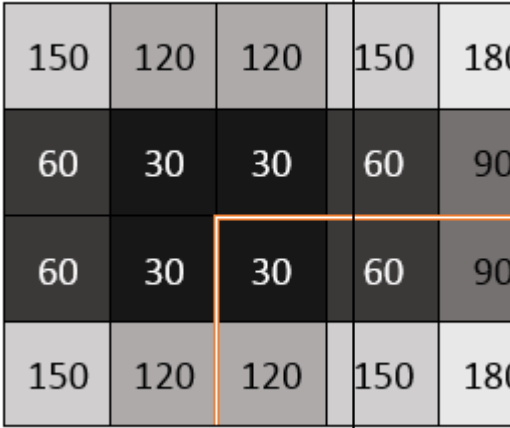
To perform a kernel-based operation such as filtering on a pixel at the edge of a frame, Vision HDL Toolbox algorithms pad the edges of the frame with extra pixels. These padding pixels are used for internal calculation only. The output frame has the same dimensions as the input frame. The padding operation 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 also support opting out of setting the padding pixel values. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame.

The diagrams show the top-left corner of a frame, with padding added to accommodate a 5-by-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.

Type of Padding	Description	Diagram
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. Choosing a small number, such as 16, as a near-black padding value, is common.	In the diagram, C is the constant value assigned to the inactive pixels around the active frame.

Type of Padding	Description	Diagram
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. <table><tr><td>30</td><td>30</td><td>30</td><td>60</td></tr><tr><td>30</td><td>30</td><td>30</td><td>60</td></tr><tr><td>30</td><td>30</td><td>30</td><td>60</td></tr><tr><td>120</td><td>120</td><td>120</td><td>150</td></tr></table>
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. <table><tr><td>150</td><td>120</td><td>120</td><td>150</td><td>18</td></tr><tr><td>60</td><td>30</td><td>30</td><td>60</td><td>90</td></tr><tr><td>60</td><td>30</td><td>30</td><td>60</td><td>90</td></tr><tr><td>150</td><td>120</td><td>120</td><td>150</td><td>18</td></tr></table>

Type of Padding	Description	Diagram
None	This option excludes padding logic. The line buffer does not set the pixels outside the image frame to any particular value. The kernel calculation uses the current value in the line buffer. To maintain pixel stream timing, the output frame is the same size as the input frame. However, to avoid using pixels calculated from undefined padding values, mask off the *KernelSize*/2 pixels around the edge of the frame for downstream operations. Excluding padding can useful for applications that meet any of these conditions. • The output video stream does not need to maintain physical timing. • The resulting image is not displayed. For example, finding the location of objects in an image. • The information of interest is always in the center of the image. For an example, see "Increase Throughput with Padding None" on page 2-5.	The diagram shows the undefined values of the inactive pixels around the active frame.

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 *KernelSize* pixels horizontally and *KernelSize* lines vertically.

When you set the **Padding method** to None, the horizontal blanking period must have at least 6 pixels of front porch and 6 pixels of back porch. For the Median Filter block with the **Padding method** set to None, the horizontal blanking must have at least 10 pixels of front porch and 10 pixels of back porch. The vertical blanking still must be *KernelSize* lines.

See Also
Image Filter | `visionhdl.ImageFilter`

More About
• "Streaming Pixel Interface" on page 1-2

Increase Throughput with Padding None

This example shows how to reduce latency and save hardware resources by not adding padding pixels at the edge of each frame.

Most image filtering operations pad the image to fill in the neighborhoods for pixels at the edge of the image. Padding can help avoid border artifacts in the output image. In a hardware implementation, the padding operation uses extra resources and introduces extra latency.

Vision HDL Toolbox™ blocks that perform neighborhood processing with padding require horizontal blanking that is twice the kernel width. This behavior means that larger filter sizes result in a longer blanking requirement. Excluding the padding by setting the **Padding method** parameter to `None` enables you to use a smaller period of horizontal blanking. Without padding, the horizontal blanking requirement is independent of the image resolution and kernel size.

This example includes two models. The first model shows how to use this option with library blocks, and the second model demonstrates using it when constructing algorithms that use the Line Buffer block. This example also explains some design considerations when you do not use padding.

Use of Library Blocks with Padding None

This example model shows how to use padding none with a predefined algorithm from Vision HDL Toolbox libraries. This model includes an Image Filter block configured for an n-by-n blur filter and with its **Padding method** parameter set to `None`. You can change the size of the filter kernel by changing the value of `n` in the workspace. The model opens with `n` set to 15.

When using edge padding, most blocks have `floor(KernelHeight/2)` lines of latency and require `2*KernelWidth` cycles of horizontal blanking. When you omit padding, most blocks require only 12 cycles of horizontal blanking. Because the internal line buffer latency no longer depends on the kernel size, this blanking interval accommodates any kernel size.

To show the reduced blanking requirements of using **Padding method** set to `None`, the Frame To Pixels block is configured for a custom 240p format that uses only 12 cycles of combined front and back porch.

When you run the model, it shows these three figures.

- Input Video -- Original 240p input video.
- Padding None Full Frame -- Output video from the filter without padding, showing border artifacts.
- Padding None ROI -- Output video from the filter without padding, with border pixels trimmed from the edges of the frame. The frame size is smaller than the size of the input video.
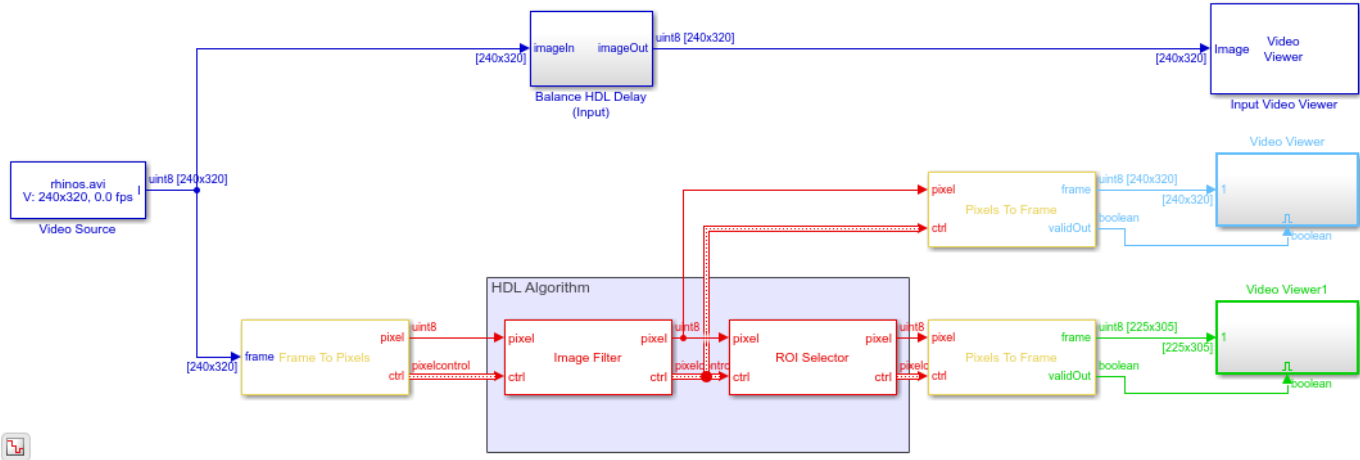
Border Artifacts

In the Padding None Full Frame viewer, shown, a dark border is visible around the edge of each frame. This effect is because, without padding pixels, the filter neighborhoods are not fully defined at the edges of the frame. Output from a filter that has padding pixels does not show any border artifacts because the padding logic ensures that the edge neighborhoods are fully defined.

Removing or masking off these border pixels from nonpadded output before further analysis is common. Border artifacts can decrease the accuracy of subsequent processing. For example, these artifacts can affect the statistical distribution of the overall image. Vision HDL Toolbox blocks return the border pixels for nonpadded images to maintain the input and output timing. The values of these pixels are undefined and cannot be assumed to have any particular relation to the surrounding pixels.

The ROI Selector block removes `floor(KernelHeight/2)` and `floor(KernelWidth/2)` pixels from the edges of each frame. The Padding None ROI viewer, shown, shows the video with the border artifacts removed. The resulting frame for a 15-by-15 kernel is 225-by-305 pixels in size, reduced from 240-by-320 pixels.

Use of Line Buffer Block with Padding None

This model shows how to design algorithms by using a Line Buffer with the **Padding method** parameter set to None. This model contains a Padding None subsystem, and a Padding Symmetric subsystem.

The Frame To Pixels block connected to the Padding Symmetric subsystem uses the standard 240p format. The standard horizontal blanking (combined front and back porch) is 82 cycles. Increasing the resolution increases the blanking interval.For example, the 1080p format has 280 idle cycles between lines.

The Frame To Pixels block connected to the Padding None subsystem implements a custom 240p format that uses only 12 cycles of combined front and back porch, the same as in the Image Filter model shown earlier.

This model implements a 15-by-15 Gaussian filter, with a large standard deviation, by using the Line Buffer block.

When you run the model, it shows three figures:

- Input Video -- Original 240p input video.
- Padding None ROI -- Output video from the filter without padding, with border pixels trimmed from the edges of the frame. The frame size is smaller than the size of the input video.
- Padding Symmetric -- Output video from the filter with symmetric padding. This video is full size but has no edge effects because the padding bits define the neighborhoods around the edge pixels.
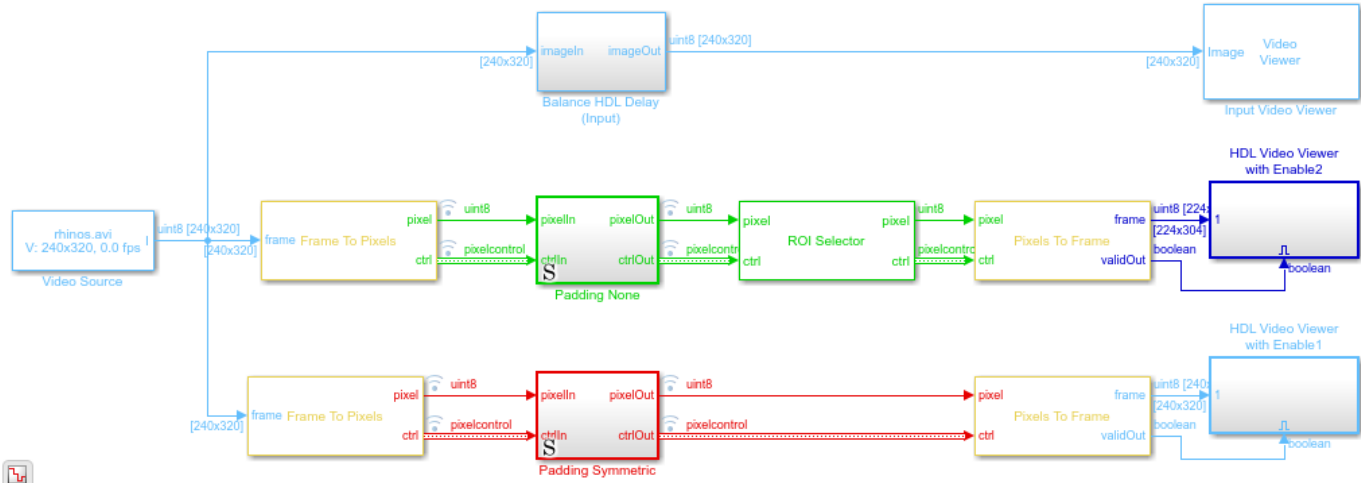
`pixelcontrol` Delay Balancing

When you construct algorithms that use the Line Buffer block, you must delay-balance the `pixelcontrol` bus to account for the kernel latency. When you use padding, the Line Buffer returns **shiftEnable** set to 1 for `floor(KernelWidth/2)` cycles before **hStart** and after **hEnd**. The delay-balancing logic uses this extended **shiftEnable** signal to control the delay registers for the `pixelcontrol` signals. You can see this logic in the Padding Symmetric/pixelctrldelay subsystem.

When you set **Padding method** to `None`, the Line Buffer returns **shiftEnable** to 1 between **hStart** and **hEnd**. The delay-balancing logic must use the clock, instead of **shiftEnable**, to control the delay registers for **hEnd**, **vEnd**, and **valid**. The **valid** signal must also respond to **shiftEnable** being set to 0 during a line, which can occur when interfacing with external memory. The **valid** signal must also be set to 1 on the last pixel of the line, to match with with **hEnd** and **vEnd**. To meet both requirements, the delay-balancing logic delays the **valid** signal by using a register enabled by **shiftEnable**, and uses a Unit Delay Enabled block to set the **valid** signal to 1 with **hEnd** at the end of the line. The Padding None/pixelctrldelay subsystem shows this logic.
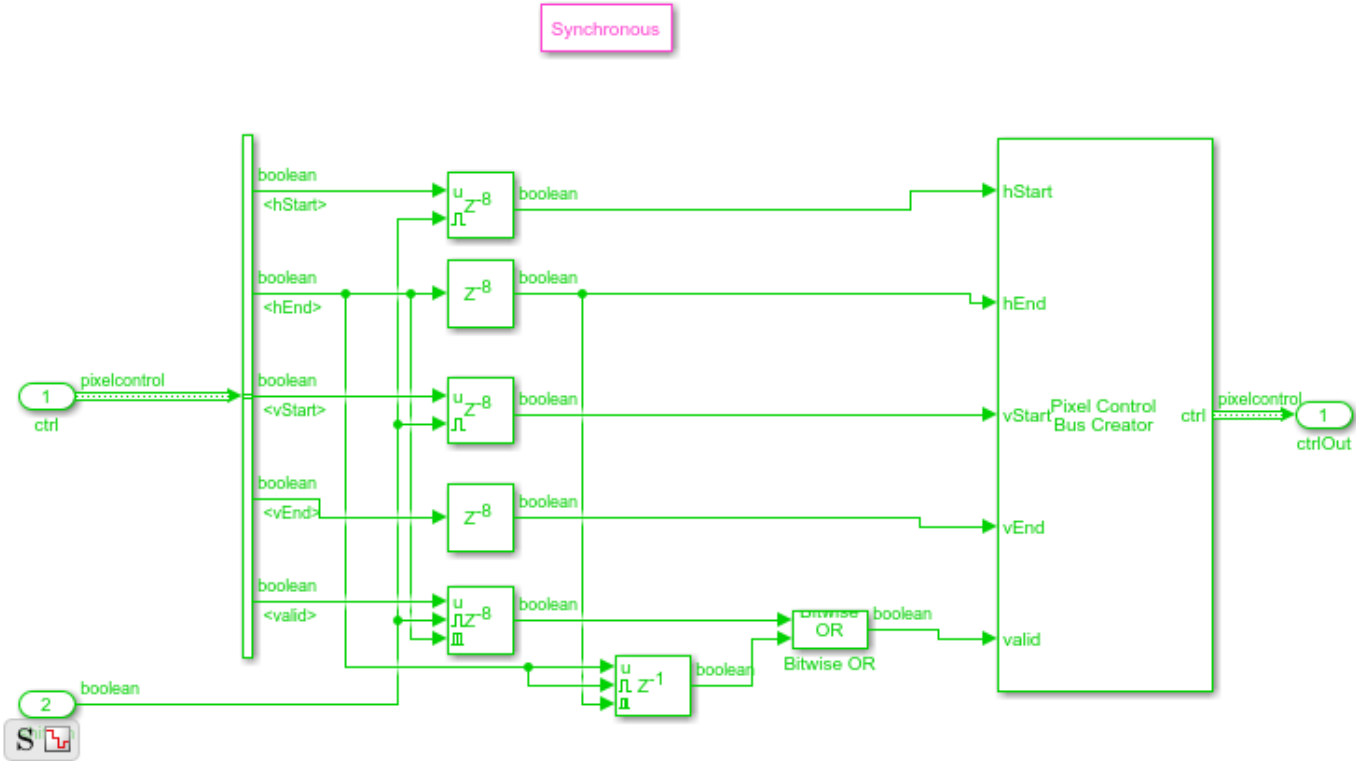
Conclusion

Excluding padding logic enables you to achieve higher throughput by using a video format with reduced horizontal blanking. This option also reduces hardware resource usage. However, your design must account for the border artifacts later in the processing chain. When you use the Line Buffer block, you must delay the `pixelcontrol` bus to match the kernel latency by using control logic that accounts for the modified behavior of the **shiftEnable** output signal. Using this example as a starting point, you can design algorithms and systems that achieve higher throughput by excluding padding logic.

Gamma Correction

This example shows how to model pixel-streaming gamma correction for hardware designs. The model compares the results from the Vision HDL Toolbox™ Gamma Corrector block with the results generated by the full-frame Gamma block from Computer Vision System Toolbox™.

This example model provides a hardware-compatible algorithm. You can implement this algorithm on a board using a Xilinx™ Zynq™ reference design. See "Gamma Correction with Zynq-Based Hardware" (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

Structure of the Example

The Computer Vision System Toolbox product models at a high level of abstraction. The blocks and objects perform full-frame processing, operating on one image frame at a time. However, FPGA or ASIC systems perform pixel-stream processing, operating on one image pixel at a time. This example simulates full-frame and pixel-streaming algorithms in the same model.

The GammaCorrectionHDL.slx system is shown below.
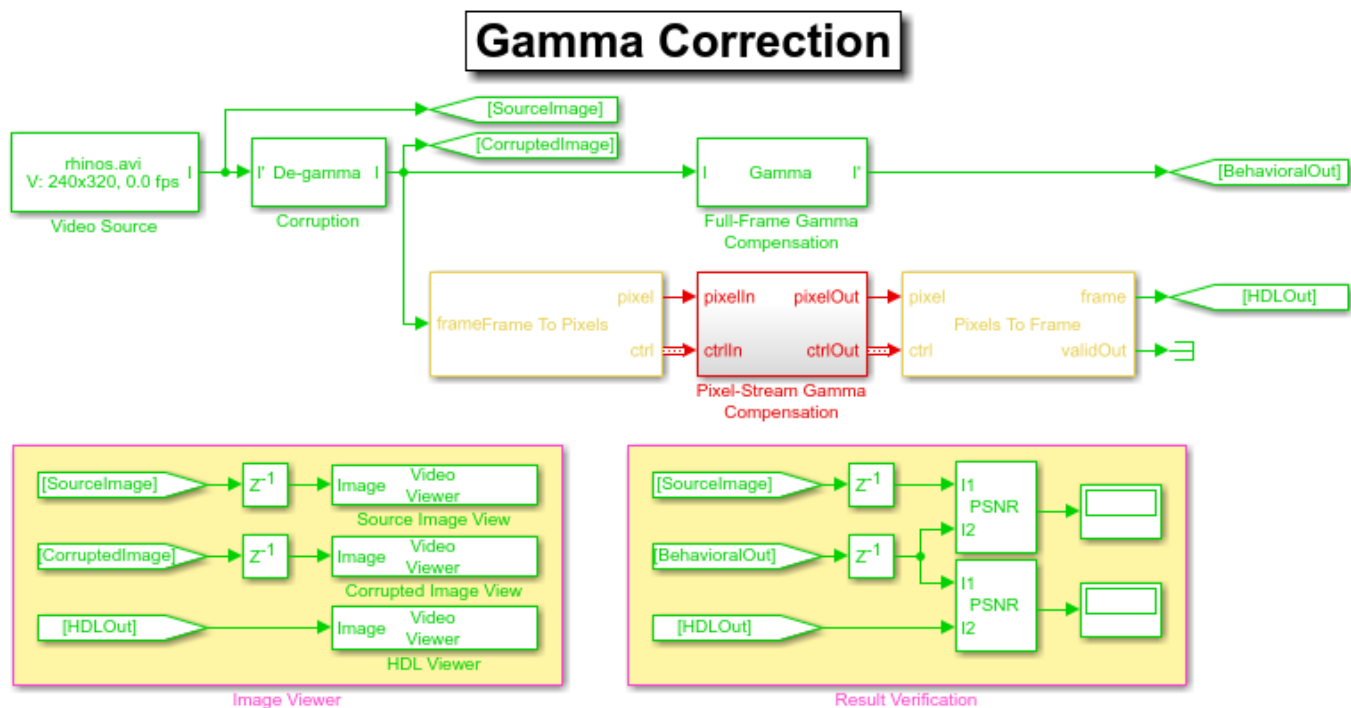
The difference in the color of the lines feeding the **Full-Frame Gamma Compensation** and **Pixel-Stream Gamma Compensation** subsystems indicates the change in the image rate on the streaming branch of the model. This rate transition is because the pixel stream is sent out in the same amount of time as the full video frames and therefore it is transmitted at a higher rate.

In this example, the Gamma correction is used to correct dark images. Darker images are generated by feeding the **Video Source** to the **Corruption** block. The **Video Source** outputs a 240p grayscale video, and the **Corruption** block applies a De-gamma operation to make the source video perceptually darker. Then, the downstream **Full-Frame Gamma Compensation** block or **Pixel-**

Stream Gamma Compensation subsystem removes the previous De-gamma operation from the corrupted video to recover the source video.

One frame of the source video, its corrupted version, and recovered version, are shown from left to right in the diagram below.
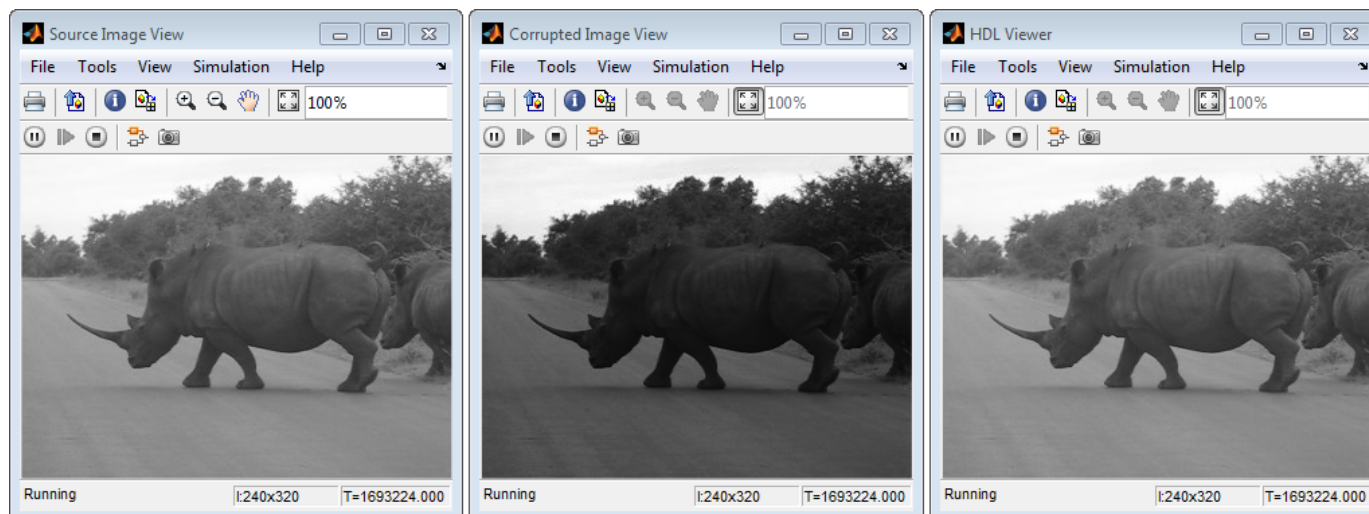
It is a good practice to develop a behavioral system using blocks that process full image frames, the **Full-Frame Gamma Compensation** block in this example, before moving forward to working on an FPGA-targeting design. Such a behavioral model helps verify the video processing design. Later on, it can serve as a reference for verifying the implementation of the algorithm targeted to an FPGA. Specifically, the lower **PSNR** (peak signal-to-noise ratio) block in the **Result Verification** section at the top level of the model compares the results from full-frame processing with those from pixel-stream processing.

Frame To Pixels: Generating a Pixel Stream

The task of the **Frame To Pixels** is to convert a full-frame image to 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:

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

In this example, the Active Video region corresponds to the 240x320 matrix of the dark image from the upstream **Corruption** block. 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** is determined by the product of **Total pixels per line** and **Total video lines**.

Gamma Correction

As shown in the diagram below, the **Pixel-Stream Gamma Compensation** subsystem contains only a **Gamma Corrector** block.
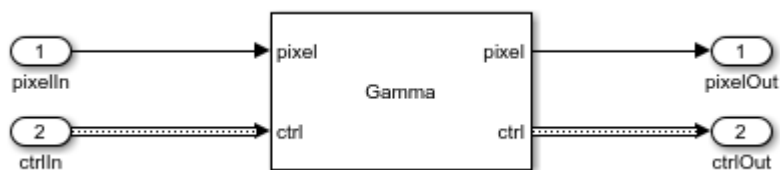
The **Gamma Corrector** block accepts the pixel stream, as well as a bus containing five synchronization signals, from the **Frame To Pixels** block. It passes the same set of signals to the downstream **Pixels To Frame** block. Such signal bundle and maintenance are necessary for pixel-stream processing.

Pixels To Frame: Converting Pixel Stream Back to Full Frame

As a companion to **Frame To Pixels** that converts a full image frame to pixel stream, the **Pixels To Frame** block, reversely, converts the pixel stream back to the full frame by making use of the synchronization signals. Since the output of the **Pixels To Frame** block is a 2-D matrix of a full image, there is no need to further carry on the bus containing five synchronization signals.

The **Number of components** field and the **Video format** fields of both Frame To Pixels and Pixels To Frame are set at 1 and 240p, respectively, to match the format of the video source.

Image Viewer and Result Verification

When you run the simulation, three images will be displayed (refer to the images shown in the "Structure of the Example" Section):

- The source image given by the **Image Source** subsystem
- The dark image produced by the **Corruption** block
- The HDL output generated by the **Pixel-Stream gamma Compensation** subsystem

The presence of the four **Unit Delay** blocks on top level of the model is to time-align the 2-D matrices for a fair comparison.

While building the streaming portion of the design, the **PSNR** block continuously verifies the **HDLOut** results against the original full-frame design **BehavioralOut**. During the course of the simulation, this **PSNR** block should give **inf** output, indicating that the output image from the **Full-Frame Gamma Compensation** matches the image generated from the stream processing **Pixel-Stream Gamma Compensation** model.

Exploring the Example

The example allows you to experiment with different Gamma values to examine their effect on the Gamma and De-gamma operation. Specifically, a workspace variable $gammaValue$ with an initial value 2.2 is created upon opening the model. You can modify its value using the MATLAB command line as follows:

```
gammaValue=4
```

The updated $gammaValue$ will be propagated to the **Gamma** field of the **Corruption** block, the **Full-Frame Gamma Compensation** block, and the **Gamma Corrector** block inside **Pixel-Stream Gamma Compensation** subsystem. Closing the model clears $gammaValue$ from your workspace.

Although Gamma operation is conceptually the inverse of De-gamma, feeding an image to Gamma followed by a De-gamma (or De-gamma first then Gamma) does not necessarily perfectly restore the original image. Distortions are expected. To measure this, in our example, another **PSNR** block is placed between the **SourceImage** and **BehavioralOut**. The higher the PSNR, the less distortion has been introduced. Ideally, if HDL output and the source image are identical, PSNR outputs **inf**. In our example, this happens only when $gammaValue$ equals 1 (i.e., both Gamma and De-gamma blocks pass the source image through).

We can also use Gamma to corrupt a source image by making it brighter, followed by a De-gamma correction for image recovery.

Generate HDL Code and Verify Its Behavior

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('GammaCorrectionHDL/Pixel-Stream Gamma Compensation')
```

To infer a RAM to implement a lookup table used in the **Gamma Corrector**, the LUTRegisterResetType property is set to none. To access this property, right click the **Gamma Corrector** block inside **Pixel-Stream Gamma Compensation**, and navigate to HDL Coder -> HDL Block Properties ...

To generate test bench, use the following command:

```
makehdltb('GammaCorrectionHDL/Pixel-Stream Gamma Compensation')
```

Histogram Equalization

This example shows how to use the Vision HDL Toolbox Histogram library block to implement histogram equalization.

This example model provides a hardware-compatible algorithm. You can generate HDL code from this algorithm, and implement it on a board using a Xilinx™ Zynq™ reference design. See "Histogram Equalization with Zynq-Based Hardware" (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

Introduction

The model shows how to use the Histogram library block to enhance the contrast of images by applying the histogram equalization. To learn more, refer to the Histogram block reference page. There are three components in this histogram equalization example.

- **Video Partition** partitions a big image into four non-overlapping small images for parallel histogram computation.
- **HDLHistogram** computes the accumulated histogram of the image.
- **Equalization** applies the equalized histogram to the original image and generates the contrast-enhanced image.

Copyright 2015 The MathWorks, Inc.
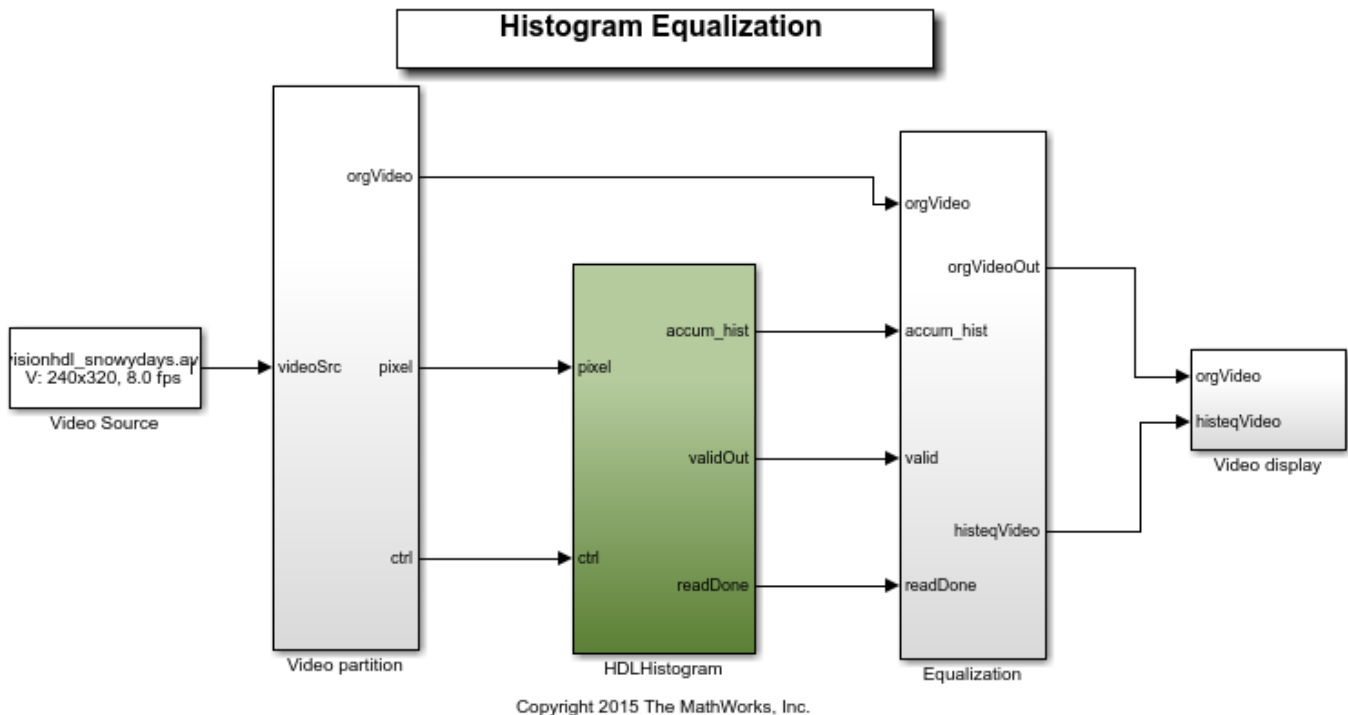
Video Partition

There are use cases where histogram is computed over an entire image, or over small regions-of-interest representing sections of the image. Computing histogram of a big image is time consuming. The video partition component in this example divides a big image into four non-overlapping small images. Histogram is computed over the four small images simultaneously. Each input frame is

partitioned into four 120 by 160 small images. Each small image is connected to a Frame To Pixels block to generate pixel streams and corresponding control signals.
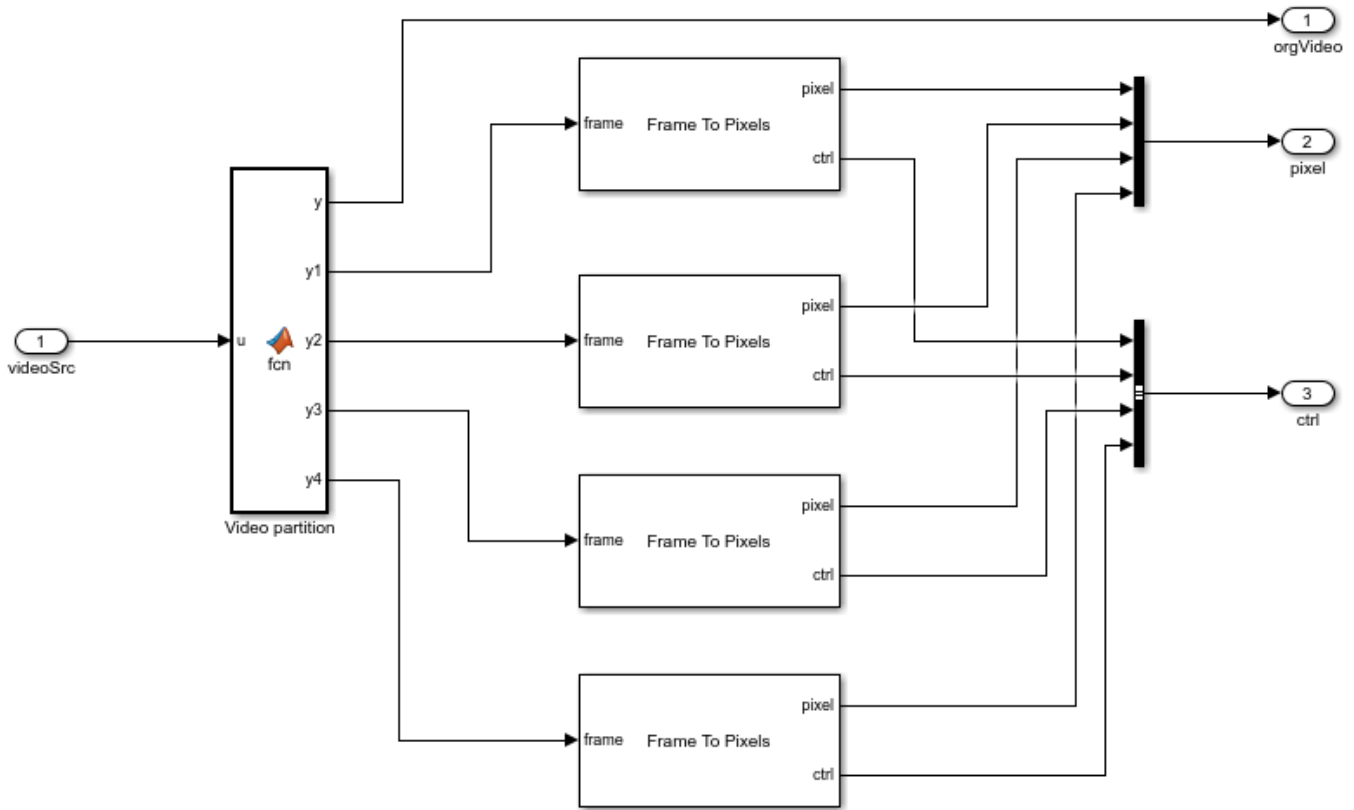
HDLHistogram

HDLHistogram subsystem is optimized for HDL code generation. The histogram of the pixel streams is computed using the Vision HDL Toolbox Histogram library block. Because the input image is grey scale with data type uint8, the input pixels are grouped into 256 bins. The model reads the calculated histogram bins sequentially once the block asserts the *readRdy* signal. The bin values are sent for cumulative histogram calculation. After all 256 bin values are read, the model asserts *binReset* to reset all bins to zero. The collected histogram of each small image is then added together to compute the accumulated histogram of the big image.
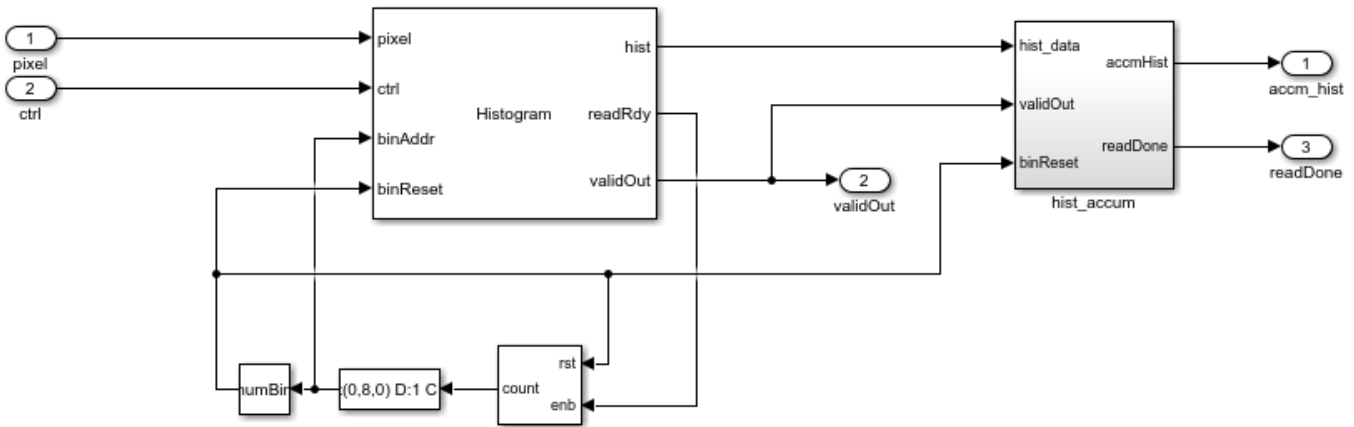
The timing diagram of reading and resetting the histogram bins is shown in the following figure.
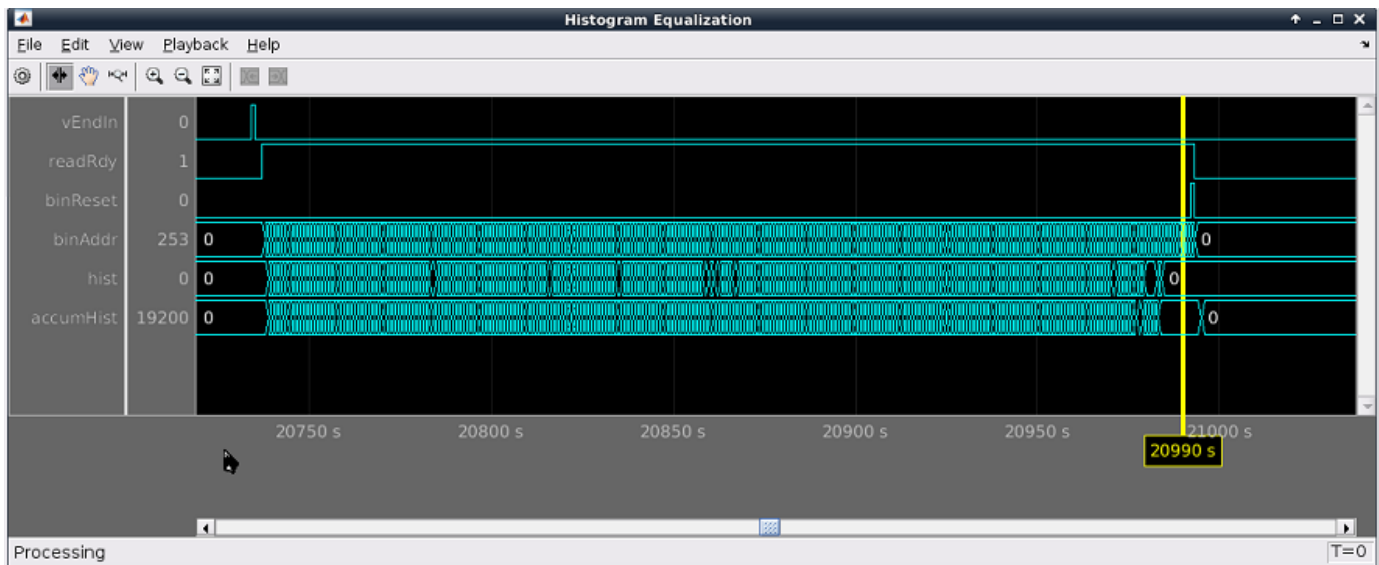
Equalization

Histogram equalization can be applied to the current frame where the accumulated histogram was calculated, or the frame after. If applying to the current frame, the input video needs to be stored. This example delays the input video by one frame and performs uniform equalization to the original video. The equalized video is then compared with the original video.
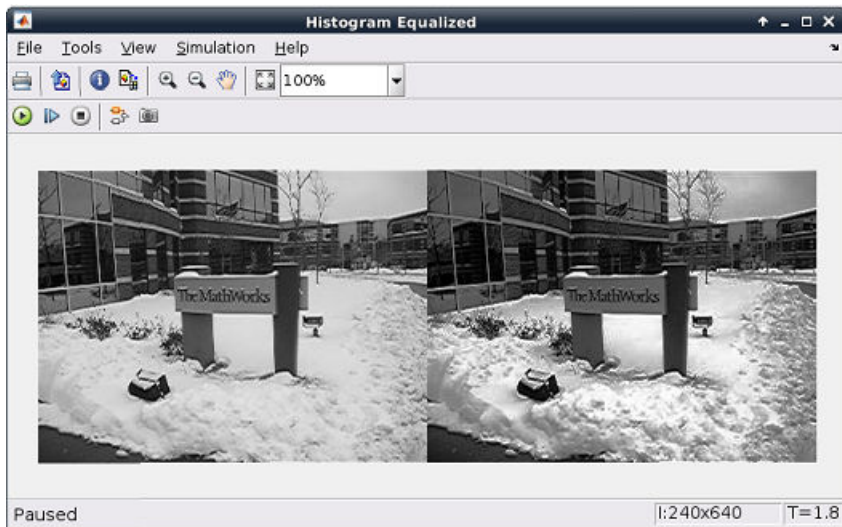
HDL Code Generation

The HDL code generated from the Histogram was synthesized using Xilinx ISE on a Virtex6 (XC6VLX240T-1FFG1156) FPGA, and the circuit ran at about 190 MHz, which is sufficient to process the data in real time.

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

You can use the commands

```
makehdl('HistogramEqualizationHDL/HDLHistogram')
```

or

```
makehdltb('HistogramEqualizationHDL/HDLHistogram')
```

to generate HDL code and test bench for the HDLHistogram subsystem. **Note:** Test bench generation takes a long time due to the large data size. Consider reducing the simulation time before generating the test bench.

Edge Detection and Image Overlay

This example demonstrates how to detect and highlight object edges in a video stream. The behavior of the pixel-stream Sobel Edge Detector, video stream alignment, and overlay, is verified by comparing the results with the same algorithm calculated by the full-frame blocks from the Computer Vision Toolbox™.

This example model provides a hardware-compatible algorithm. You can implement this algorithm on a board using a Xilinx™ Zynq™ reference design. See "Developing Vision Algorithms for Zynq-Based Hardware" (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

Structure of the Example

The EdgeDetectionAndOverlayHDL.slx system is shown below.
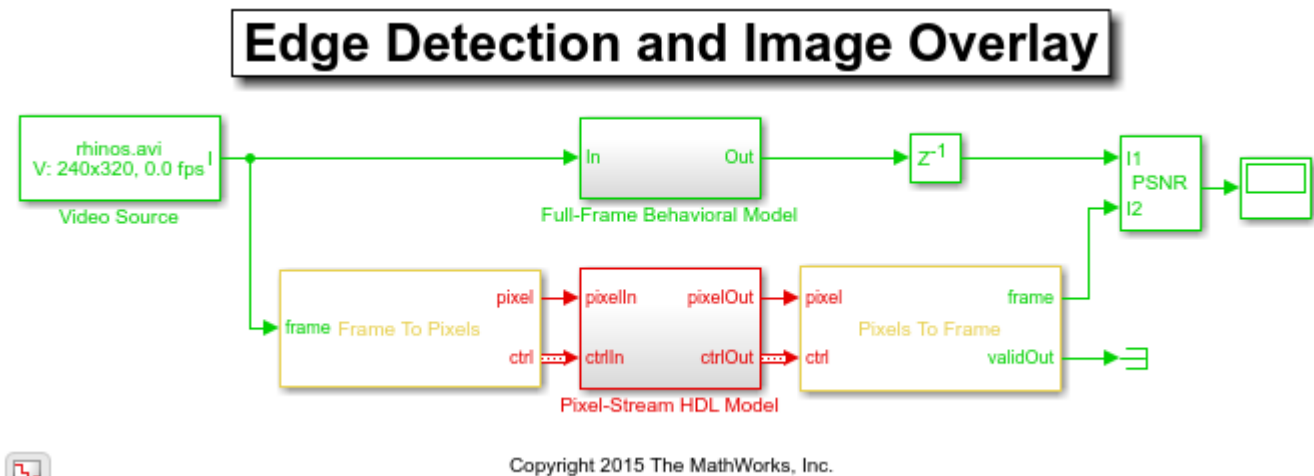
The difference in the color of the lines feeding the **Full-Frame Behavioral Model** and **Pixel-Stream HDL Model** subsystems indicates the change in the image rate on the streaming branch of the model. This rate transition is because the pixel stream is sent out in the same amount of time as the full video frames and therefore it is transmitted at a higher rate.

Full-Frame Behavioral Model

The following diagram shows the structure of the **Full-Frame Behavioral Model** subsystem, which employs the frame-based **Edge Detection** block.
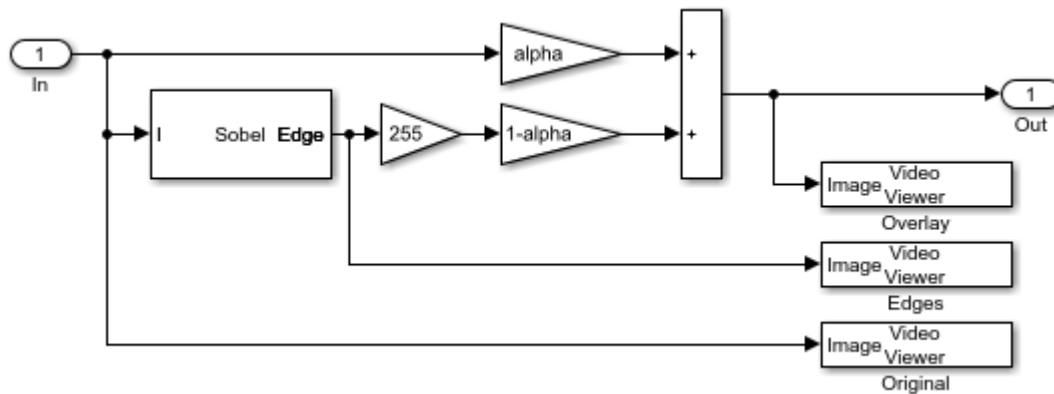
Given that the frame-based **Edge Detection** block does not introduce latency, image overlay is performed by weighting the source image and the **Edge Detection** output image, and adding them together in a straightforward manner.

One frame of the source video, the edge detection result, and the overlaid image are shown from left to right in the diagram below.
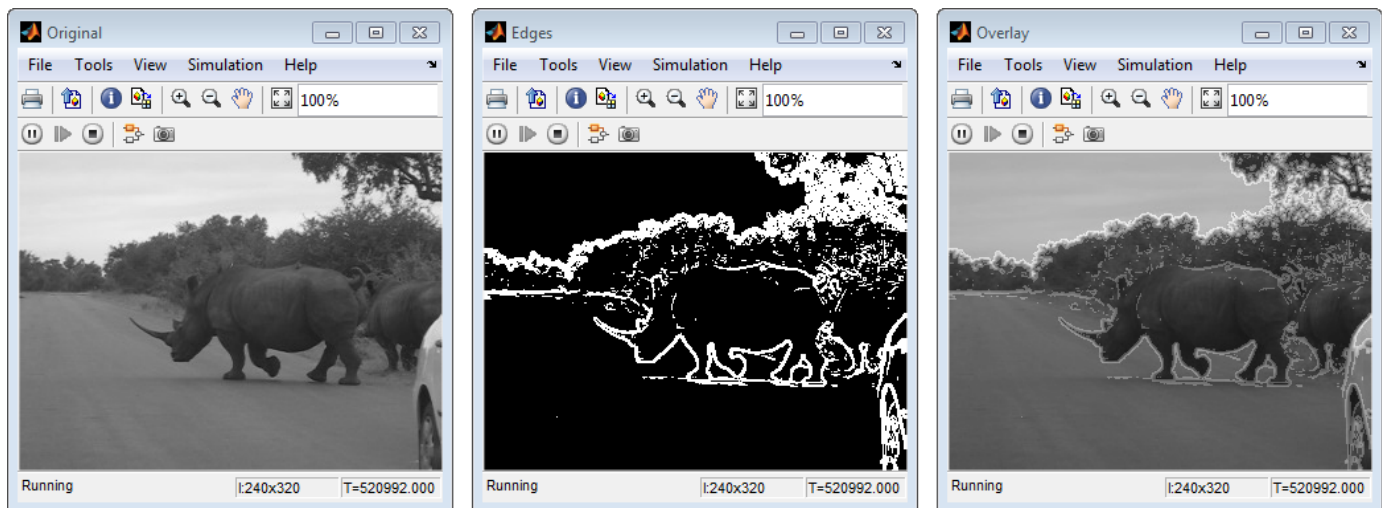
It is a good practice to develop a behavioral system using blocks that process full image frames, the **Full-Frame Behavioral Model** subsystem in this example, before moving forward to working on an FPGA-targeting design. Such a behavioral model helps verify the video processing design. Later on, it can serve as a reference for verifying the implementation of the algorithm targeted to an FPGA. Specifically, the **PSNR** (peak signal-to-noise ratio) block at the top level of the model compares the results from full-frame processing with those from pixel-stream processing.

Frame To Pixels: Generating a Pixel Stream

The task of the **Frame To Pixels** is to convert a full frame image to 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** field is set to 1 for grayscale image input, and the **Video format** field is 240p to match that of the video source.

In this example, the Active Video region corresponds to the 240x320 matrix of the dark image from the upstream **Corruption** block. 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** is determined by the product of **Total pixels per line** and **Total video lines**.

Pixel-Stream Edge Detection and Image Overlay

The **Pixel-Stream HDL Model** subsystem is shown in the diagram below. You can generate HDL code from this subsystem.
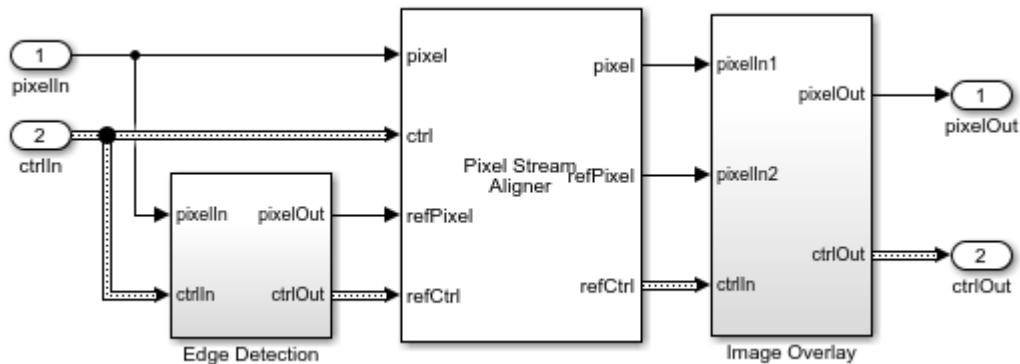
Due to the nature of pixel-stream processing, unlike the **Edge Detection** block in the **Full-Frame Behavioral Model**, the **Edge Detector** block from the Vision HDL Toolbox™ will introduce latency. The latency prevents us from directly weighting and adding two images to obtain the overlaid image. To address this issue, the **Pixel Stream Aligner** block is used to synchronize the two pixel streams before the sum.

To properly use this block, refPixel and refCtrl must be connected to the pixel and control bus that are associated with a delayed pixel stream. In our example, due to the latency introduced by the **Edge Detector**, the pixel stream coming out of the **Edge Detector** is delayed with respect to that feeding into it. Therefore, the upstream source of refPixel and refCtrl are the Edge and ctrl output of the **Edge Detector**.

Pixels To Frame: Converting Pixel Stream Back to Full Frame

As a companion to **Frame To Pixels** that converts a full image frame to pixel stream, the **Pixels To Frame** block, reversely, converts the pixel stream back to the full frame by making use of the synchronization signals. Since the output of the **Pixels To Frame** block is a 2-D matrix of a full image, there is no need to further carry on the bus containing five synchronization signals.

The **Number of components** field and the **Video format** fields of both Frame To Pixels and Pixels To Frame are set at 1 and 240p, respectively, to match the format of the video source.

Verifying the Pixel Stream Processing Design

While building the streaming portion of the design, the **PSNR** block continuously verifies results against the original full-frame design. The **Delay** block on the top level of the model time-aligns the 2-D matrices for a fair comparison. During the course of the simulation, the **PSNR** block should give **inf** output, indicating that the output image from the **Full-Frame Behavioral Model** matches the image generated from the stream processing **Pixel-Stream HDL Model**.

Exploring the Example

The example allows you to experiment with different threshold and alpha values to examine their effect on the quality of the overlaid images. Specifically, two workspace variables $thresholdValue$ and $alpha$ with initial values 7 and 0.8, respectively, are created upon opening the model. You can modify their values using the MATLAB command line as follows:

```
thresholdValue=8
alpha=0.5
```

The updated $thresholdValue$ will be propagated to the **Threshold** field of the **Edge Detection** block inside the **Full-Frame Behavioral Model** and the **Edge Detector** block inside **Pixel-Stream HDL Model/Edge Detection**. The $alpha$ value will be propagated to the **Gain1** block in the **Full-Frame Behavioral Model** and **Pixel-Stream HDL Model/Image Overlay**, and the value of $1 - alpha$ goes to **Gain2** blocks. Closing the model clears both variables from your workspace.

In this example, the valid range of $thresholdValue$ is between 0 and 256, inclusive. Setting $thresholdValue$ equal to or greater than 257 triggers a message **Parameter overflow occurred for 'threshold'**. The higher you set the $thresholdValue$, the smaller the amount of edges the example finds in the video.

The valid range of $alpha$ is between 0 and 1, inclusive. It determines the weights for edge detection output image and the original source image before adding them. The overlay operation is a linear interpolation according to the following formula.

```
overlaid image = alpha*source image + (1-alpha)*edge image.
```

Therefore, when $alpha = 0$, the overlaid image is the edge detection output, and when $alpha = 1$ it becomes the source image.

Generate HDL Code and Verify Its Behavior

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('EdgeDetectionAndOverlayHDL/Pixel-Stream HDL Model');
```

To generate a test bench, use the following command:

```
makehdltb('EdgeDetectionAndOverlayHDL/Pixel-Stream HDL Model');
```

Edge Detection and Image Overlay with Impaired Frame

This example shows how to introduce impairments in order to test a design with imperfect video input. When designing video processing algorithms, an important concern is the quality of the incoming video stream. Real-life video systems, like surveillance cameras or camcorders, produce imperfect signals. The stream can contain errors such as active lines of unequal length, glitches, or incomplete frames. In simulation, a streaming video source will usually produce perfect signals. When you use the **Frame To Pixels** block from the Vision HDL Toolbox™, all lines are of equal size, and all frames are complete. A video algorithm that simulates well under these conditions does not guarantee its effectiveness on an FPGA that connects to a real-world video source. To assess the robustness of a video algorithm under nonideal real-world video signals, it is practical to introduce impairments in the pixel stream.

This example extends the "Edge Detection and Image Overlay" on page 2-19 example by manually masking off the leading control signals of a frame to resemble a scenario where the algorithm starts in the middle of a frame. Such test scenarios are necessary to prove robustness of streaming video designs.

It is beneficial to go over the "Edge Detection and Image Overlay" on page 2-19 example before proceeding to this example.

Structure of the Example

The structure of this example is shown below, which closely follows the structure of the pixel-stream processing unit of the model in "Edge Detection and Image Overlay" on page 2-19.
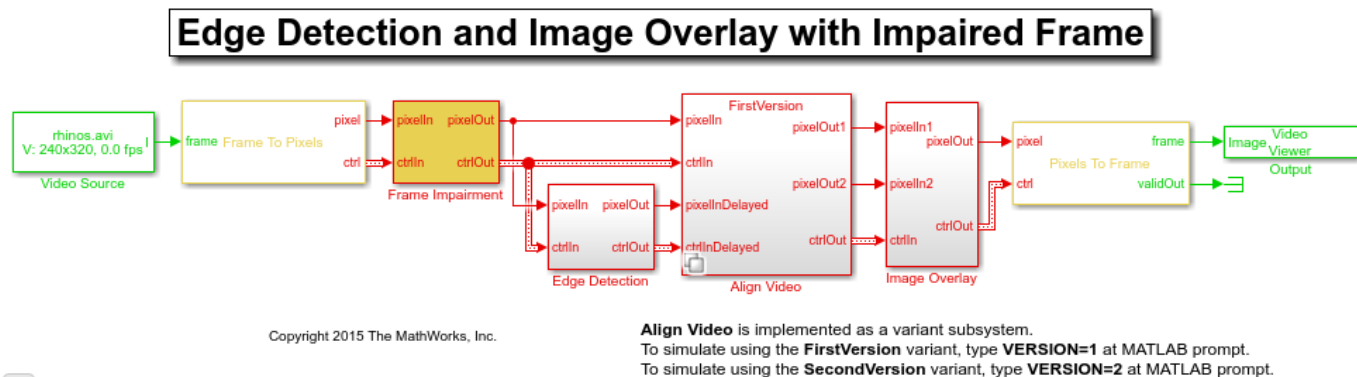
The **Edge Detection** subsystem implements a Sobel algorithm to highlight the edge of an image. The **Align Video** subsystem is used to synchronize the delayed output of the **EdgeDetector** with the original frame. **Image Overlay** weights and sums up the two time-aligned images.

This material is organized as follows. We first develop an **Align Video** subsystem that works well with perfect video signals. Then, we use the **Frame Impairment** subsystem to mask off the leading control signals of a frame to resemble a scenario where the algorithm starts in the middle of a frame. We will see that such impairment makes **Align Video** ineffective. Finally, a revised version of **Align Video** is developed to address the issue.

Align Video is implemented as a variant subsystem. You can use the variable VERSION in workspace to select which one of the two versions you want to simulate.

Note: Starting in R2017a the Pixel Stream Aligner block replaces the Align Video subsystem shown here. This new block makes setting the line buffer size and number of lines much easier and generates HDL code. In new designs, use the Pixel Stream Aligner block rather than the Align Video subsystem. For an example of how to use the block, see "Edge Detection and Image Overlay" on page 2-19.

First Version of Align Video

The following diagram shows the structure of the first version of the **Align Video** subsystem.
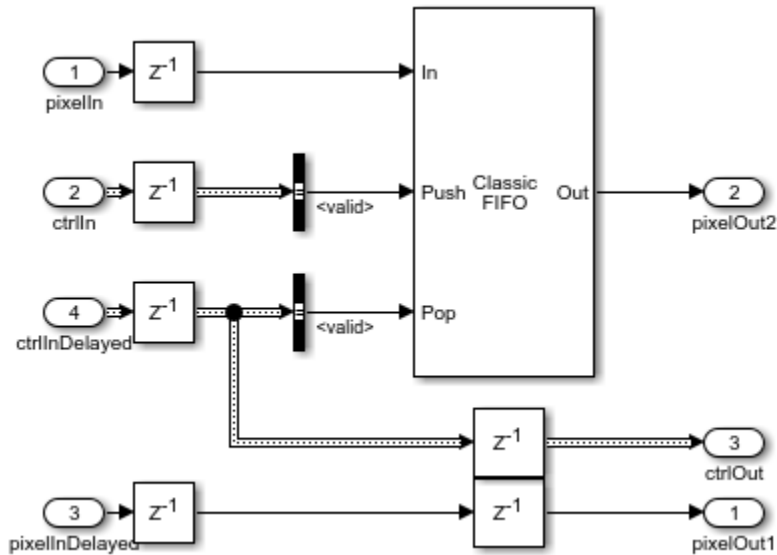
Align Video uses control signals to detect the active region of a frame. For more information on the streaming pixel protocol, see "Streaming Pixel Interface" on page 1-2.

The basic idea of aligning two pixel streams is to buffer valid pixels that come earlier into a FIFO based only on valid signals, and appropriately pop individual pixel from this FIFO based on the valid signal of the delayed pixel-stream.

Test Align Video Using Frame Impairment Subsystem

To illustrate how the **Frame Impairment** subsystem works, consider a 2-by-3 pixel frame. In the figure below, this frame is showed in the dashed rectangle with inactive pixels surrounding it. Inactive pixels include a 1-pixel-wide back porch, a 2-pixel-wide front porch, 1 line before the first active line, and 1 line after the last active line. Both active and inactive pixels are labeled with their grayscale values.
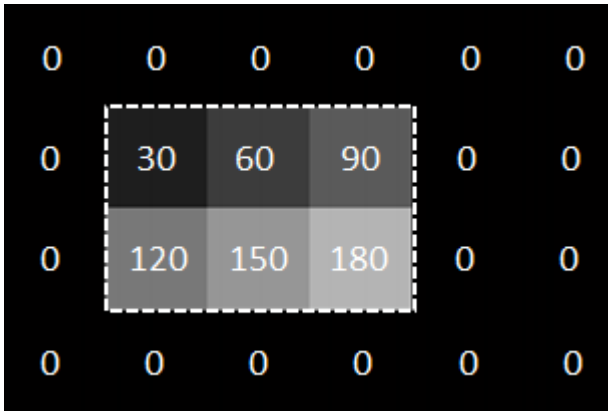
If the **Frame To Pixels** block accepts this 2-by-3 frame as an input and its settings correspond to the porch lengths shown above, then the timing diagram of the **Frame To Pixels** output is illustrated in the upper half of the following diagram.

Perfect Frame

Impaired Frame: 4 Valid Pixels Skipped
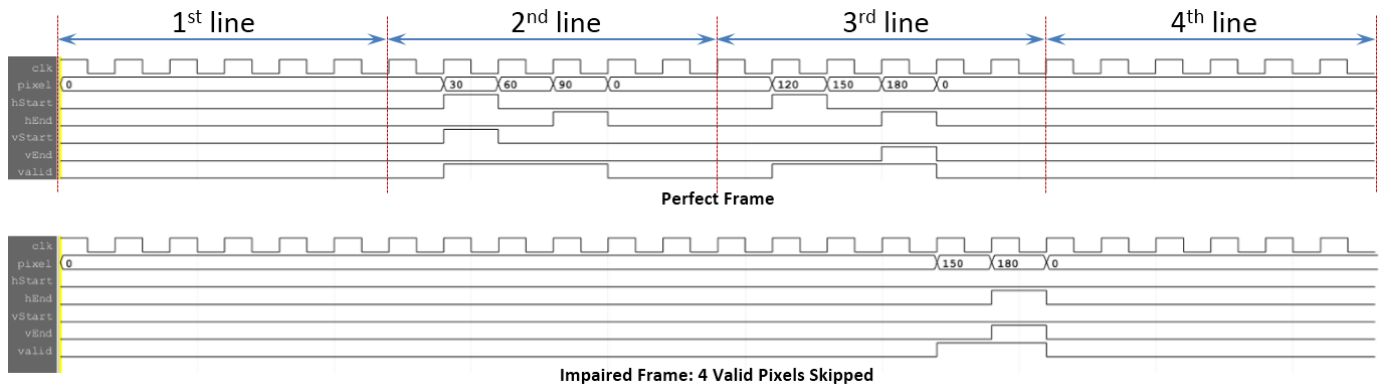
The **Frame Impairment** subsystem skips a configurable number of valid pixels at the beginning of the simulation. For example, if it was configured to skip 4 pixels of the example frame, the result would be as in the lower half of the timing diagram. We can see that by skipping 4 valid pixels, the three valid pixels on the second line (i.e., with intensity values of 30, 60, and 90), and the first valid pixel on the third line, are masked off, along with their associated control signals. Moreover, the **Frame Impairment** subsystem introduces two clock cycle delays. If we enter 0 pixels to skip, it just delays both pixel and ctrl outputs from **Frame To Pixels** by two clock cycles.

Double-click the **Frame Impairment** subsystem and ensure 'Number of valid pixels to skip' is set to 0. As mentioned before, this setting does not impair the frame, all it does is to delay both pixel and ctrl outputs from **Frame To Pixels** by two clock cycles. The output from the video output is shown below, which is expected.
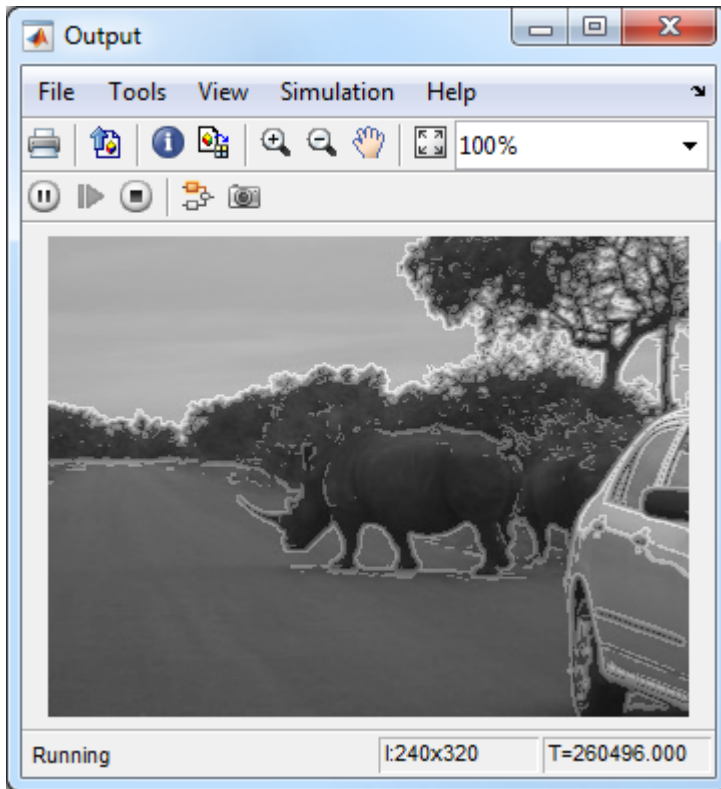
Now, double-click **Frame Impairment** again and enter any positive integer number, say 100, in the 'Number of valid pixels to skip' field.

Rerun the model and the resulting video output is shown below.
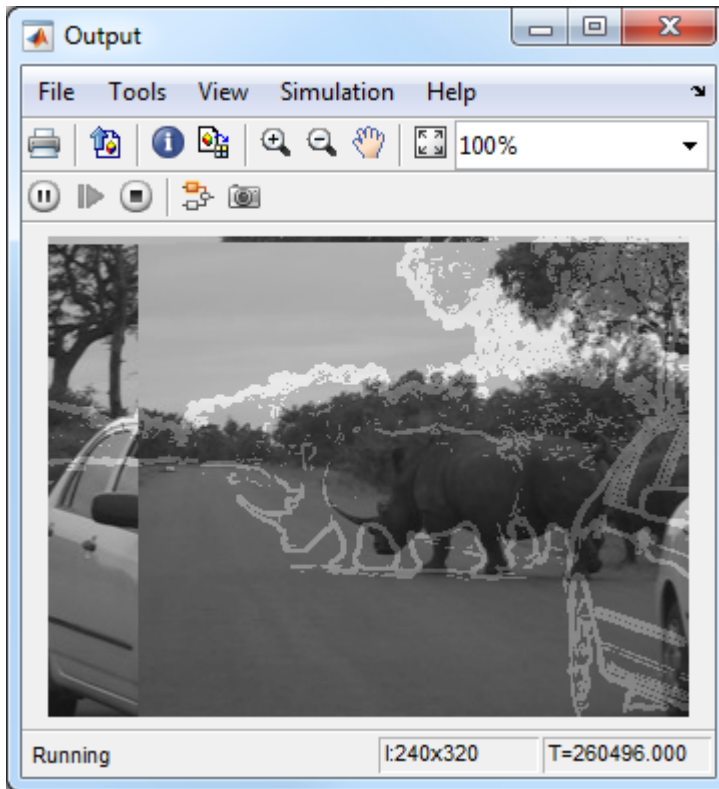
We can see that the edge output is at the right place but the original image is shifted. This output clearly suggests that our first version of **Align Video** is not robust against a pixel stream that starts in the middle of a frame.

Two reasons explain this behavior. Firstly, **EdgeDetector** block starts processing only after seeing a valid frame start, indicated by hStart, vStart, and valid going high at the same clock cycle. The block does not output anything for a partial frame. Secondly, the FIFO, inside the **Align Video** subsystem, starts buffering the frame once the valid signal is true, whether it is a partial frame or a complete frame. Therefore, at the start of the second frame, FIFO has been contaminated with the pixels of the previous partial frame.

Corrected Version of Align Video

Based on the insight gained from the previous section, a revised version of **Align Video** is shown below.
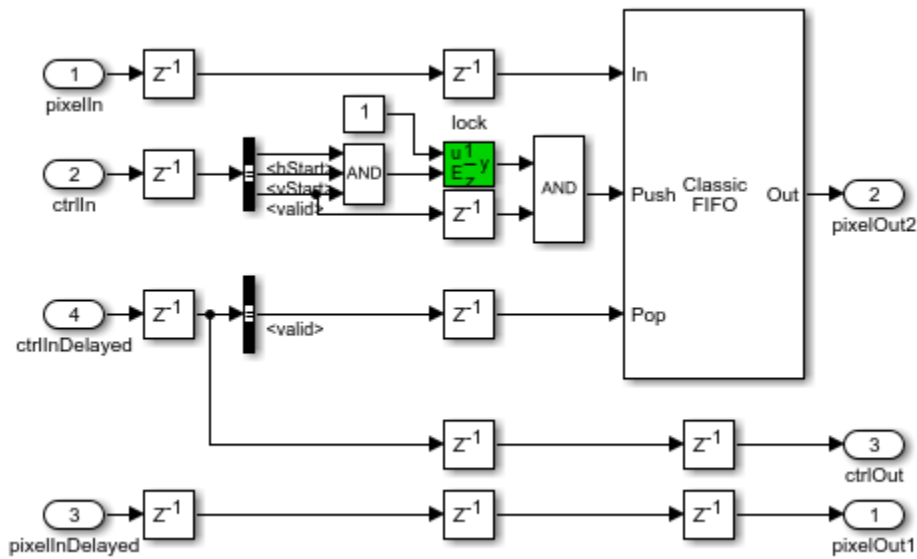
The goal is to only push the pixels of complete frames into the FIFO. If the leading frames are not complete, their valid pixels are ignored.

To achieve this, an enabled register called **lock** is used (highlighted in the diagram above). Its initial value is logical 0. ANDing this 0 with a delayed version of valid always gives logical 0. This prevents any valid pixels from being pushed into FIFO. The **lock** toggles its output from logical 0 to 1 only when hStart, vStart, and valid signals assert high, an indicator of the start of a new frame. After **lock** toggles to 1, the 'push' input of FIFO now follows a delayed version of the valid signal. So the valid pixels of a new frame will be buffered in FIFO.

To test this revised implementation, type the following command at MATLAB prompt.

```
VERSION=2;
```

Rerun the simulation. Now the edge output and the original image are perfectly aligned.

Noise Removal and Image Sharpening

This example shows how to implement a front-end module of an image processing design. This front-end module removes noise and sharpens the image to provide a better initial condition for the subsequent processing.

An object out of focus results in a blurred image. Dead or stuck pixels on the camera or video sensor, or thermal noise from hardware components, contribute to the noise in the image. In this example, the front-end module is implemented using two pixel-stream filter blocks from the Vision HDL Toolbox™. The median filter removes the noise and the image filter sharpens the image. The example compares the pixel-stream results with those generated by the full-frame blocks from the Computer Vision System Toolbox™.

This example model provides a hardware-compatible algorithm. You can implement this algorithm on a board using a Xilinx™ Zynq™ reference design. See "Image Sharpening with Zynq-Based Hardware" (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

Structure of the Example

Computer Vision System Toolbox blocks operate on an entire frame at a time. Vision HDL Toolbox blocks operate on a stream of pixel data, one pixel at a time. The conversion blocks in Vision HDL Toolbox, Frame To Pixels and Pixels To Frame, enable you to simulate streaming-pixel designs alongside full-frame designs.

The NoiseRemovalAndImageSharpeningHDL.slx system is shown below.
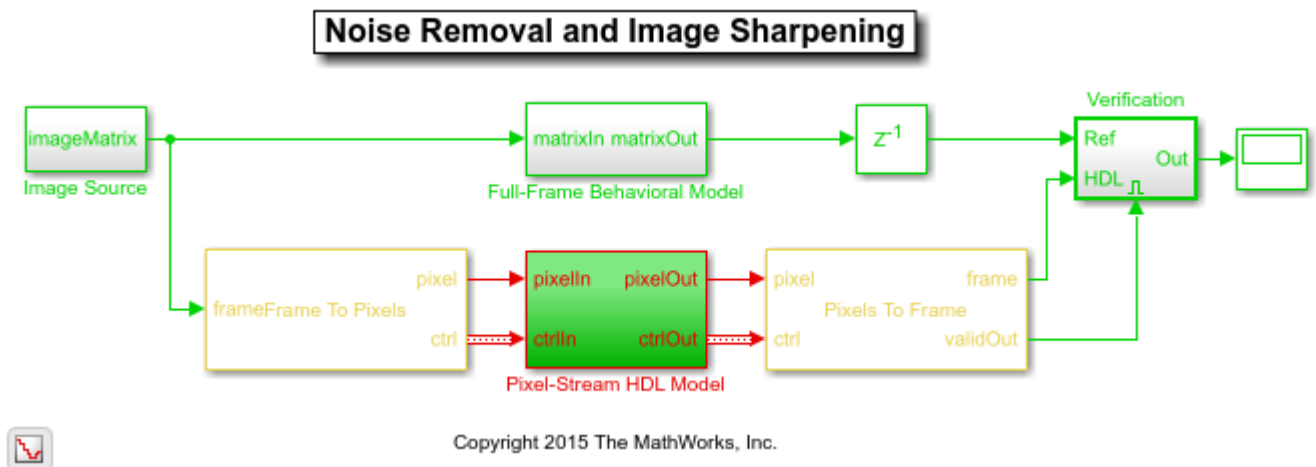
The following diagram shows the structure of the Full-Frame Behavioral Model subsystem, which consists of the frame-based Median Filter and 2-D FIR Filter. As mentioned before, median filter removes the noise and 2-D FIR Filter is configured to sharpen the image.
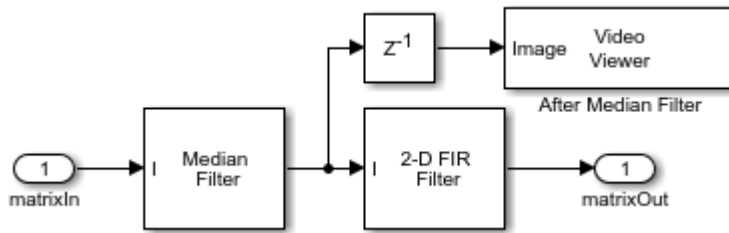
The Pixel-Stream HDL Model subsystem contains the streaming implementation of the median filter and 2-D FIR filter, as shown in the diagram below. You can generate HDL code from the Pixel-Stream HDL Model subsystem.
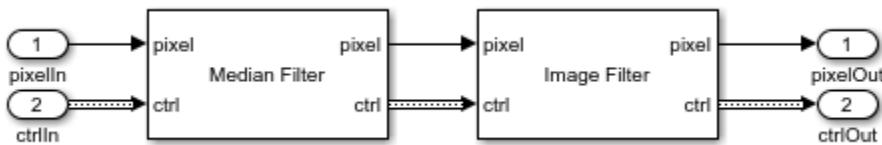
The Verification subsystem compares the results from full-frame processing with those from pixel-stream processing.

One frame of the blurred and noisy source video, its de-noised version after median filtering, and the sharpened output after 2-D FIR filtering, are shown from left to right in the diagram below.
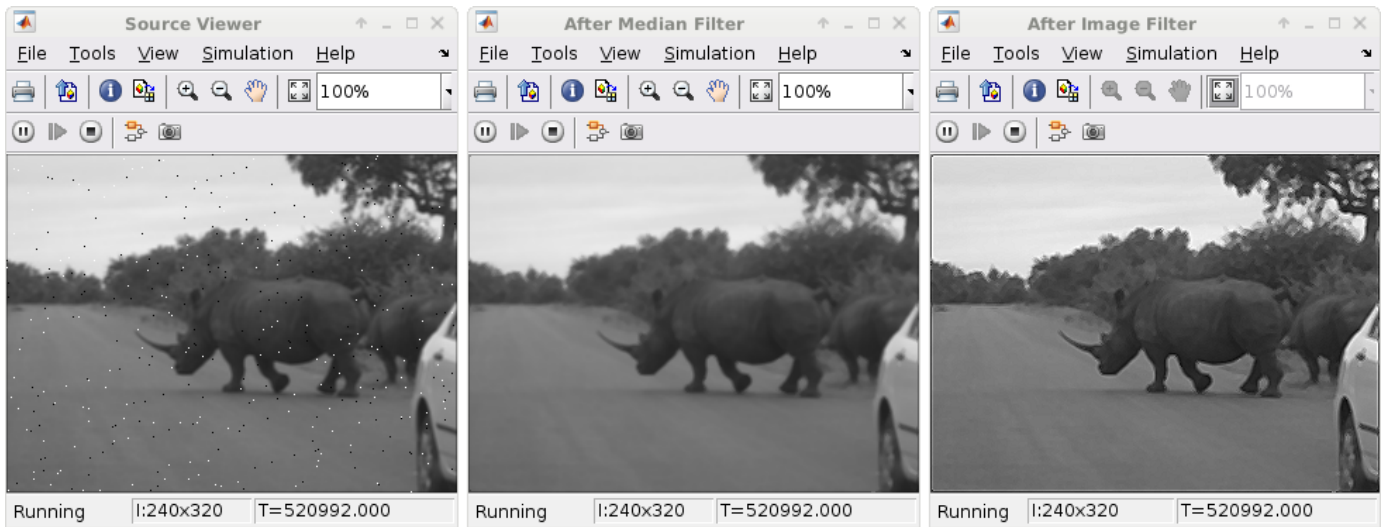
Image Source

The following figure shows the Image Source subsystem.
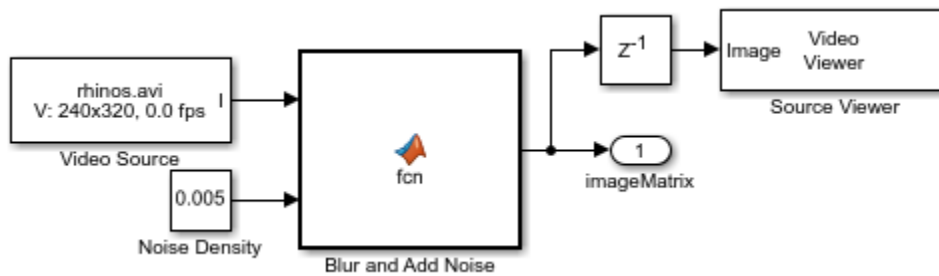
The Image Source block imports a grayscale image, then uses a MATLAB function block named Blur and Add Noise to blur the image and inject salt-and-pepper noise. The IMFILTER function uses a 3-by-3 averaging kernel to blur the image. The salt-and-pepper noise is injected by calling the IMNOISE(I,'salt & pepper',D) command, where D is the noise density defined as the ratio of the combined number of salt and pepper pixels to the total pixels in the image. This density value is specified by the Noise Density constant block, and it must be between 0 and 1. The Image Source subsystem outputs a 2-D matrix of a full image.

Frame To Pixels: Generating a Pixel Stream

The Frame To Pixels block converts a full image frame to a pixel stream. The Number of components field is set to 1 for grayscale image input, and the Video format field is 240p to match that of the video source. The sample time of the Video Source is determined by the product of Total pixels per line and Total video lines in the Frame To Pixels block. For more information, see the Frame To Pixels block reference page.

Pixel-Stream HDL Model

The Median Filter block is used to remove the salt and pepper noise. To learn more, refer to the Median Filter block reference page.

Based on the filter coefficients, the Image Filter block can be used to blur, sharpen, or detect the edges of the recovered image after median filtering. In this example, Image Filter is configured to sharpen an image. To learn more, refer to the Image Filter block reference page.

Pixels To Frame: Converting Pixel Stream Back to Full Frame

The Pixels To Frame block converts a pixel stream to the full frame by making use of the synchronization signals. The Number of components field and the Video format field of the Pixels To Frame are set at 1 and 240p, respectively, to match the format of the video source.

Verifying the Pixel-Stream Processing Design

The Verification subsystem, as shown below, verifies the results from the pixel-stream HDL model against the full-frame behavioral model.
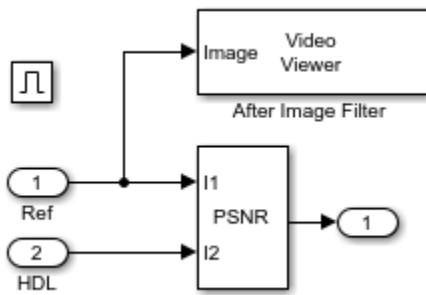
The peak signal to noise ratio (PSNR) is calculated between the reference image and the stream processed image. Ideally, the ratio should be inf, indicating that the output image from the Full-Frame Behavioral Model matches that generated from the Pixel-Stream HDL Model.

Generate HDL Code and Verify Its Behavior

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('NoiseRemovalAndImageSharpeningHDL/Pixel-Stream HDL Model');
```

To generate test bench, use the following command:

```
makehdltb('NoiseRemovalAndImageSharpeningHDL/Pixel-Stream HDL Model');
```

Multi-Zone Metering

This example shows how to use the Image Statistics block to perform multi-zone metering to extract a region of interest (ROI).

There are numerous applications where the input video is divided into several zones, and the statistic is then computed over each zone. For example, many auto-exposure algorithms compute the difference in the mean intensity between zones. This allows the shutter controller logic to determine whether the image is under-exposed (overall low illumination), correctly-exposed (uniform illumination) or over-exposed (one or more ROIs have a larger mean).

Introduction

The MultizoneMeteringHDL.slx system is shown below.
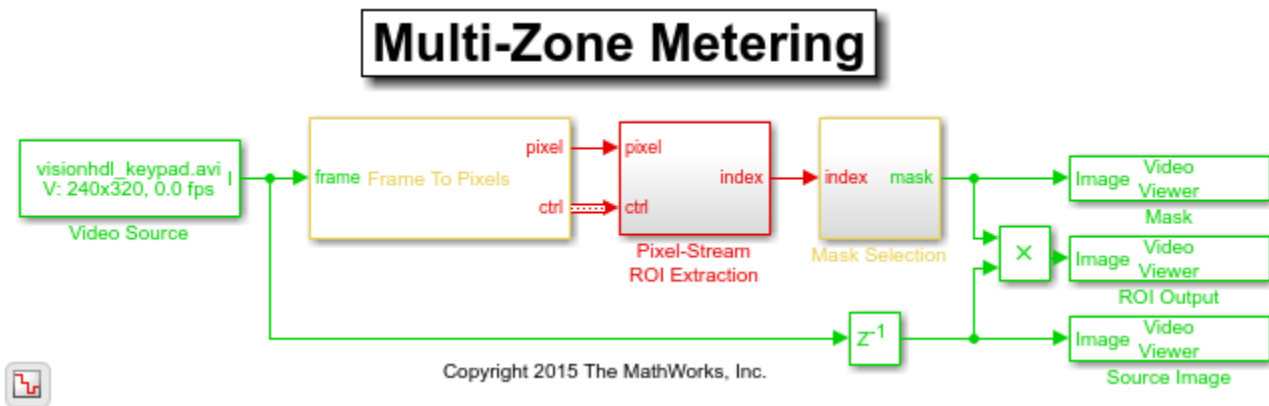
The green and red lines represent full-frame processing and pixel-stream processing, respectively. The color difference indicates the change in the image rate on the streaming branch of the model. This rate transition is because the pixel stream is sent out in the same amount of time as the full video frames and therefore it is transmitted at a higher rate.

In this example, the **Pixel-Stream ROI extraction** subsystem calculates the mean intensity value over 12 predefined ROIs in a frame and outputs the index number (1-12) that corresponds to the most illuminated ROI. The downstream **Mask Selection** subsystem accepts this index number and outputs the associated binary mask image. The binary mask image is applied to the source video to display only the most illuminated ROI, and mask off the other 11 ROIs. The **Delay** block at the top level of the model is used to match the latency introduced by pixel-stream processing.

One frame of the source image, the binary mask image, and the ROI output, are shown from left to right in the diagram below.
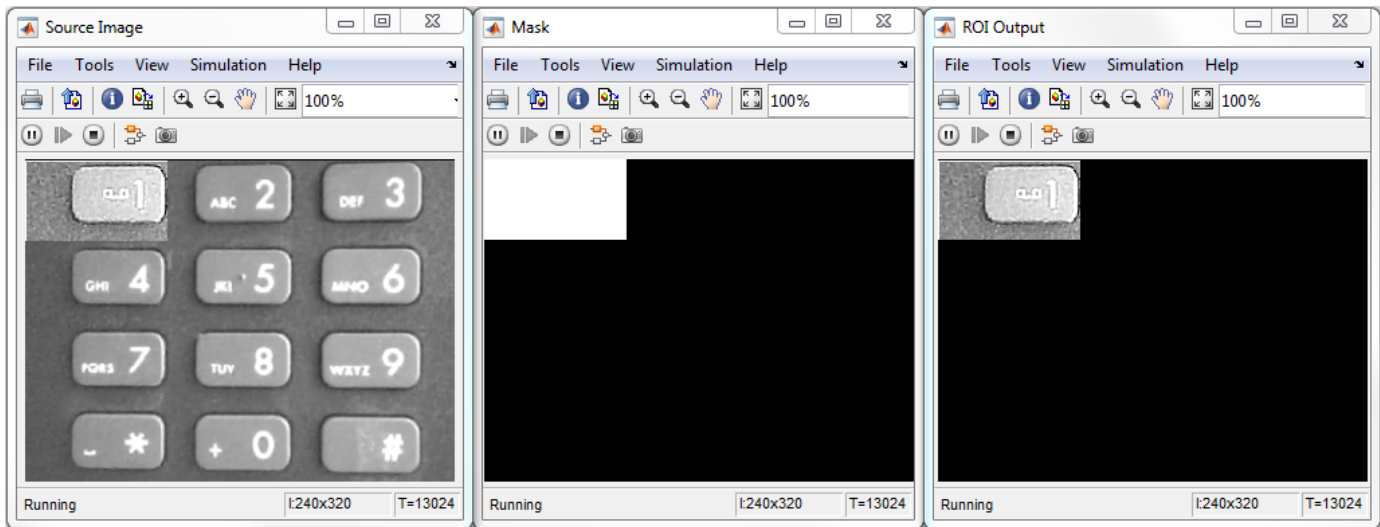
You can generate HDL code from the **Pixel-Stream ROI Extraction** subsystem.

Video Source

The video format is 240p. Each frame consists of 240 lines and 320 pixels per line. In this example, video frames are divided into 12 non-overlapping rectangular ROIs, denoted as ROI number 1 to 12, as shown in the diagram below. Each ROI includes one key of the input keypad image.

ROI number 1 has a 107-pixel width and a 60-pixel height, and the (x,y) coordinate of its top-left pixel is (1,1). ROI number 2 has a 107-pixel width and a 60-pixel height, and the coordinate of its top left pixel is (108,1), and so on. The first frame of the input video has brighter pixels within ROI number 1, as shown above. The second frame has brighter pixels within ROI number 2, and so on.

Frame To Pixels: Generating a Pixel Stream

Frame To Pixels converts a full-frame image to a 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:

The **Number of components** field is set to 1 for grayscale image input, and the **Video format** field is 240p to match that of 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** is determined by the product of **Total pixels per line** and **Total video lines**.

Pixel-Stream ROI Extraction

The **Pixel-Stream ROI Extraction** subsystem contains two subsystems, namely, **Multi-Zone Metering** and **ROI Indexer**.
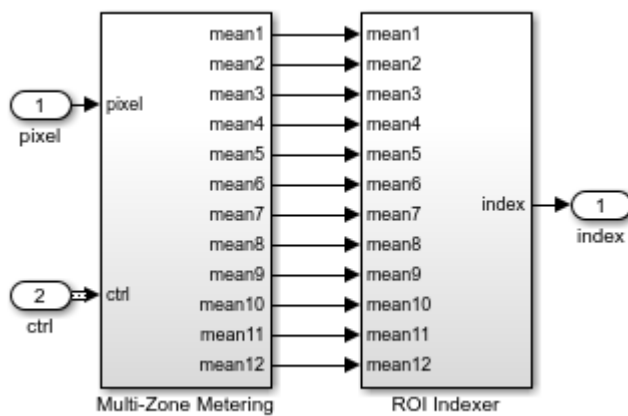
The **Multi-Zone Metering** subsystem computes the mean intensity value over the 12 predefined ROIs. The resulting 12 mean values are passed to the downstream **ROI Indexer** subsystem. **ROI Indexer** outputs the index (1-12) of the ROI that has the maximum mean intensity value (or equivalently, the most illuminated ROI) among the 12 candidates.

The structure of the **Multi-Zone Metering** subsystem is shown in the diagram below.
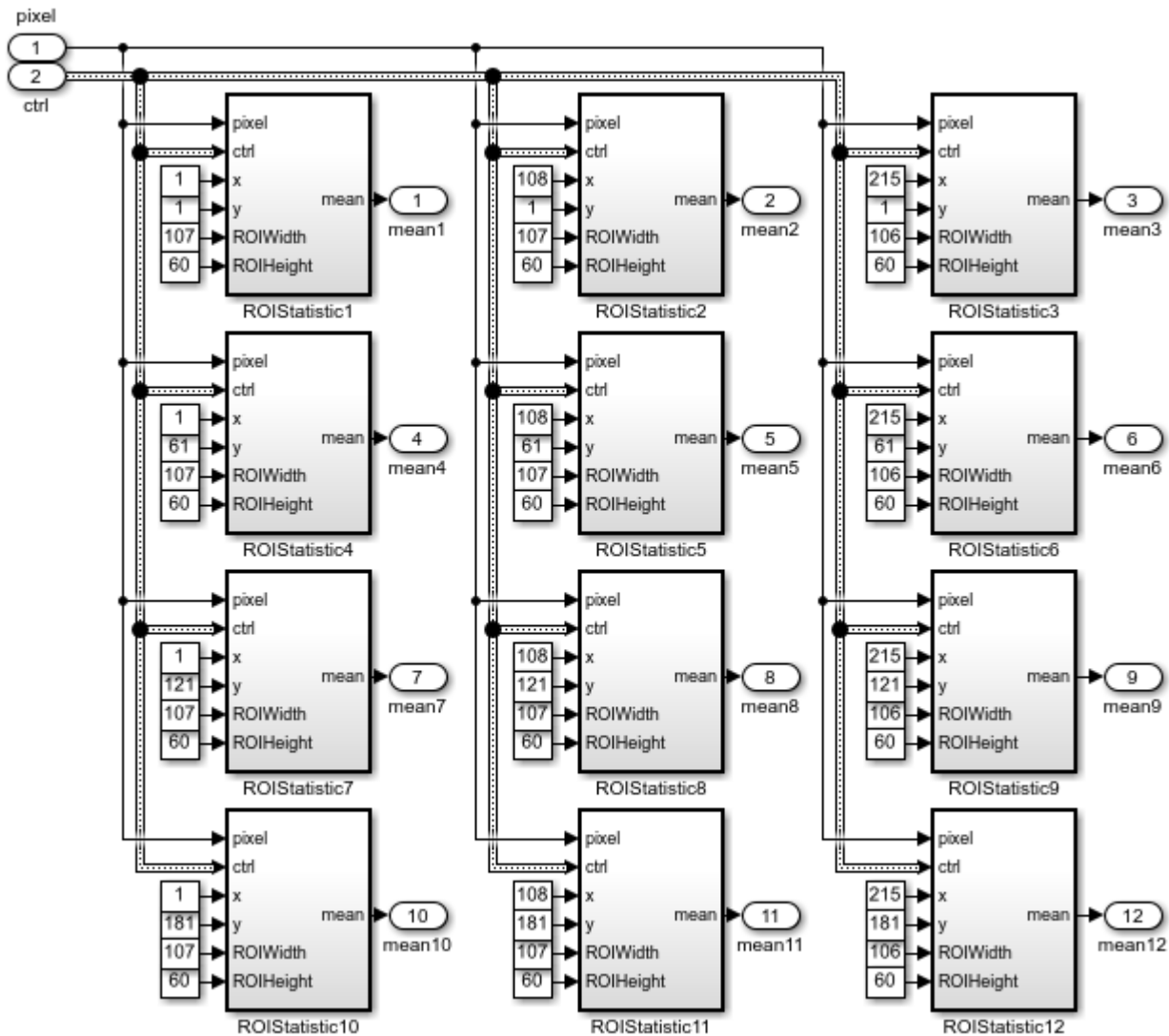
The **Multi-Zone Metering** subsystem contains 12 identical **ROIStatistic** subsystems. Each instance of **ROIStatistic** calculates the mean intensity value over one ROI. All of the 12 **ROIStatistic** subsystems take pixel and ctrl as their first two inputs. The remaining four inputs specify which ROI this subsystem works on and they are different from one subsystem to another. For example, the **ROIStaticstic1** subsystem focuses on ROI number 1 by accepting the (x,y) coordinate of the top left pixel (1,1), ROI width of 107, and height 60. Similarly, the **ROIStaticstic12** subsystem focuses on ROI number 12, whose (x,y) coordinate of the top left pixel is (215,181), and whose width and height are 106 and 60, respectively.

The **ROIStatistic1** - **ROIStatistic12** subsystems share the same structure shown below.
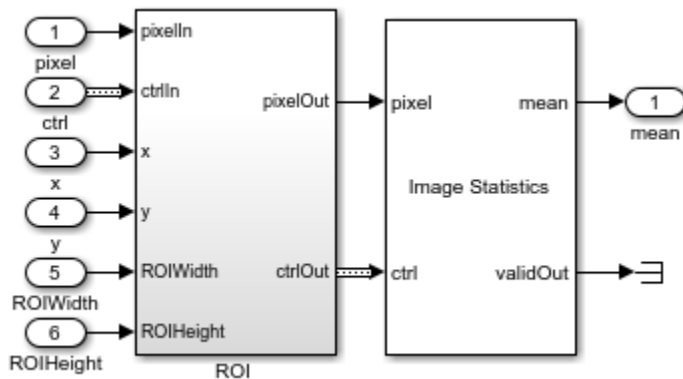
It contains a **ROI** subsystem followed by an **Image Statistics** block. The **ROI** subsystem manipulates the control signal of the original 240p image, and constructs the control signals associated only with the ROI specified by (x,y) pair, ROIWidth, and ROIHeight.

Mask Selection

The structure of the **Mask Selection** subsystem is shown below.
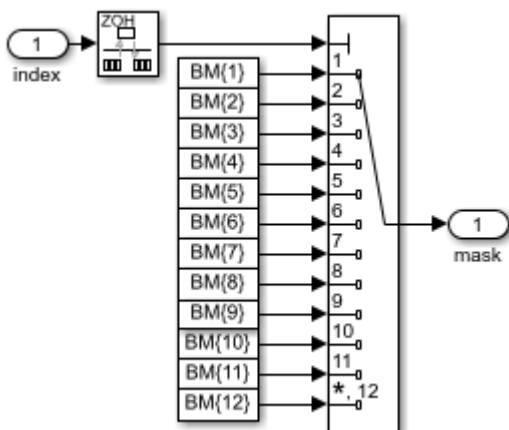
Twelve mask images are available, corresponding to the 12 different ROIs. These mask patterns are shown as BM{1} to BM{12} in the above diagram. When you open the model, the model loads the predefined BM cell array into the workspace. Masks are binary images with 240p video format. For mask BM{n} (n=1,2,...,12), the ROI number n is filled with logical 1 pixels (white) and all the other 11 ROIs are filled with logical 0 pixels (black). Based on the index input (1-12), the **Mask Selection** subsystem outputs the associated binary mask image.

HDL Code Generation

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('MultizoneMeteringHDL/Pixel-Stream ROI Extraction')
```

2-39

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

```
makehdltb('MultizoneMeteringHDL/Pixel-Stream ROI Extraction')
```

Harris Corner Detection

This example shows how to use edge detection as the first step in corner detection.

Corner detection is used in computer vision systems to find features in an image. It is often one of the first steps in applications like motion detection, tracking, image registration and object recognition.

A corner is intuitively defined as the intersection of two edges. This example uses the Harris & Stephens algorithm [1] in which the computation is simplified using an approximation of the eigenvalues of the Harris matrix. For an alternative corner detection design, see the "FAST Corner Detection" on page 2-47 example.

This example model provides a hardware-compatible algorithm. You can implement this algorithm on a board using a Xilinx™ Zynq™ reference design. See "Corner Detection and Image Overlay with Zynq-Based Hardware" (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

Introduction

The CornerDetectionHDL.slx system is shown below.
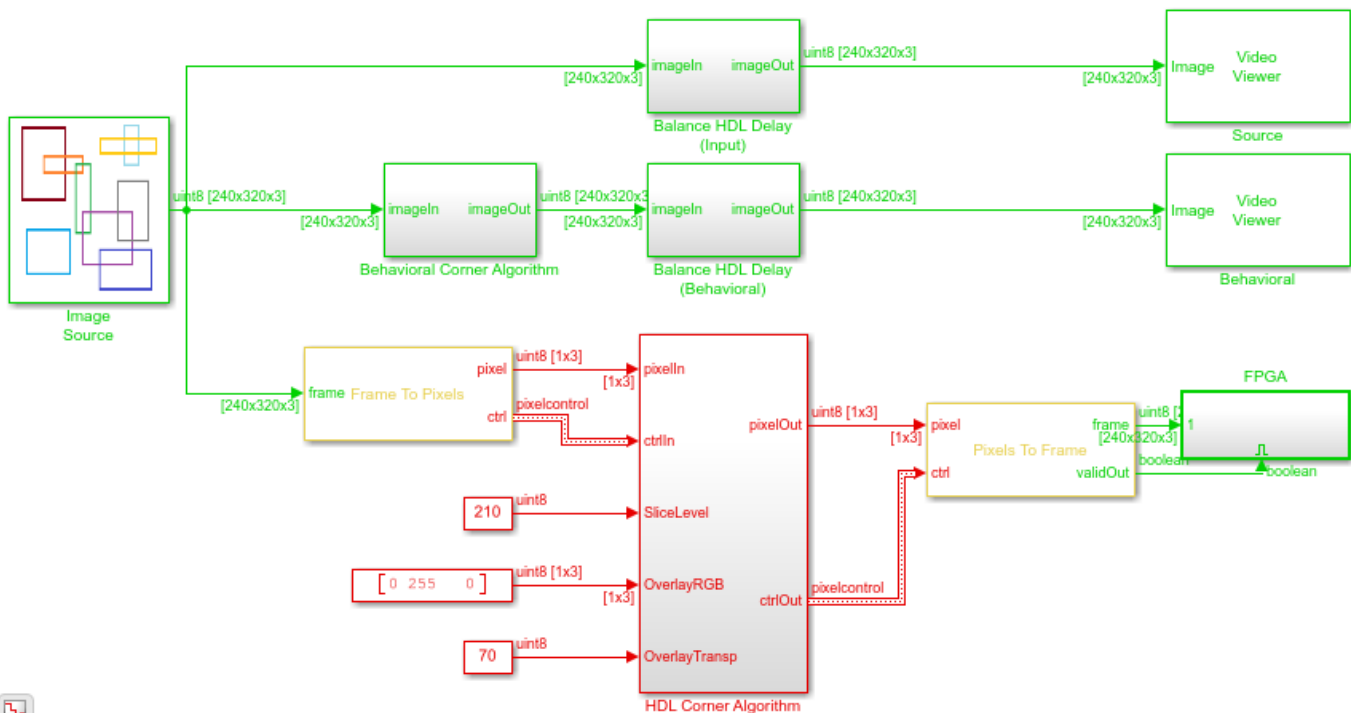
First Step: Find the Gradients

The first step is to find the edges in the image. Use two gradient image filters with coefficients
$\begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$ and $\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$ to produce gradients G_x and G_y. Square and cross-multiply to form G_x^2, G_y^2 and G_{xy}.
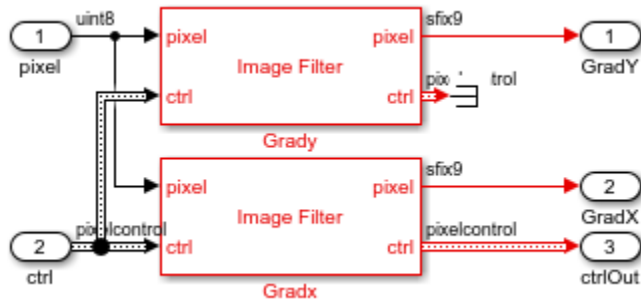
Second Step: Circular Filtering

The second step of the algorithm is to perform Gaussian filtering to average G_x^2, G_y^2 and G_{xy} over a circular window. The size of the circular window determines the scale of the detected corner. Here a 5x5 Gaussian is selected. Since we have three components, use three filters, each with the same filter coefficients.
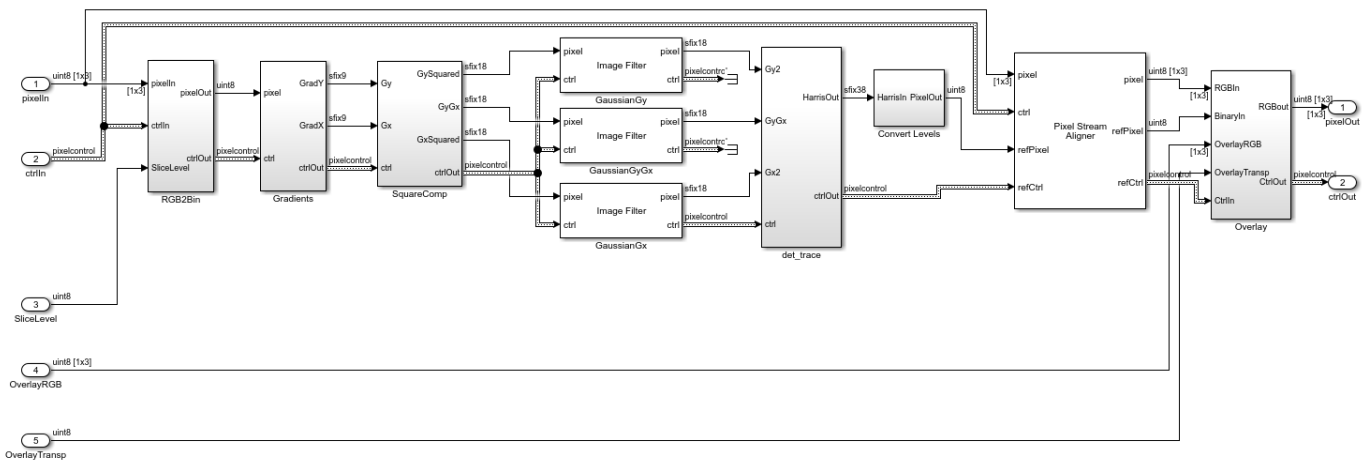
Final Step: Form the Harris Matrix

The final step of the algorithm is to estimate the eigenvalue of the Harris matrix. The Harris matrix is a symmetric matrix similar to a covariance matrix. The main diagonal is composed of the two averages of the gradients squared $\langle G_x^2 \rangle$ and $\langle G_y^2 \rangle$. The off diagonal elements are the averages of the gradient cross-product $\langle G_{xy} \rangle$. The Harris matrix is:

$$A_{Harris} = \begin{bmatrix} \langle G_x^2 \rangle & \langle G_{xy} \rangle \\ \langle G_{xy} \rangle & \langle G_y^2 \rangle \end{bmatrix}$$

Compute the Response from the Harris Matrix

The key simplification of the Harris algorithm is estimating the eigenvalues of the Harris matrix as the determinant minus the scaled trace squared.

$R = det(A_{Harris}) - k \cdot Tr^2(A_{Harris})$ where k is a constant typically 0.04.

The corner metric response, R, expressed using the gradients is:

$$R = \left(\langle G_x^2 \rangle \cdot \langle G_y^2 \rangle - \langle G_{xy} \rangle^2 \right) - k \cdot \left(\langle G_x^2 \rangle + \langle G_y^2 \rangle \right)^2$$

When the response is larger than a predefined threshold, a corner is detected:

$$R \quad > \quad k_{thresh}$$

$$\left(\langle G_x^2 \rangle \cdot \langle G_y^2 \rangle - \langle G_{xy} \rangle^2 \right) - k \cdot \left(\langle G_x^2 \rangle + \langle G_y^2 \rangle \right)^2 \quad > \quad k_{thresh}$$

The det_trace subsystem computes R:
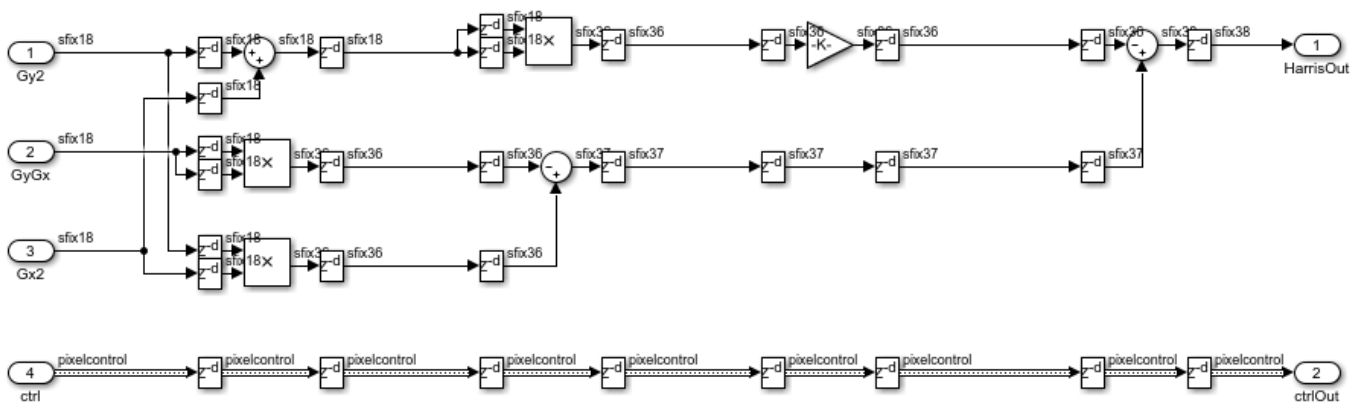
Fixed-Point Settings

The overall function from input image to output corner metric response is a fourth-order polynomial. This leads to some challenges determining the fixed-point scaling for each step of the computation. Since we are targeting FPGAs with built-in multipliers, the best strategy is to allow bit growth until the multiplier size is reached and then start to quantize results on a selective basis to stay within the bounds of the provided multipliers.

The input pixel stream is 8-bit grayscale pixel data. Computing the gradients does not add much bit-growth since the filter kernel has only +1 and -1 coefficients. The result is a full-precision 9-bit signed fixed-point type.

Squaring and cross-multiplying the gradients produces signed 18-bit results, still in full precision. Many common FPGA multipliers have 18-bit or 20-bit input wordlengths, so you will have to quantize at the next step.

The next step is to apply a circular window to the three components using three Image Filters with Gaussian coefficients. The coefficients are quantized to 18-bit unsigned numbers to fit the FPGA multipliers. To find the best fraction precision for the coefficients, create a fixed-point number using the fi() function but only specifying the wordlength. In this case a fractional scaling of 21-bits is best since the largest value in the coefficient matrix is between 1/8 and 1/16.

```
coeffs = fi(fspecial('gaussian',[5,5],1.5),0,18)
```

```
coeffs =
```

```
0.0144      0.0281      0.0351      0.0281      0.0144
0.0281      0.0547      0.0683      0.0547      0.0281
0.0351      0.0683      0.0853      0.0683      0.0351
0.0281      0.0547      0.0683      0.0547      0.0281
0.0144      0.0281      0.0351      0.0281      0.0144

       DataTypeMode: Fixed-point: binary point scaling
         Signedness: Unsigned
         WordLength: 18
      FractionLength: 21
```
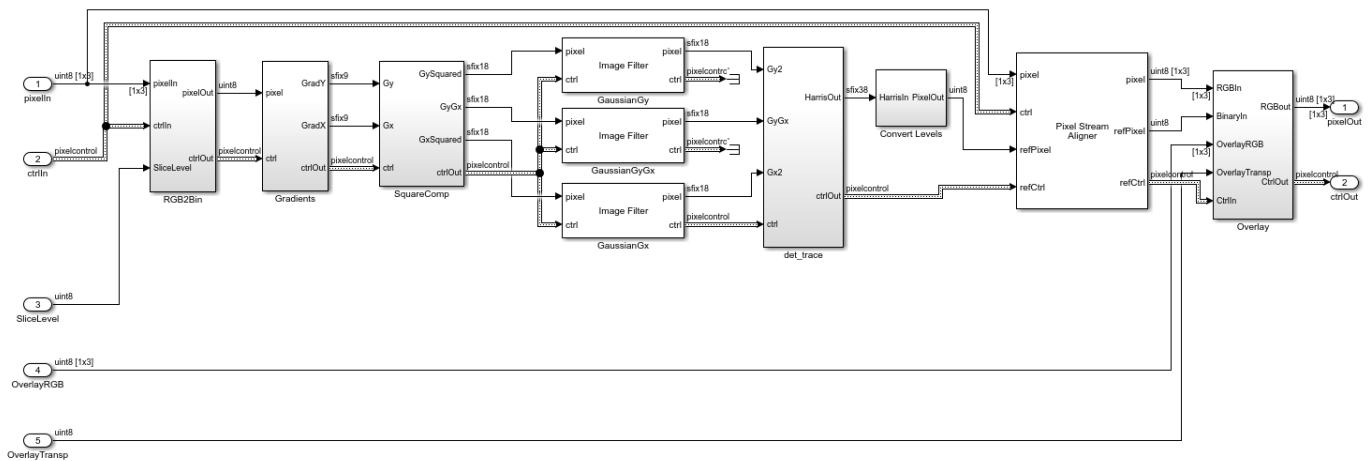

Results of the Simulation

You can see that the resulting images from the simulation are very similar but not exactly the same. The small differences in simulation results are because the behavioral model uses C integer arithmetic rules and the quantization is different from the HDL-ready corner detection block.

Using Simulink, you can understand these differences and decide if the errors are allowable for your application. If they are not acceptable, you can increase the bit-widths of the operators, although this increases the area used in the FPGA.

HDL Code Generation

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('CornerDetectionHDL/HDL Corner 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. You may want to reduce the simulation time before generating the test bench.

```
makehdltb('CornerDetectionHDL/HDL Corner Algorithm')
```

The part of this model that you can implement on an FPGA is the part between the Frame To Pixels and Pixels To Frame blocks. That is the subsystem called HDL Corner Algorithm, which includes all

elements of the corner detection algorithm seen above. The rest of the model, including the Behavioral Corner Algorithm and the sources and sinks, form our Simulink test bench.

Going Further

The Harris & Stephens algorithm is based on approximating the eigenvalues of the Harris matrix as shown above. The Harris algorithm uses $R = det(A_{Harris}) - k \cdot Tr^2(A_{Harris})$ as a metric, avoiding any division or square-root operations. Another way to do corner detection is to compute the actual eigenvalues.

The analytical solution for the eigenvalues of a 2x2 matrix is well-known and can also be used in corner detection. When the eigenvalues are both positive and large with the same scale, a corner has been found.

$$\lambda_1 = \frac{Tr(A)}{2} + \sqrt{\frac{Tr^2(A)}{4} - det(A)}$$

$$\lambda_2 = \frac{Tr(A)}{2} - \sqrt{\frac{Tr^2(A)}{4} - det(A)}$$

Substituting in our A_{Harris} values we get:

$$\lambda_1 = \left(\frac{\langle G_x^2 \rangle + \langle G_y^2 \rangle}{2}\right) + \sqrt{\left(\frac{\langle G_x^2 \rangle + \langle G_y^2 \rangle}{2}\right)^2 - \left(\langle G_x^2 \rangle \cdot \langle G_y^2 \rangle - \langle G_{xy} \rangle^2\right)}$$

$$\lambda_2 = \left(\frac{\langle G_x^2 \rangle + \langle G_y^2 \rangle}{2}\right) - \sqrt{\left(\frac{\langle G_x^2 \rangle + \langle G_y^2 \rangle}{2}\right)^2 - \left(\langle G_x^2 \rangle \cdot \langle G_y^2 \rangle - \langle G_{xy} \rangle^2\right)}$$

For FPGA implementation it is important to notice the repeated value of $\frac{Tr(A)}{2}$. We can compute this value once and then square to combine with $det(A)$. This means that the eigenvalue algorithm requires only two multipliers but at the expense of more adders and subtractors and a square-root function, which requires several multipliers on its own.

You must then compare both eigenvalues to a constant value to make sure they are large. Since the eigenvalues scale up with image intensity, you also need to make sure they are both around the same size. You can do this by subtracting one from another and making sure that result is smaller than some predefined threshold value. Notice that in this subtraction, the first terms cancel out and you are left with:

$$\lambda_1, \lambda_2 \quad > \quad k_{minimum}$$

$$\lambda_1 - \lambda_2 \quad < \quad k_{thresh}$$

$$2\sqrt{\frac{Tr^2(A)}{4} - det(A)} \quad < \quad k_{thresh}$$

$$\frac{Tr^2(A)}{4} - det(A) \quad < \quad \left(\frac{k_{thresh}}{2}\right)^2$$

You can rearrange this so that it is very similar to Harris metric R above:

$$det(A) - \frac{Tr^2(A)}{4} \geq \left(\frac{k_{thresh}}{2}\right)^2$$

Expanding the matrix gives:

$$\left(\langle G_x^2\rangle \cdot \langle G_y^2\rangle - \langle G_{xy}\rangle^2\right) - \left(\frac{\langle G_x^2\rangle + \langle G_y^2\rangle}{2}\right)^2 \geq \left(\frac{k_{thresh}}{2}\right)^2$$

The similarity between the difference of the eigenvalues and the Harris R metric shows how the Harris approximation works. If you rearrange the terms under the square-root and swap the signs so the result must be greater than or equal to a predefined threshold, you arrive at essentially the Harris metric with some scaling.

Conclusion

You implemented the Harris corner detection algorithm for an FPGA using Vision HDL Toolbox. You applied fixed-point quantization steps to the algorithm to make it FPGA capable. You also showed that the Harris metric R can be derived from the difference of eigenvalues.

References

[1] C. Harris and M. Stephens (1988). "A combined corner and edge detector". Proceedings of the 4th Alvey Vision Conference. pp. 147-151.

FAST Corner Detection

This example shows how to perform corner detection using the features-from-accelerated-segment test (FAST) algorithm. This algorithm is an alternative to the "Harris Corner Detection" on page 2-41 example. The FAST algorithm determines if a corner is present by testing a circular area around the potential center of the corner. The test detects a corner if a contiguous section of pixels are either brighter than the center plus a threshold or darker than the center minus a threshold.

In a software implementation this algorithm allows for a quick test to rule out potential corners by only testing the four pixels along the axes. Software algorithms only perform the full test if the quick test passes. A hardware implementation can easily perform all the tests in parallel so a quick test is not particularly advantageous and is not included in this example.

The FAST algorithm can be used at many sizes or scales. This example detects corners using a sixteen-pixel circle. In these sixteen pixels, if any nine contiguous pixel meet the brighter or darker limit then a corner is detected.

MATLAB FAST Corner Detection

The Computer Vision System Toolbox™ includes a software FAST corner detection algorithm in the `detectFASTFeatures` function. This example uses this function as the behavioral model to compare against the FAST algorithm design for hardware in Simulink®. The function has parameters for setting the minimum contrast and the minimum quality.

The minimum contrast parameter is the threshold value that is added or subtracted from the center pixel value before comparing to the ring of pixels.

The minimum quality parameter controls which detected corners are "strong" enough to be marked as actual corners. The strength metric in the original FAST paper is based on summing the differences of the pixels in the circular area to the central pixel [2]. Later versions of this algorithm use a different strength metric based on the smallest change in pixel value that would make the detection no longer a corner. `detectFastFeatures` uses the smallest-change metric.

This code reads the first frame of video, converts it to gray scale, and calls `detectFASTFeatures`. The result is a vector of corner locations. To display the corner locations, use the vector to draw bright green dots over the corner pixels in the output frame.

```matlab
v = VideoReader('rhinos.avi');
I = rgb2gray(readFrame(v));
% create output RGB frame
Y = repmat(I,[1 1 3]);
corners = detectFASTFeatures(I,'minContrast',15/255,'minQuality',1/255);
locs = corners.Location;
for ii = 1:size(locs,1)
    Y(floor(locs(ii,2)),floor(locs(ii,1)),2) = 255; % green dot
end
imshow(Y)
```

Limitations of the FAST Algorithm

Other corner detection methods work very differently from the FAST method and a surprising result is that FAST does not detect corners on computer generated images that are perfectly aligned to the x and y axes. Since the detected corner must have a ring of darker or lighter pixel values around the center that includes both edges of the corner, crisp images do not work well. For example, try the FAST algorithm on the input image used in the Harris "Harris Corner Detection" on page 2-41 example.

```
I = imread('cornerboxes.png');
Ig = rgb2gray(I);
corners = detectFASTFeatures(Ig,'minContrast',15/255,'minQuality',1/255)
```

```
corners =

  0x1 cornerPoints array with properties:

    Location: [0x2 single]
      Metric: [0x1 single]
       Count: 0
```

You can see that the function detected zero corners. This because the FAST algorithm requires a ring of contrasting pixels more than halfway around the center of corner. In the computer generated image, both edges of a box at a corner are in the ring of pixel used, so the test for a corner fails. A work-around to this problem is to add blur (by applying a Gaussian filter) to the image so that the corners are less precise but can be detected. After blurring, the FAST algorithm now detects over 100 corners.

```
h = fspecial('gauss',5);
Ig = imfilter(Ig,h);
corners = detectFASTFeatures(Ig,'minContrast',15/255,'minQuality',1/255)
```

```
locs = corners.Location;
for ii = 1:size(locs,1)
    I(floor(locs(ii,2)),floor(locs(ii,1)),2) = 255; % green dot
end
imshow(I)


corners =

  136x1 cornerPoints array with properties:

    Location: [136x2 single]
      Metric: [136x1 single]
       Count: 136
```
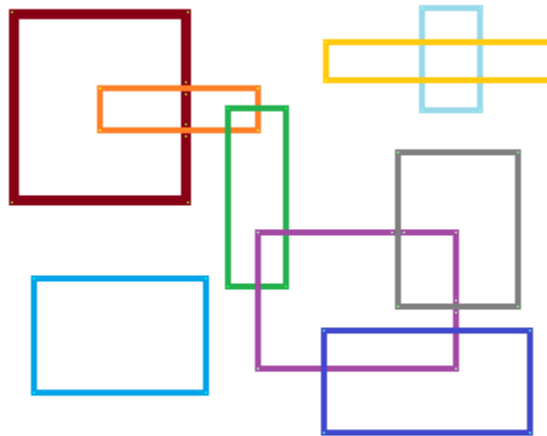

Behavioral Model for Verification

The Simulink model uses the `detectFASTFeatures` function as a behavioral model to verify the results of the hardware algorithm. You can use a MATLAB Function block to run MATLAB code in Simulink.

```
modelname = 'FASTCornerHDL';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
```
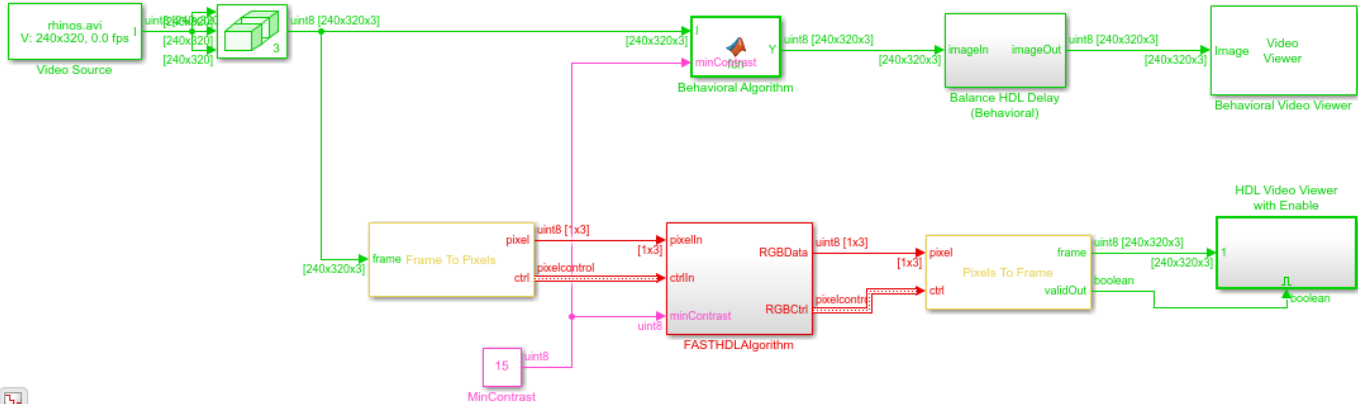
The code in a MATLAB Function block must either generate C code or be declared extrinsic. An extrinsic declaration allows the specified function to run in MATLAB while the rest of the MATLAB Function block runs in Simulink. The `detectFASTFeatures` function does not support code generation, so the MATLAB Function block must use an extrinsic helper function.

For frame-by-frame visual comparison, and the ability to vary the contrast parameter, the helper function takes an input image and the minimum contrast as inputs. It returns an output image with green dots marking the detected corners.

```
function Y = FASTHelper(I,minContrast)
Y = I;
corners = detectFASTFeatures(I(:,:,1),'minContrast',double(minContrast)/255,'minQuality',1/255);
locs = corners.Location;
for ii = 1:size(locs,1)
    Y(floor(locs(ii,2)),floor(locs(ii,1)),2) = 255; % green dot
end

end
```

The MATLAB Function block must have a defined size for the output array. A fast way to define the output size is to copy the input to the output before calling the helper function. This is the code inside the MATLAB Function block:

```
function Y = fcn(I,minContrast)
    coder.extrinsic('FASTHelper');
    Y = I;
    Y = FASTHelper(I,minContrast);
end
```

Implementation for HDL

The FAST algorithm implemented in the Vision HDL Toolbox Corner Detector block in this model tests 9 contiguous pixels from a ring of 16 pixels, and compares their values to the center pixel value. A kernel of 7x7 pixels around each test pixel includes the 16-pixel ring. The diagram shows the center pixel and the ring of 16 pixels around it that is used for the test. The ring pixels, clockwise from the top-middle, are

```
indices = [22 29 37 45 46 47 41 35 28 21 13 5 4 3 9 15];
```

These pixel indices are used for selection and comparison. The order must be contiguous, but the ring can begin at any point.

1	8	15	22	29	36	43
2	9	16	23	30	37	44
3	10	17	24	31	38	45
4	11	18	25	32	39	46
5	12	19	26	33	40	47
6	13	20	27	34	41	48
7	14	21	28	35	42	49

After computing corner metrics using these rings of pixels, the algorithm determines the maximum corner metric in each region and suppresses other detected corners. The model then overlays the non-suppressed corner markers onto the original input image.

The hardware algorithm is in the FASTHDLAlgorithm subsystem. This subsystem supports HDL code generation.

```
open_system([modelname '/FASTHDLAlgorithm'],'force');
```
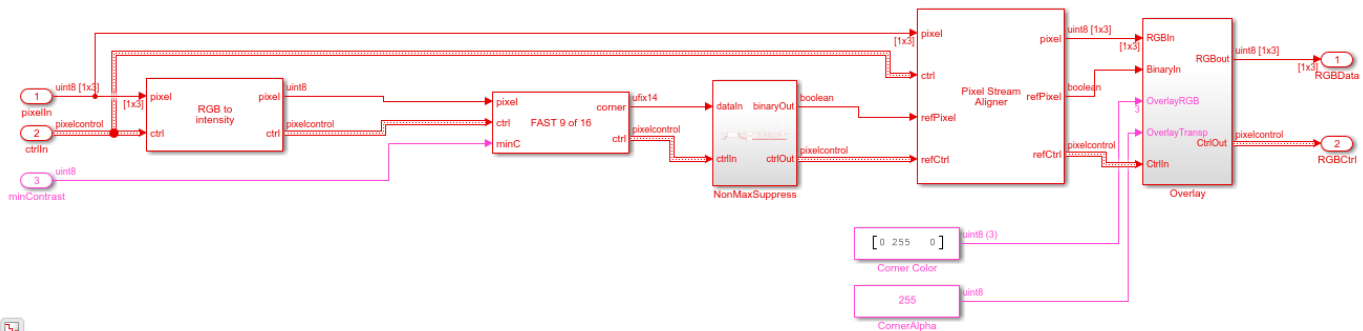

Corner Detection

To determine the presence of a corner, look for all possible 9-pixel contiguous segments of the ring that have values either greater than or less than the threshold value.

In hardware, you can perform all these comparisons in parallel. Each comparator block expands to 16 comparators. The output of the block is 16 binary decisions representing each segment of the ring.

Non-Maximal Suppression

The FAST algorithm identifies many, many potential corners. To reduce subsequent processing, all corners except the corners with the maximum corner metric in a particular region can be removed or suppressed. There are many algorithms for non-maximal suppression suitable for software implementation, but few suitable for hardware. In software, a gradient-based approach is used, which can be resource intensive in hardware. In this model a simple but very effective technique is to compare corner metrics in a 5x5 kernel and produce a boolean result. The boolean output is true if

the corner metric in the center of the kernel is greater than zero (i.e. it is a corner) and also it is the maximum of all the other corner metrics in the 5x5 region. The greater-than-zero condition matches setting `minQuality` to 1 for the `detectFASTFeatures` function.

Since the processing of the pixel stream is from left to right and top to bottom, the results contain some directional effects, such as that the detected corners do not always perfectly align with the objects. The NonMaxSuppress subsystem includes a constant block that allows you to disable suppression and visualize the complete results.

```
open_system([modelname '/FASTHDLAlgorithm/NonMaxSuppress'],'force');
```
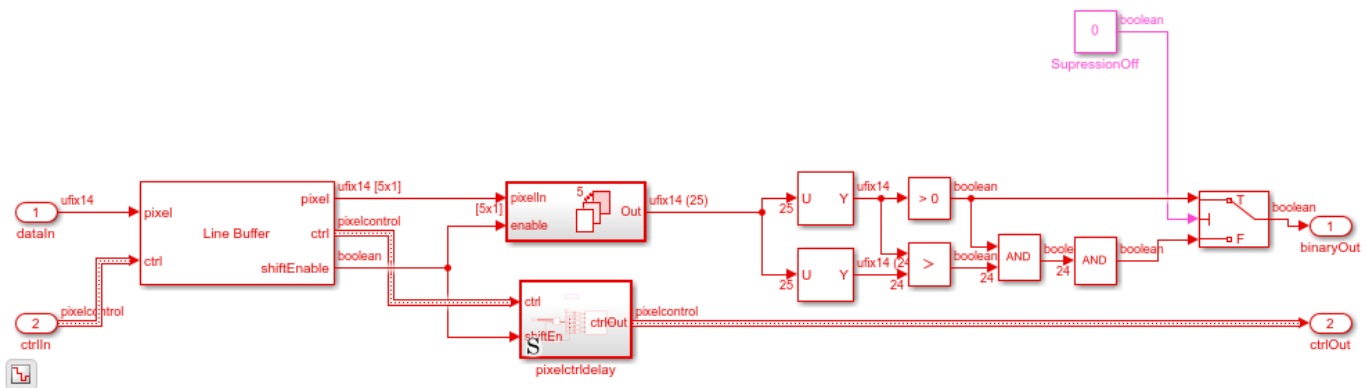

Align and Overlay

At the output of the NonMaxSuppress subsystem, the pixel stream includes markers for the strongest corner in each 5x5 region. Next, the model realigns the detected corners with the original pixel stream using the Pixel Stream Aligner block. After the original stream and the markers are aligned in time, the model overlays a green dot on the corners. The Overlay subsystem contains an alpha mixer with constants for the color and alpha values.

The output viewers show the overlaid green dots for corners detected. The Behavioral Video Viewer shows the output of the `detectFastFeatures` function, and the HDL Video Viewer shows the output of the HDL algorithm.
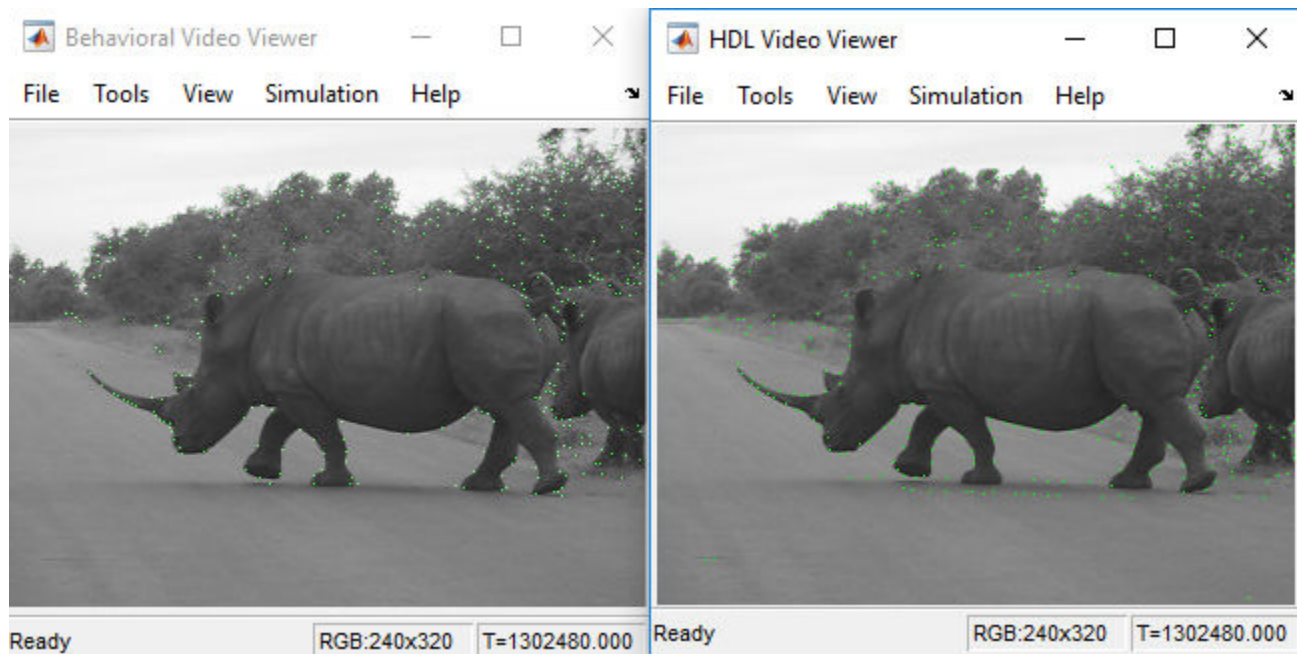
Going Further

The non-maximal suppression algorithm could be improved by following gradients and using a multiple-pass strategy, but that computation would also use more hardware resources.

Conclusion

This example shows how to start using `detectFASTFeatures` in MATLAB and then move to Simulink for the FPGA portion of the design. The hardware algorithm in the Corner Detector block includes a test of the ring around the central pixel in a kernel, and a corner strength metric. The model uses a non-maximal suppression function to remove all but the strongest detected corners. The design then overlays the corner locations onto the original video input, highlighting the corners in green.

References

[1] Rosten, E., and T. Drummond. "Fusing Points and Lines for High Performance Tracking" Proceedings of the IEEE International Conference on Computer Vision, Vol. 2 (October 2005): pp. 1508-1511.

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

Lane Detection

This example shows FPGA acceleration of lane marking detection.

Lane detection is a critical processing stage in Advanced Driving Assistance Systems (ADAS). Automatically detecting lane boundaries from a video stream is computationally challenging and therefore hardware accelerators such as FPGAs and GPUs are often required to achieve real time performance.

In this example model, an FPGA-based lane candidate generator is coupled with a software-based polynomial fitting engine, to determine lane boundaries.

System Overview

The LaneDetectionHDL.slx system is shown below. The HDLLaneDetector subsystem represents the hardware accelerated part of the design, while the SWLaneFitandOverlay subsystem represent the software based polynomial fitting engine. Prior to the Frame to Pixels block, the RGB input is converted to intensity color space.
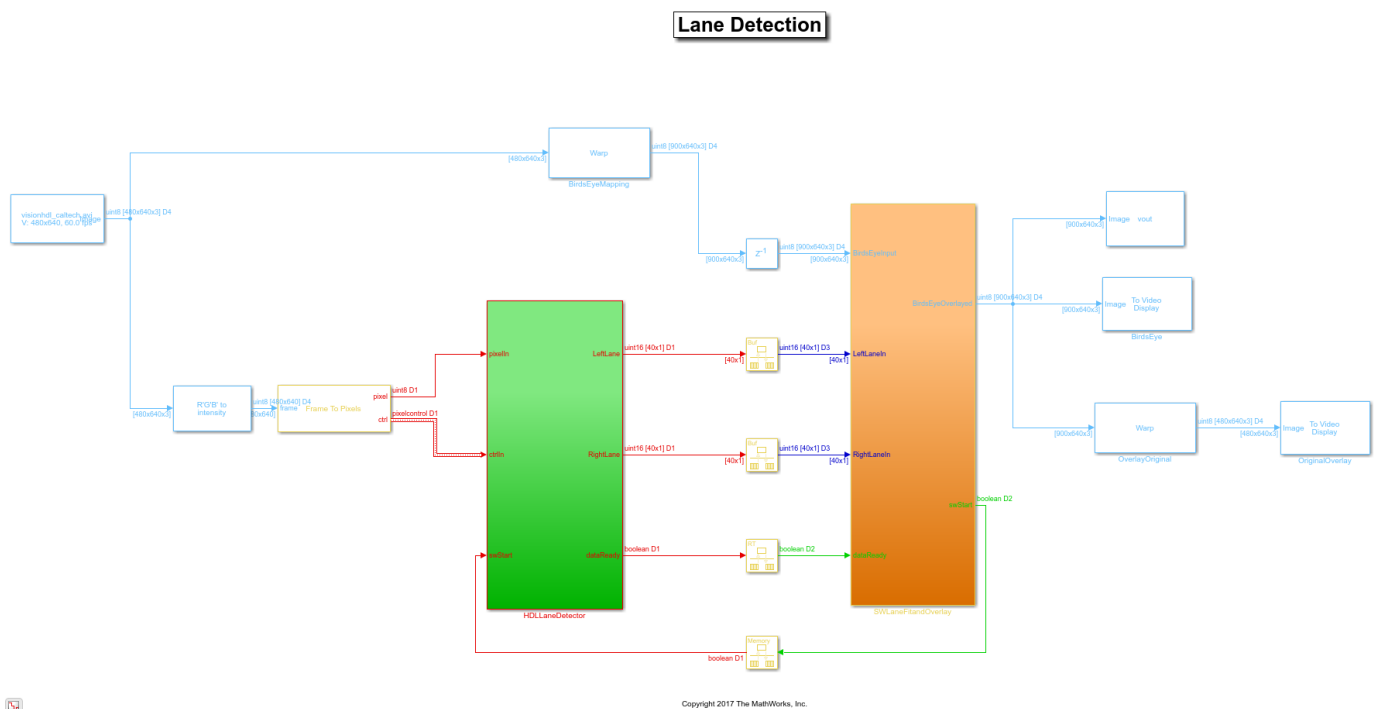
HDL Lane Detector

The HDL Lane Detector represents the hardware-accelerated part of the design. This subsystem receives the input pixel stream from the front-facing camera source, transforms the view to obtain the birds-eye view, locates lane marking candidates from the transformed view and then buffers them up into a vector to send to the software side for curve fitting and overlay.
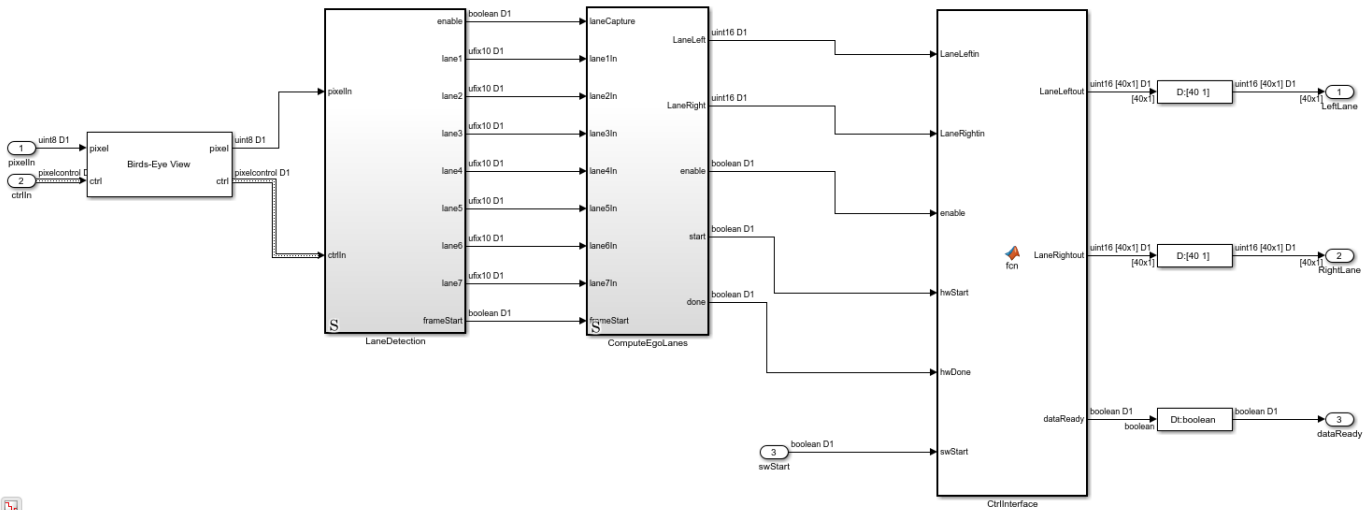
Birds-Eye View

The Birds-Eye View block transforms the front-facing camera view to a birds-eye perspective. Working with the images in this view simplifies the processing requirements of the downstream lane detection algorithms. The front-facing view suffers from perspective distortion, causing the lanes to converge at the vanishing point. The first stage of the system corrects the perspective distortion by transforming to the birds-eye view

The Inverse Perspective Mapping is given by the following expression:

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

The homography matrix, **h**, is derived from four intrinsic parameters of the physical camera setup, namely the focal length, pitch, height and principle point (from a pinhole camera model). Please refer to Computer Vision System Toolbox™ documentation for further details.

Direct evaluation of the source (front-facing) to destination (birds-eye) mapping in real time on FPGA/ASIC hardware is challenging. The requirement for division along with the potential for non-sequential memory access from a frame buffer mean that the computational requirements of this part of the design are substantial. Therefore instead of directly evaluating the IPM calculation in real time, an offline analysis of the input to output mapping has been performed and used to pre-compute a mapping scheme. This is possible as the homography matrix is fixed after factory calibration/installation of the camera, due to the camera position, height and pitch being fixed.

In this particular example, the birds-eye output image is a frame of [700x640] dimensions, whereas the front-facing input image is of [480x640] dimensions. There is not sufficient blanking available in order to output the full birds-eye frame before the next front-facing camera input is streamed in. Internally, the Birds-Eye view block will therefore lock up when it begins to process a new input frame, and will not accept new frame data until it has finished outputting the current birds-eye frame.

Line Buffering and Address Computation

A full sized projective transformation from input to output would result in a [900x640] output image. This requires that the full [480x640] input image is stored in memory, while the source pixel location is calculated using the source location and homography matrix. Ideally on-chip memory should be

used for this purpose, removing the requirement for an off-chip frame buffer. Analysis of the mapping of input line to output line reveals that in order to generate the first 700 lines of the top down birds eye output image, around 50 lines of the input image are required. This is an acceptable number of lines to store using on-chip memory.

Lane Detection

With the birds-eye view image obtained, the actual lane detection can be performed. There are many techniques which can be considered for this purpose. To achieve an implementation which is robust, works well on streaming image data and which can be implemented in FPGA/ASIC hardware at reasonable resource cost, this example uses the approach described in [1]. This algorithm performs a full image convolution with a vertically oriented first order Gaussian derivative filter kernel, followed by sub-region processing.
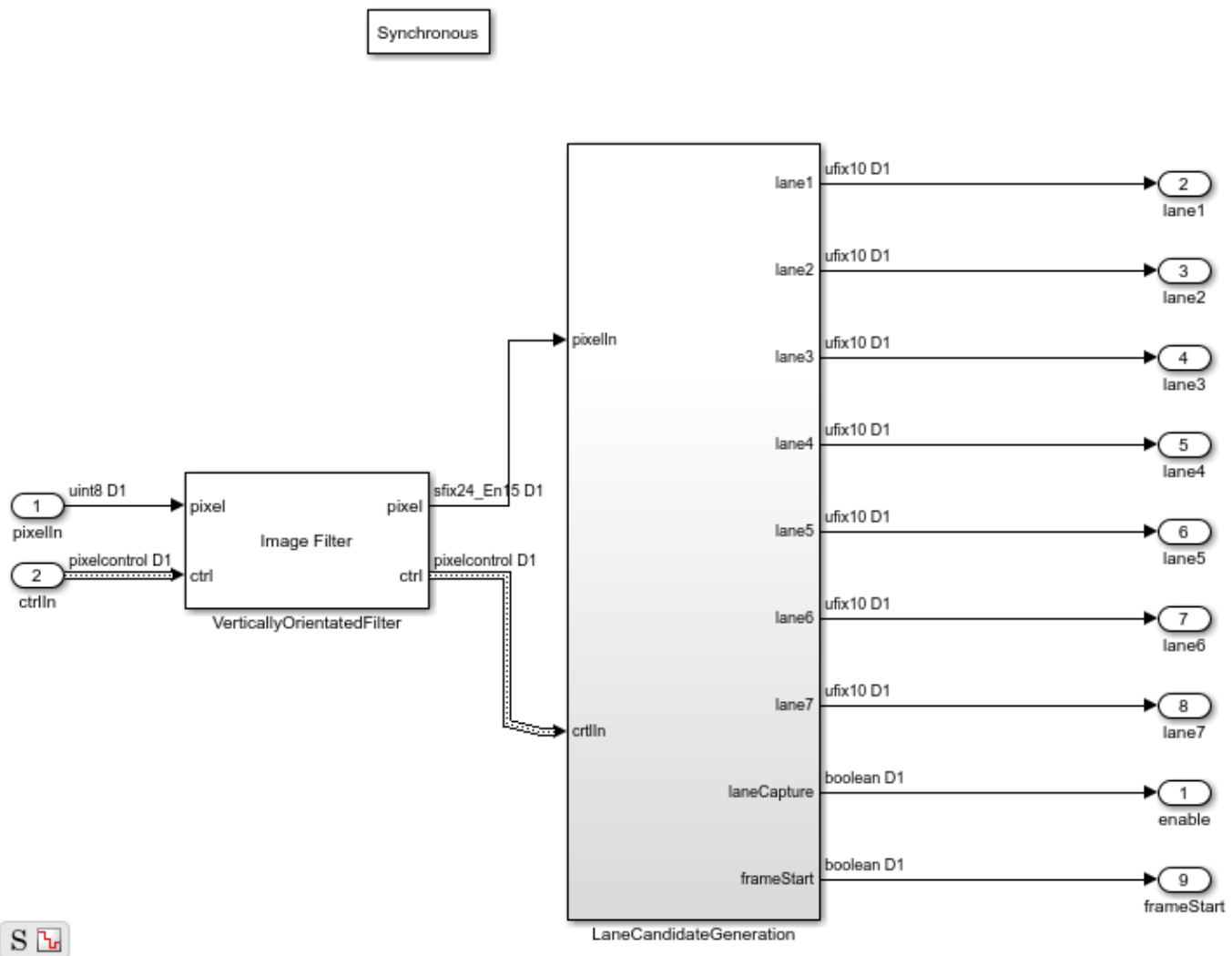
Vertically Oriented Filter Convolution

Immediately following the birds-eye mapping of the input image, the output is convolved with a filter designed to locate strips of high intensity pixels on a dark background. The width of the kernel is 8

pixels, which relates to the width of the lines that appear in the birds-eye image. The height is set to 16 which relates to the size of the dashed lane markings which appear in the image. As the birds-eye image is physically related to the height, pitch etc. of the camera, the width at which lanes appear in this image is intrinsically related to the physical measurement on the road. The width and height of the kernel may need to be updated when operating the lane detection system in different countries.
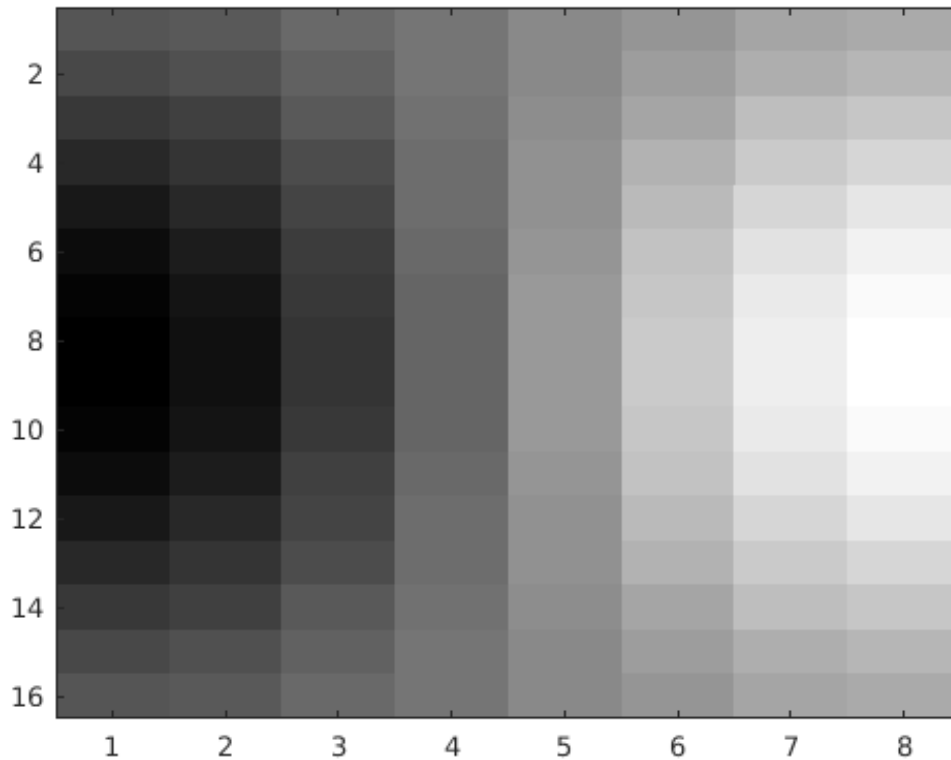
The output of the filter kernel is shown below, using jet colormap to highlight differences in intensity. Because the filter kernel is a general, vertically oriented Gaussian derivative, there is some response from many different regions. However, for the locations where a lane marking is present, there is a strong positive response located next to a strong negative response, which is consistent across columns. This characteristic of the filter output is used in the next stage of the detection algorithm to locate valid lane candidates.
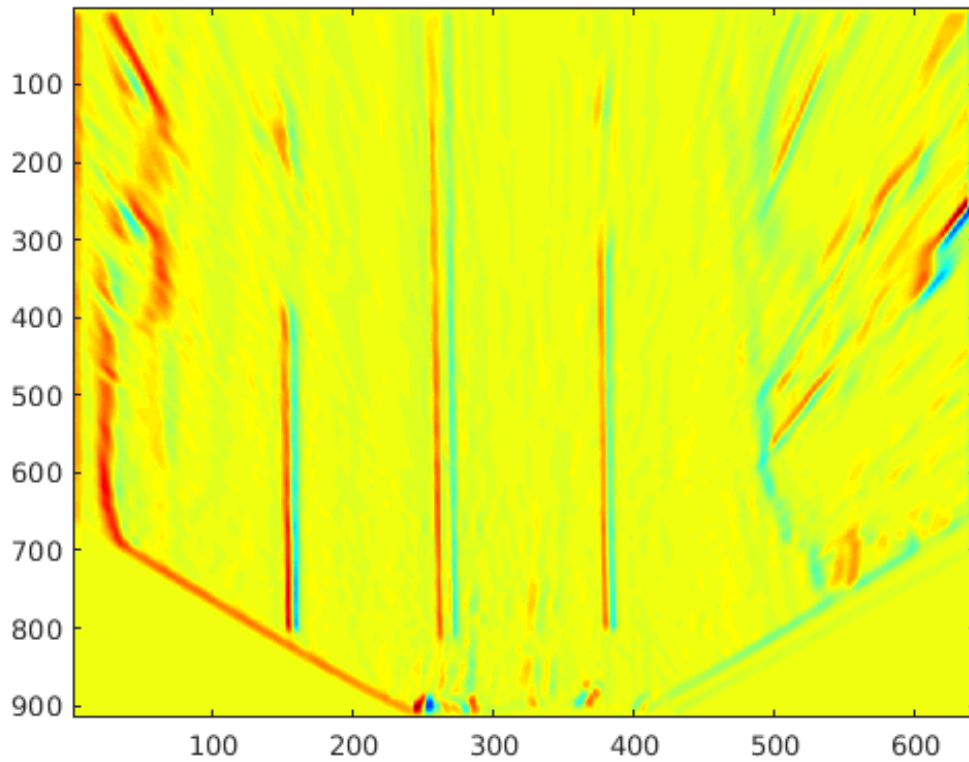
Lane Candidate Generation

After convolution with the Gaussian derivative kernel, sub-region processing of the output is performed in order to find the coordinates where a lane marking is present. Each region consists of 18 lines, with a ping-pong memory scheme in place to ensure that data can be continuously streamed through the subsystem.
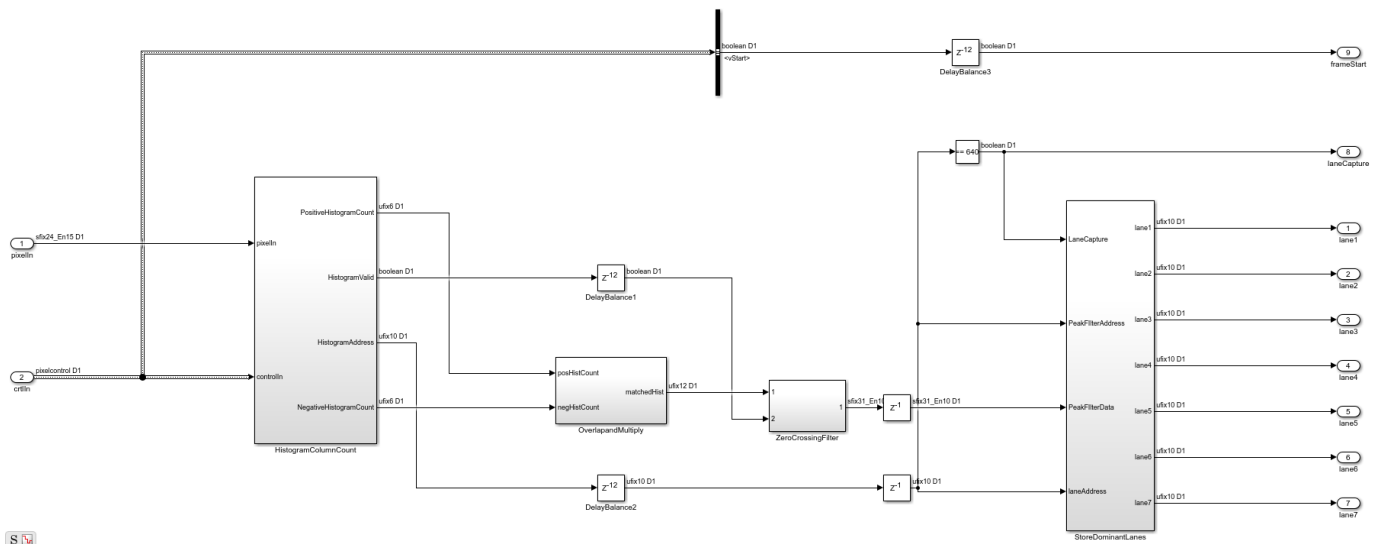
Histogram Column Count

Firstly, HistogramColumnCount counts the number of thresholded pixels in each column over the 18 line region. A high column count indicates that a lane is likely present in the region. This count is performed for both the positive and the negative thresholded images. The positive histogram counts are offset to account for the kernel width. Lane candidates occur where the positive count and negative counts are both high. This exploits the previously noted property of the convolution output where positive tracks appear next to negative tracks.

Internally, the column counting histogram generates the control signalling that selects an 18 line region, computes the column histogram, and outputs the result when ready. A ping-pong buffering scheme is in place which allows one histogram to be reading while the next is writing.

Overlap and Multiply

As noted, when a lane is present in the birds-eye image, the convolution result will produce strips of high-intensity positive output located next to strips of high-intensity negative output. The positive and negative column count histograms locate such regions. In order to amplify these locations, the positive count output is delayed by 8 clock cycles (an intrinsic parameter related to the kernel width), and the positive and negative counts are multiplied together. This amplifies columns where the positive and negative counts are in agreement, and minimizes regions where there is disagreement between the positive and negative counts. The design is pipelined in order to ensure high throughput operation.
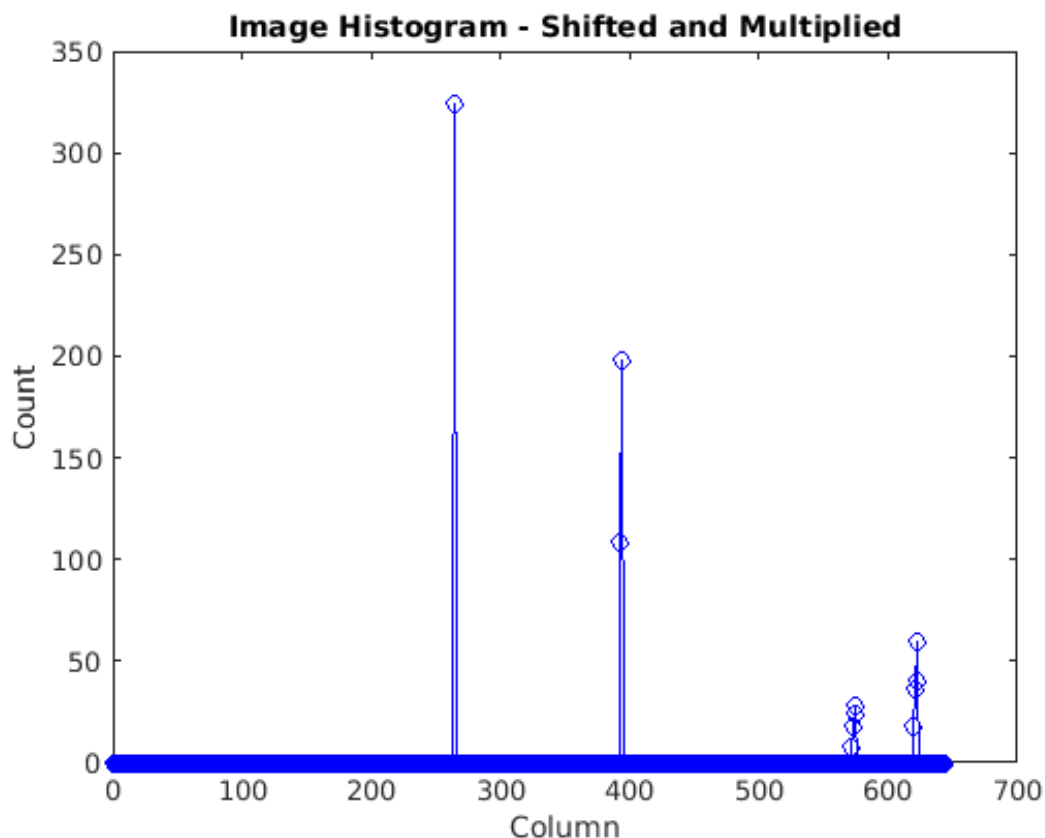
Zero Crossing Filter

At the output of the Overlap and Multiply subsystem, peaks appear where there are lane markings present. A peak detection algorithm determines the columns where lane markings are present. Because the SNR is relatively high in the data, this example uses a simple FIR filtering operation followed by zero crossing detection. The Zero Crossing Filter is implemented using the Discrete FIR Filter block from DSP System Toolbox™. It is pipelined for high-throughput operation.
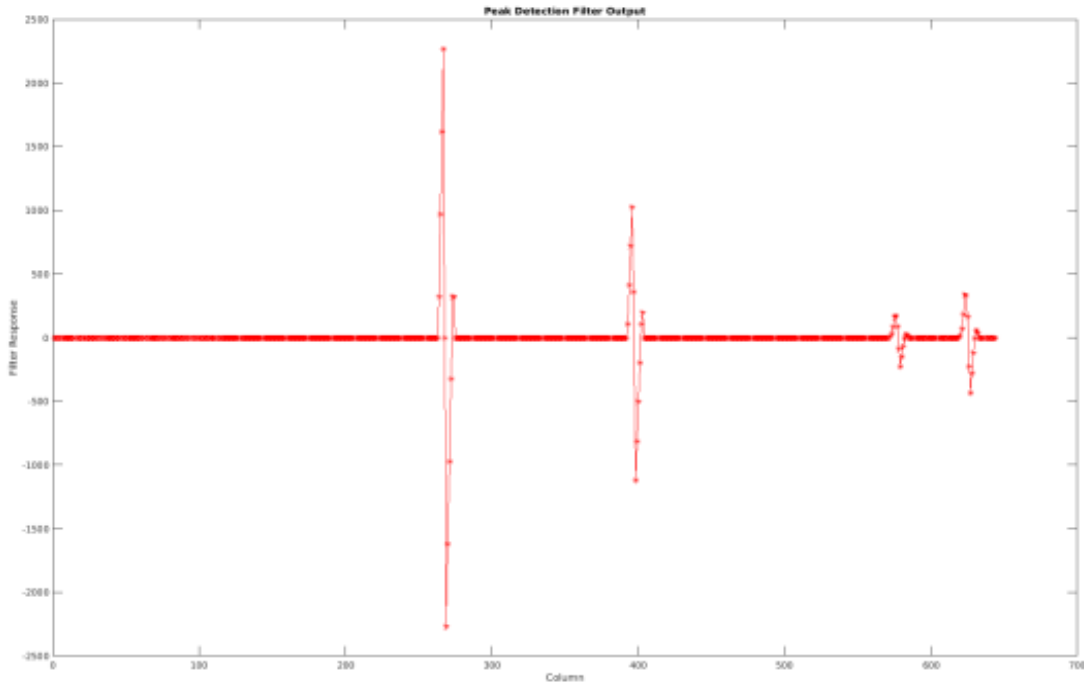
Store Dominant Lanes

The zero crossing filter output is then passed into the Store Dominant Lanes subsystem. This subsystem has a maximum memory of 7 entries, and is reset every time a new batch of 18 lines is reached. Therefore, for each sub-region 7 potential lane candidates are generated. In this subsystem, the Zero Crossing Filter output is streamed through, and examined for potential zero crossings. If a zero crossing does occur, then the difference between the address immediately prior to zero crossing and the address after zero crossing is taken in order to get a measurement of the size of the peak. The subsystem stores the zero crossing locations with the highest magnitude.
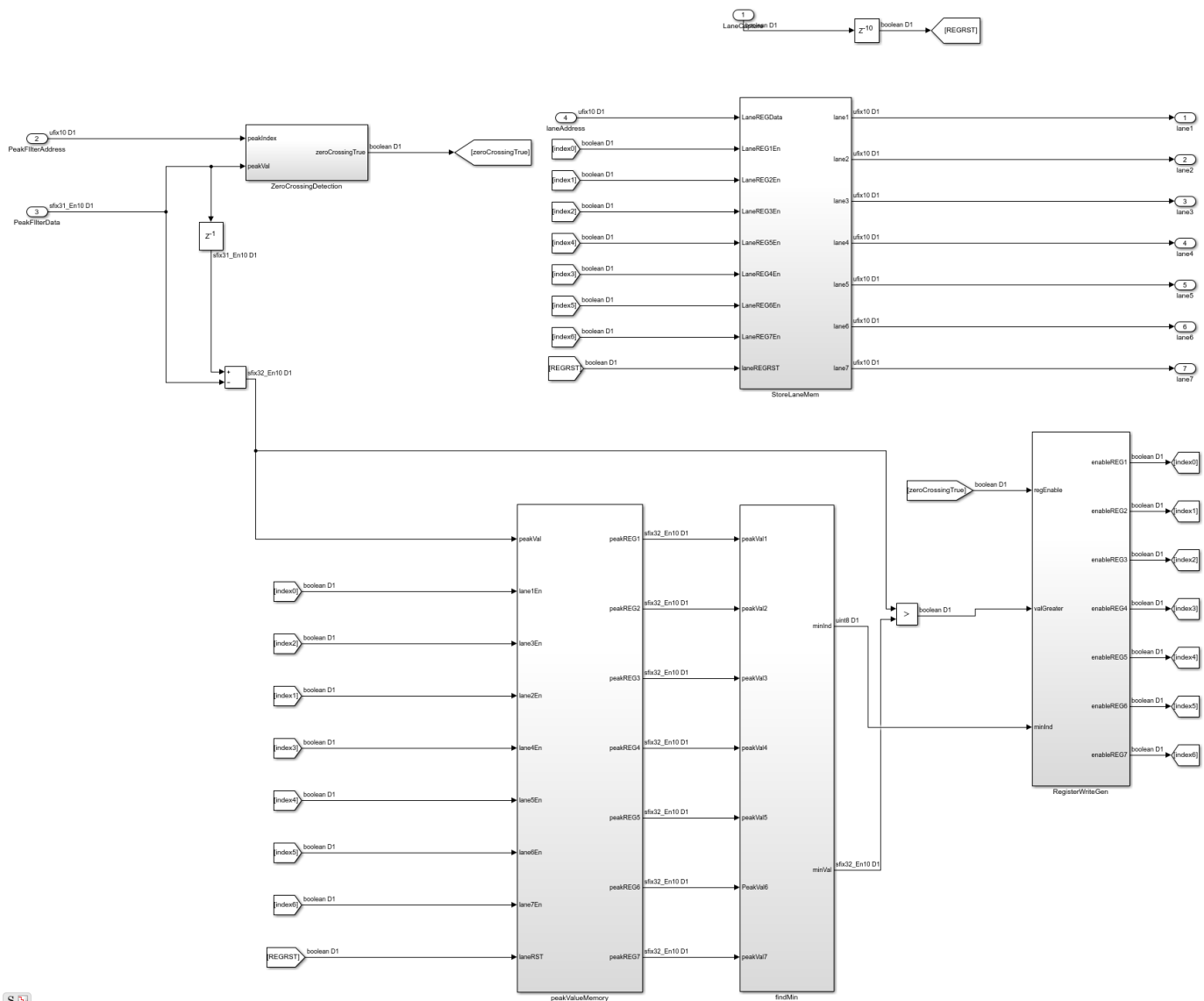
Compute Ego Lanes

The Lane Detection subsystem outputs the 7 most viable lane markings. In many applications, we are most interested in the lane markings that contain the lane in which the vehicle is driving. By computing the so called "Ego-Lanes" on the hardware side of the design, we can reduce the memory bandwidth between hardware and software, by sending 2 lanes rather than 7 to the processor. The Ego-Lane computation is split into two subsystems. The FirstPassEgoLane subsystem assumes that the centre column of the image corresponds to the middle of the lane, when the vehicle is correctly operating within the lane boundaries. The lane candidates which are closest to the center are therefore assumed as the ego lanes. The Outlier Removal subsystem maintains an average width of the distance from lane markings to centre coordinate. Lane markers which are not within tolerance of the current width are rejected. Performing early rejection of lane markers gives better results when performing curve fitting later on in the design.
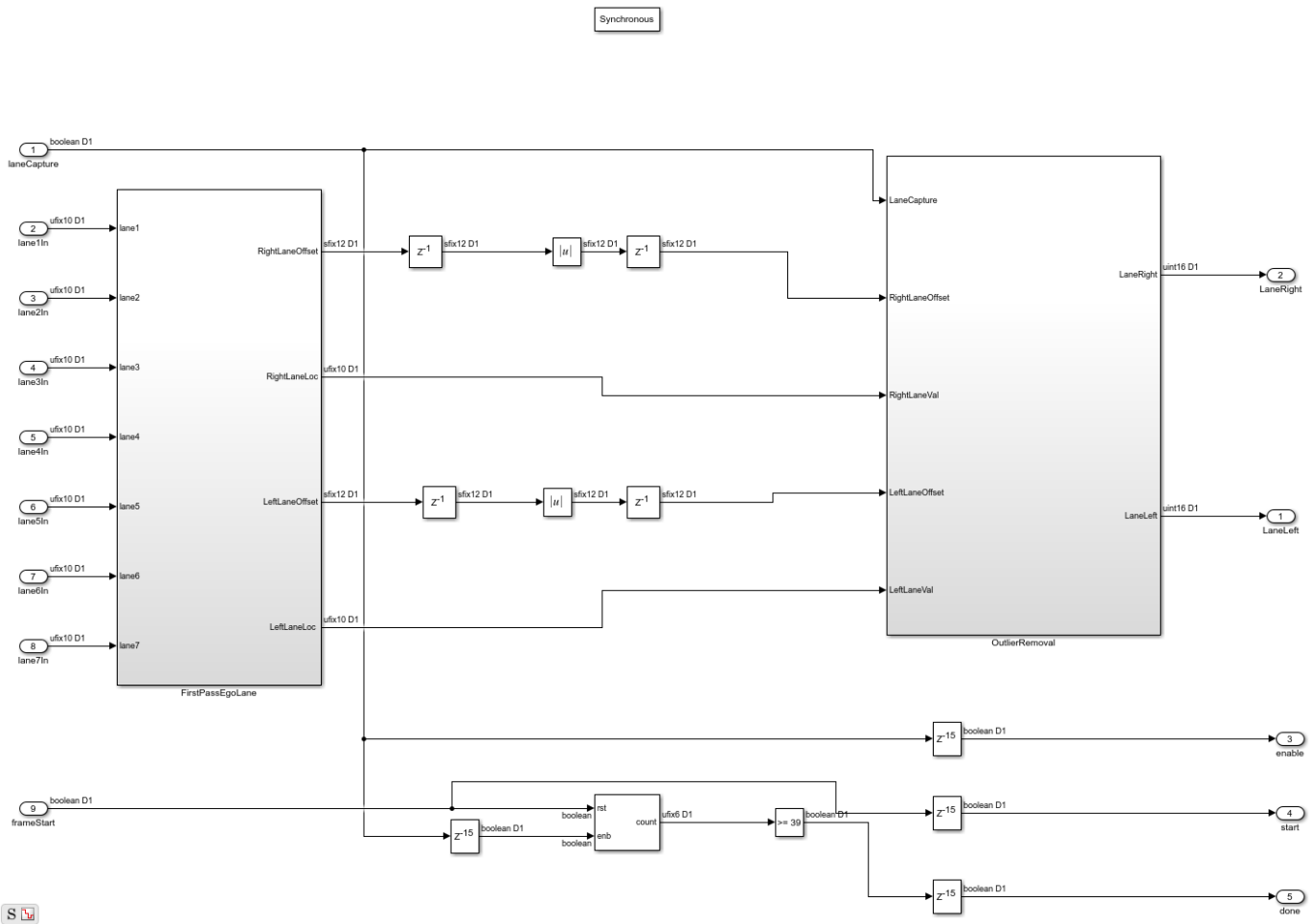
Control Interface

Finally, the computed ego lanes are sent to the CtrlInterface MATLAB function subsystem. This state machine uses the four control signal inputs - enable, hwStart, hwDone, and swStart to determine when to start buffering, accept new lane coordinate into the 40x1 buffer and finally indicate to the software that all 40 lane coordinates have been buffered and so the lane fitting and overlay can be performed. The dataReady signal ensures that software will not attempt lane fitting until all 40 coordinates have been buffered, while the swStart signal ensures that the current set of 40 coordinates will be held until lane fitting is completed.

Software Lane Fit and Overlay

The detected ego-lanes are then passed to the SW Lane Fit and Overlay subsystem, where robust curve fitting and overlay is performed. Recall that the birds-eye output is produced once every two frames or so rather than on every consecutive frame. The curve fitting and overlay is therefore placed in an enabled subsystem, which is only enabled when new ego lanes are produced.
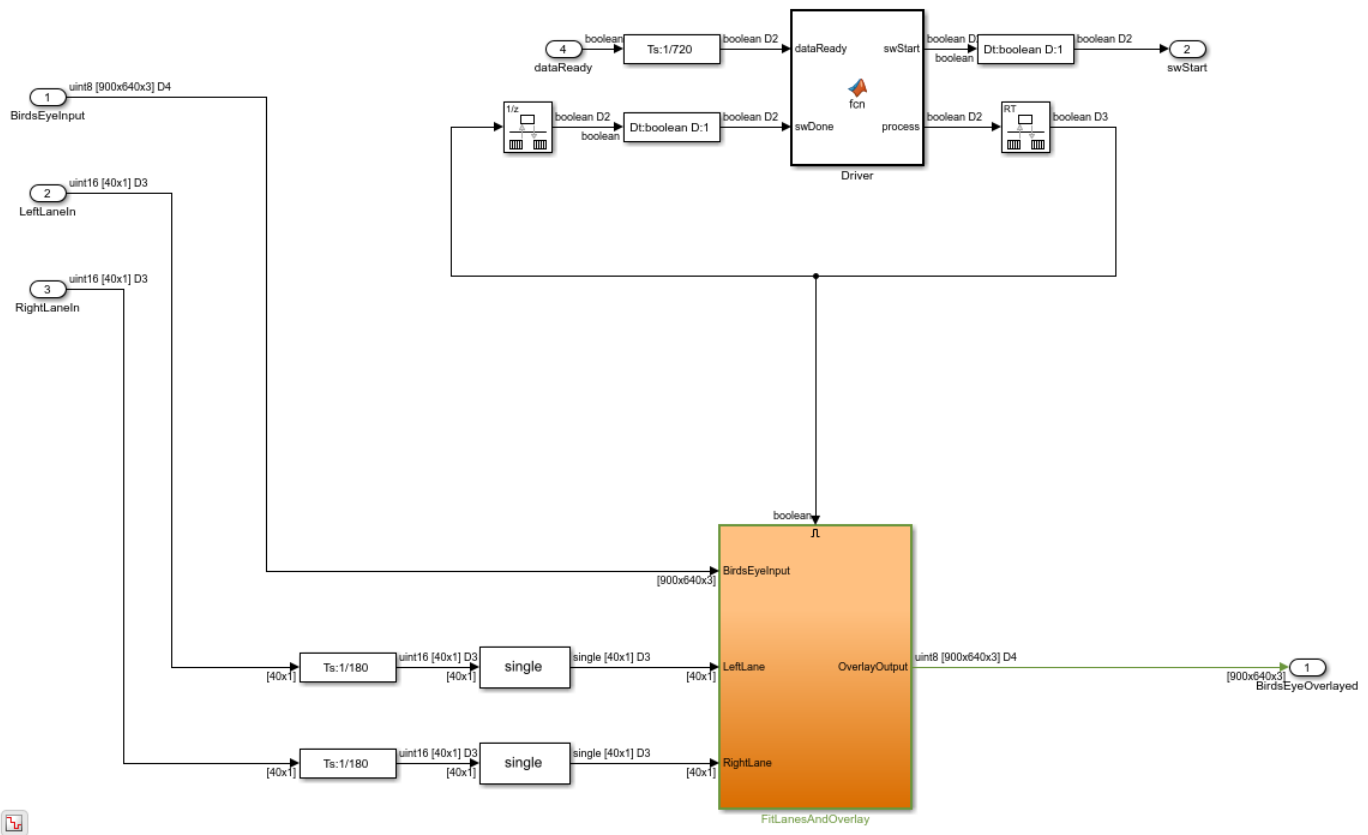
Driver

The Driver MATLAB Function subsystem controls the synchronization between hardware and software. Initially it is in a polling state, where it samples the dataReady input at regular intervals per frame to determine when hardware has buffered a full [40x1] vector of lane coordinates. Once this occurs, it transitions into software processing state where swStart and process outputs are held high. The Driver remains in the software processing state until swDone input is high. Seeing as the process output loops back to swDone input with a rate transition block in between, there is effectively a constant time budget specified for the FitLanesandOverlay subsystem to perform the fitting and overlay. When swDone is high, the Driver will transition into a synchronization state, where swStart is held low to indicate to hardware that processing is complete. The synchronization between software and hardware is such that hardware will hold the [40x1] vector of lane coordinates until the swStart signal transitions back to low. When this occurs, dataReady output of hardware will then transition back to low.

Fit Lanes and Overlay

The Fit Lanes and Overlay subsystem is enabled by the Driver. It performs the necessary arithmetic required in order to fit a polynomial onto the lane coordinate data received at input, and then draws the fitted lane and lane coordinates onto the Birds-Eye image.

Fit Lanes

The Fit Lanes subsystem runs a RANSAC based line-fitting routine on the generated lane candidates. RANSAC is an iterative algorithm which builds up a table of inliers based on a distance measure

between the proposed curve, and the input data. At the output of this subsystem, there is a [3x1] vector which specifies the polynomial coefficients found by the RANSAC routine.

Overlay Lane Markings

The Overlay Lane Markings subsystem performs image visualization operations. It overlays the ego lanes and curves found by the lane-fitting routine.
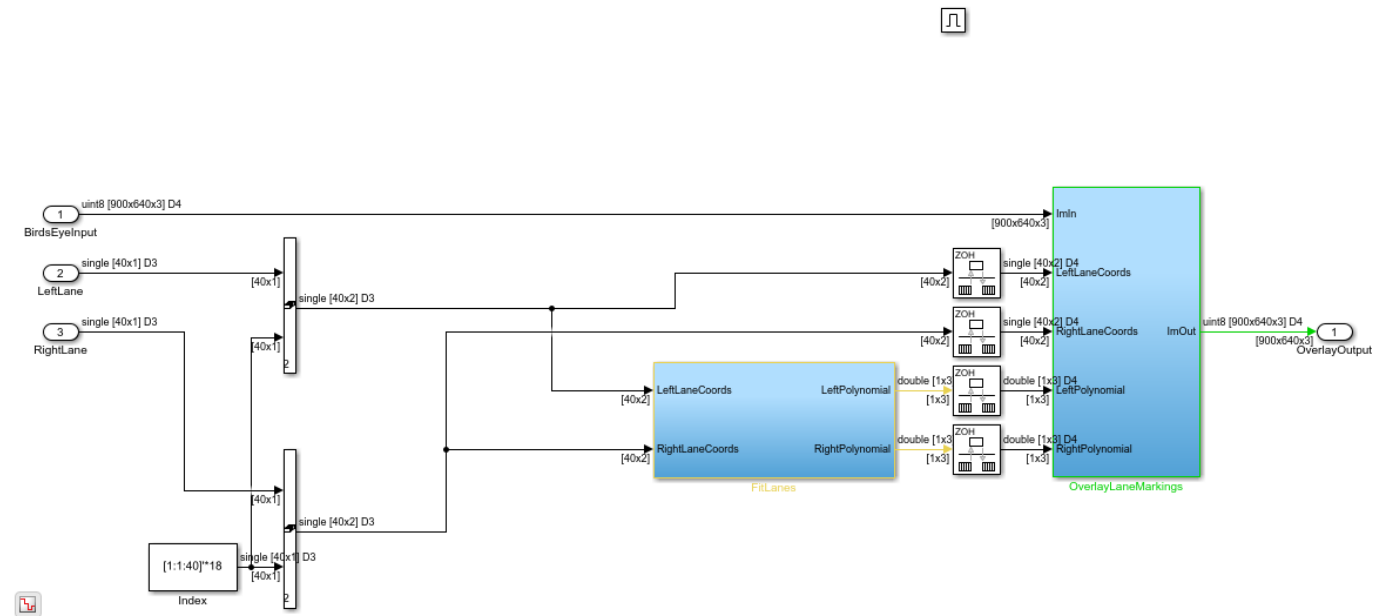
Results of the Simulation

The model includes two video displays shown at the output of the simulation results. The **BirdsEye** display shows the output in the warped perspective after lane candidates have been overlaid, polynomial fitting has been performed and the resulting polynomial overlaid onto the image. The **OriginalOverlay** display shows the **BirdsEye** output warped back into the original perspective.

Due to the large frame sizes used in this model, simulation can take a relatively long time to complete. If you have an HDL Verifier™ license, you can accelerate simulation speed by directly running the HDL Lane Detector subsystem in hardware using FPGA in the Loop (TM).
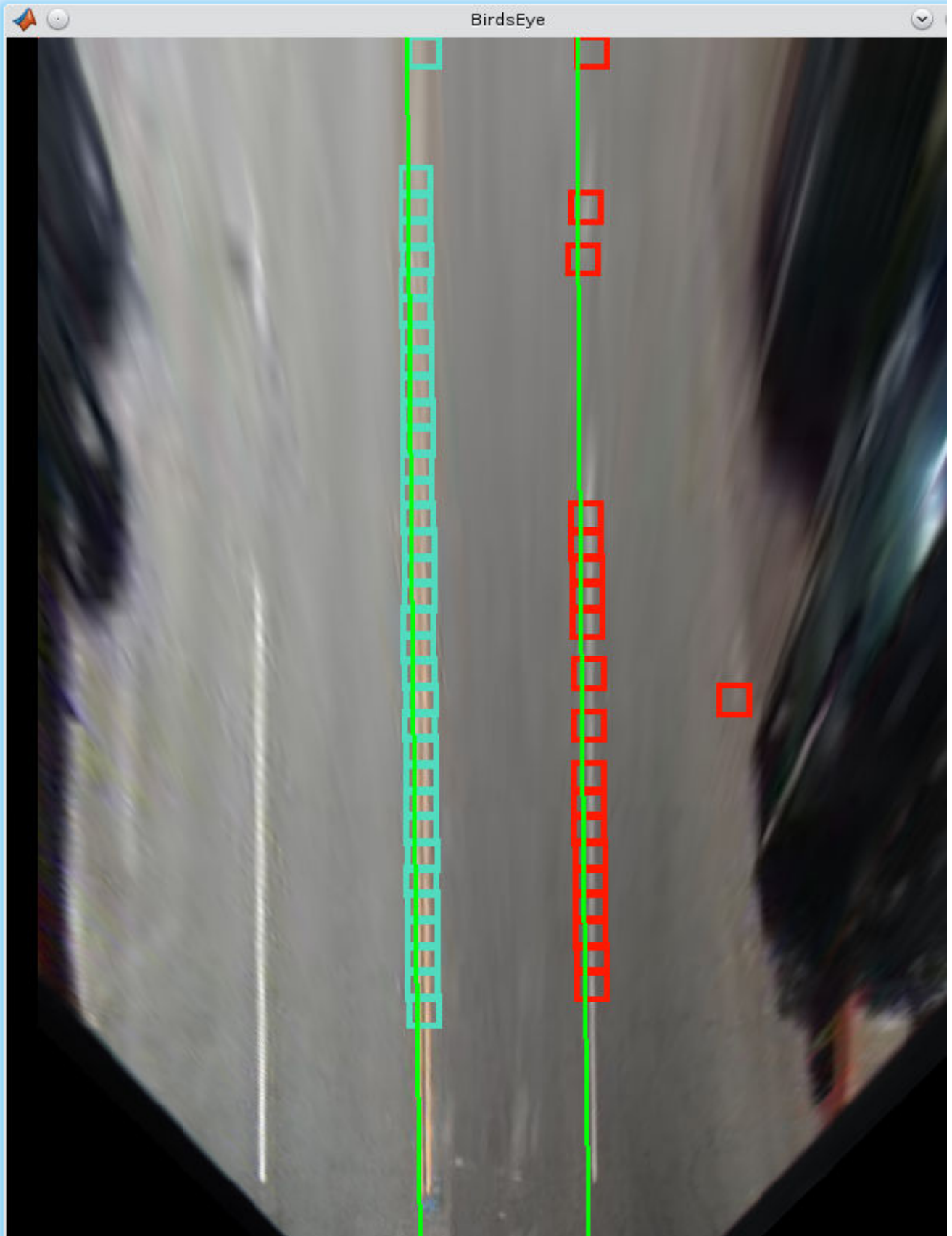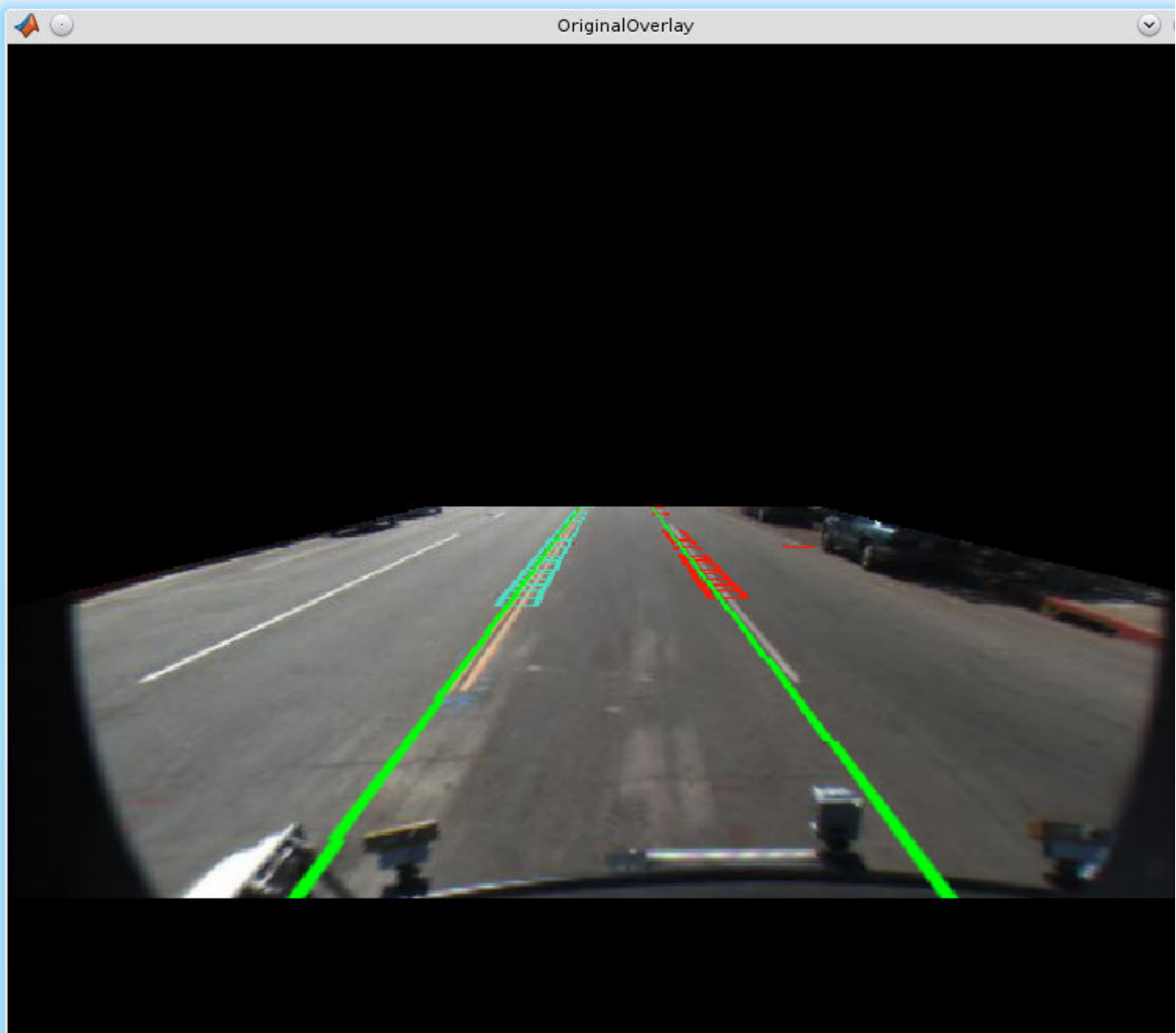
HDL Code Generation

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('LaneDetectionHDL/HDLLaneDetector')
```

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

```
makehdltb('LaneDetectionHDL/HDLLaneDetector')
```

For faster test bench simulation, you can generate a SystemVerilog DPIC test bench using the following command.

```
makehdltb('LaneDetectionHDL/HDLLaneDetector','GenerateSVDPITestBench','ModelSim')
```

Conclusion

This example has provided insight into the challenges of designing ADAS systems in general, with particular emphasis paid to the acceleration of critical parts of the design in hardware.

References

[1] R. K. Satzoda and Mohan M. Trivedi, "Vision based Lane Analysis: Exploration of Issues and Approaches for Embedded Realization", 2013 IEEE Conference on Computer Vision and Pattern Recognition.

[2] Video from Caltech Lanes Dataset - Mohamed Aly, "Real time Detection of Lane Markers in Urban Streets", 2008 IEEE Intelligent Vehicles Symposium - used with permission.

Generate Cartoon Images Using Bilateral Filtering

This example shows how to generate cartoon lines and overlay them onto an image.

Bilateral filtering [1] is used in computer vision systems to filter images while preserving edges and has become ubiquitous in image processing applications. Those applications include denoising while preserving edges, texture and illumination separation for segmentation, and cartooning or image abstraction to enhance edges in a quantized color-reduced image.

Bilateral filtering is simple in concept: each pixel at the center of a neighborhood is replaced by the average of its neighbors. The average is computed using a weighted set of coefficients. The weights are determined by the spatial location in the neighborhood (as in a traditional Gaussian blur filter), and the intensity difference from the center value of the neighborhood.

These two weighting factors are independently controllable by the two standard deviation parameters of the bilateral filter. When the intensity standard deviation is large, the bilateral filter acts more like a Gaussian blur filter, because the intensity Gaussian is less peaked. Conversely, when the intensity standard deviation is smaller, edges in the intensity are preserved or enhanced.

This example model provides a hardware-compatible algorithm. You can generate HDL code from this algorithm, and implement it on a board using a Xilinx™ Zynq™ reference design. See "Bilateral Filtering with Zynq-Based Hardware" (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

Introduction

The BilateralFilterHDLExample.slx system is shown here.

```
modelname = 'BilateralFilterHDLExample';
open_system(modelname);
set_param(modelname, 'SampleTimeColors', 'on');
set_param(modelname,'SimulationCommand','Update');
set_param(modelname, 'Open', 'on');
set(allchild(0),'Visible', 'off');
```
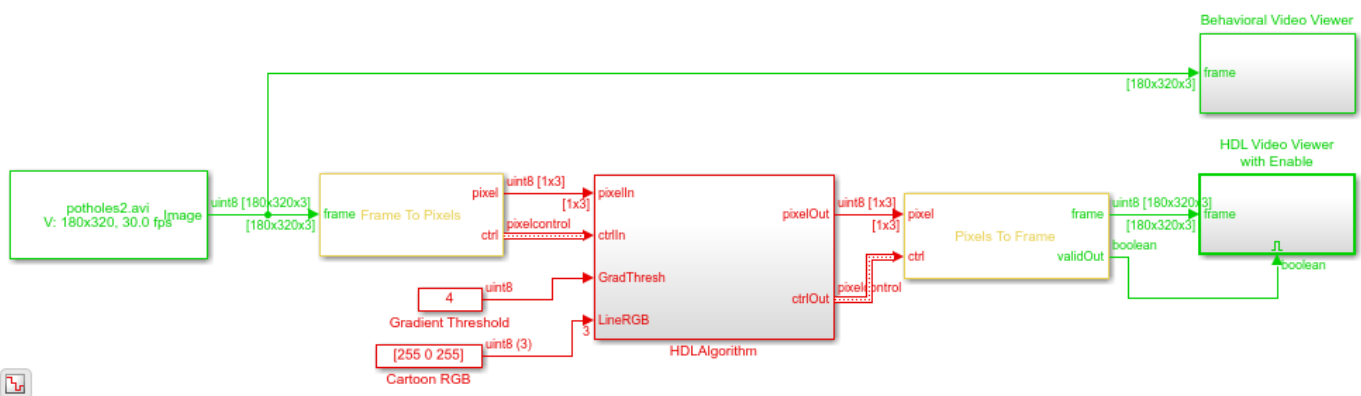

Step 1: Establish the Parameter Values

To achieve a modest Gaussian blur of the input, choose a relatively large spatial standard deviation of 3. To give strong emphasis to the edges of the image, choose an intensity standard deviation of 0.75. The intensity Gaussian is built from the image data in the neighborhood, so this plot represents the maximum possible values. Note the small vertical scale on the spatial Gaussian plot.

```
figure('units','normalized','outerposition',[0 0.5 0.75 0.45]);
subplot(1,2,1);
s1 = surf(fspecial('gaussian',[9 9 ],3));
subplot(1,2,2);
s2 = surf(fspecial('gaussian',[9 9 ],0.75));
legend(s1,'Spatial Gaussian 3.0');
legend(s2,'Intensity Gaussian 0.75');
```
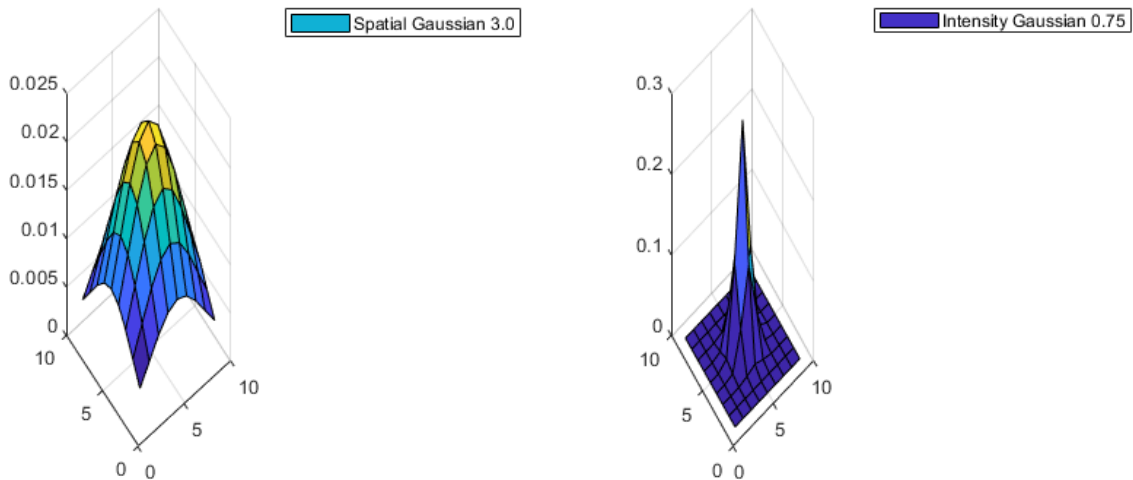
Fixed-Point Settings

For HDL code generation, you must choose a fixed-point data type for the filter coefficients. The coefficient type should be an unsigned type. For bilateral filtering, the input range is always assumed to be on the interval $[0, 1]$. Therefore, a `uint8` input with a range of values from $[0, 255]$ are treated as $\frac{[0, 255]}{255}$. The calculated coefficient values are less than 1. The exact values of the coefficients depend on the neighborhood size and the standard deviations. Larger neighborhoods spread the Gaussian function such that each coefficient value is smaller. A larger standard deviation flattens the Gaussian to produce more uniform values, while a smaller standard deviation produces a peaked response.

If you try a type and the coefficients are quantized such that more than half of the kernel becomes zero for all input, the `Bilateral Filter` block issues a warning. If all of the coefficients are zero after quantization, the block issues an error.

Step 2: Filter the Intensity Image

The model converts the incoming RGB image to intensity using the Color Space Converter block. Then the grayscale intensity image is sent to the Bilateral Filter block, which is configured for a 9-by-9 neighborhood and the parameters established previously.

The bilateral filter provides some Gaussian blur but will strongly emphasize larger edges in the image based on the 9-by-9 neighborhood size.

```
open_system([modelname '/HDLAlgorithm'],'force');
```
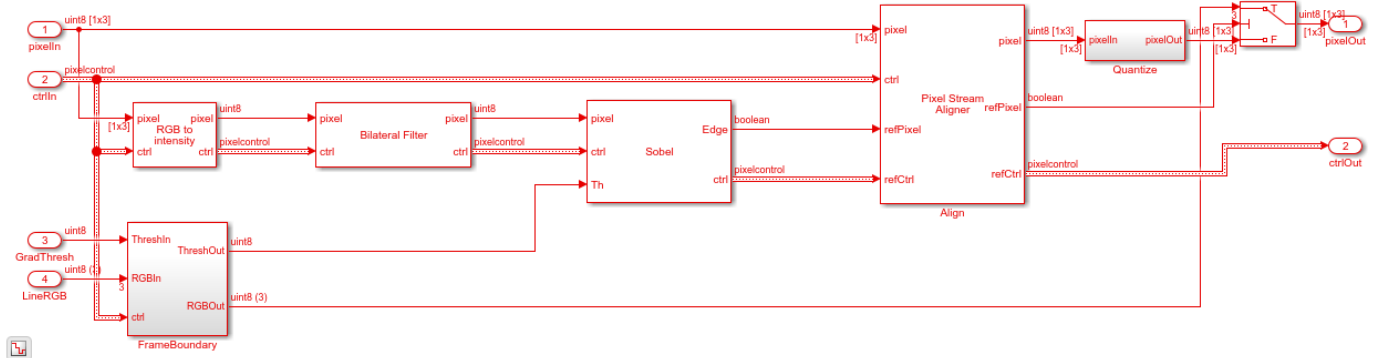
Step 3: Compute Gradient Magnitude

Next, the Sobel Edge Detector block computes the gradient magnitude. Since the image was pre-filtered using a bilateral filter with a fairly large neighborhood, the smaller, less important edges in the image will not be emphasized during edge detection.

The threshold parameter for the Sobel Edge Detector block can come from a constant value on the block mask or from a port. The block in this model uses port to allow the threshold to be set dynamically. This threshold value must be computed for your final system, but for now, you can just choose a good value by observing results.

Synchronize the Computed Edges

To overlay the thresholded edges onto the original RGB image, you must realign the two streams. The processing delay of the bilateral filter and edge detector means that the thresholded edge stream and the input RGB pixel stream are not aligned in time.

The Pixel Stream Aligner block brings them back together. The RGB pixel stream is connected to the upper pixel input port, and the binary threshold image pixel is connected to the reference input port. The block delays the RGB pixel stream to match the threshold stream.

You must set the number of lines parameter to a value that allows for the delay of both the bilateral filter and the edge detector. The 9-by-9 bilateral filter has a delay of more than 4 lines, while the edge detector has a delay of a bit more than 1 line. For safety, set the `Maximum number of lines` to 10 for now so that you can try different neighborhood sizes later. Once your design is done, you can determine the actual number of lines of delay by observing the control signal waveforms.

Color Quantization

Color quantization reduces the number of colors in an image to make processing it easier. Color quantization is primarily a clustering problem, because you want to find a single representative color for a cluster of colors in the original image.

For this problem, you can apply many different clustering algorithms, such as k-means or the median cut algorithm. Another common approach is using octrees, which recursively divide the color space into 8 octants. Normally you set a maximum depth of the tree, which controls the recursive subtrees that will be eliminated and therefore represented by one node in the subtree above.

These algorithms require that you know beforehand all of the colors in the original image. In pixel streaming video, the color discovery step introduces an undesirable frame delay. Color quantization is also generally best done in a perceptually uniform color space such as L*a*b. When you cluster colors in RGB space, there is no guarantee that the result will look representative to a human viewer.

The `Quantize` subsystem in this model uses a much simpler form of color quantization based on the most significant 4 bits of each 8-bit color component. RGB triples with 8-bit components can represent up to $2^{24} = 2^8 \cdot 2^8 \cdot 2^8$ colors but no single image can use all those colors. Similarly when you reduce the number of bits per color to 4, the image can contain up to $2^{12} = 2^4 \cdot 2^4 \cdot 2^4$ colors. In practice a 4-bit-per-color image typically contains only several hundred unique colors.

After shifting each color component to the right by 4 bits, the model shifts the result back to the left by 4 bits to maintain the 24-bit RGB format supported by the video viewer. In an HDL system, the next processing steps would pass on only the 4-bit color RGB triples.

```
open_system([modelname '/HDLAlgorithm/Quantize'],'force');
```
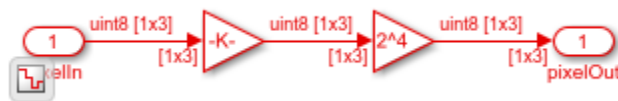

Overlay the Edges

A switch block overlays the edges on the original image by selecting either the RGB stream or an RGB parameter. The switch is flipped based on the edge-detected binary image. Because cartooning requires strong edges, the model does not use an alpha mixer.

Parameter Synchronization

In addition to the pixel and control signals, two parameters enter the HDLAlgorithm subsystem: the gradient threshold and the line RGB triple for the overlay color. The `FrameBoundary` subsystem provides run-time control of the threshold and the line color. However, to avoid an output frame with a mix of colors or thresholds, the subsystem registers the parameters only at the start of each frame.

```
open_system([modelname '/HDLAlgorithm/FrameBoundary'],'force');
```
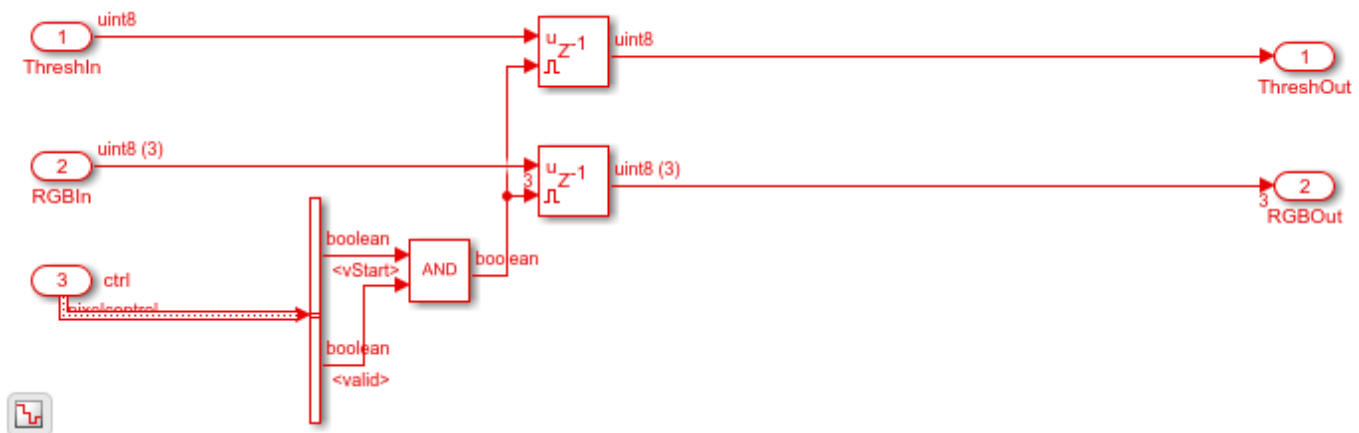

Simulation Results

After you run the simulation, you can see that the resulting images from the simulation show bold lines around the detected features in the input video.

HDL Code Generation

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('BilateralHDLExample/HDLAlgorithm')
```

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('BilateralHDLExample/HDLAlgorithm')
```

The part of the model between the Frame to Pixels and Pixels to Frame blocks can be implemented on an FPGA. The HDLAlgorithm subsystem includes all elements of the bilateral filter, edge detection, and overlay.

Going Further

The bilateral filter in this example is configured to emphasize larger edges while blurring smaller ones. To see the edge detection and overlay without bilateral filtering, right-click the Bilateral Filter block and select `Comment Through`. Then rerun the simulation. The updated results show that many smaller edges are detected and in general, the edges are much noisier.

This model has many parameters you can control, such as the bilateral filter standard deviations, the neighborhood size, and the threshold value. The neighborhood size controls the minimum width of emphasized edges. A smaller neighborhood results in more small edges being highlighted.

You can also control how the output looks by changing the RGB overlay color and the color quantization. Changing the edge detection threshold controls the strength of edges that are overlaid.

To further cartoon the image, you can try adding multiple bilateral filters. With a the right parameters, you can generate a very abstract image that is suitable for a variety of image segmentation algorithms.

Conclusion

This model generated a cartoon image using bilateral filtering and gradient generation. The model overlaid the cartoon lines on a version of the original RGB image that was quantized to a reduced number of colors. This algorithm is suitable for FPGA implementation.

References

[1] Tomasi, C., and R. Manducji. "Bilateral filtering for gray and color images." Sixth International Conference on Computer Vision, 1998.

Pothole Detection

This example extends the cartooning example to include calculating a centroid and overlaying a centroid marker and text label on detected potholes.

Road hazard or pothole detection is an important part of any automated driving system. Previous work [1] on automated pothole detection defined a pothole as an elliptical area in the road surface that has a darker brightness level and different texture than the surrounding road surface. Detecting potholes using image processing then becomes the task of finding regions in the image of the road surface that fit the chosen criterion. You can use any or all of the elliptical shape, darker brightness or texture criterion.

To measure the elliptical shape you can use a voting algorithm such as Hough circle, or a template matching algorithm, or linear algebra-based methods such as a least squares fit. Measuring the brightness level is simple in image processing by selecting a brightness segmentation value. The texture can be assessed by calculating the spatial frequency in a region using techniques such as the FFT.

This example uses brightness segmentation with an area metric so that smaller defects are not detected. To find the center of the defect, this design calculates the centroid. The model overlays a marker on the center of the defect and overlays a text label on the image.

Introduction

The PotHoleHDLDetector.slx system is shown below. The PotHoleHDL subsystem contains the pothole detector and overlay algorithms and supports HDL code generation. There are four input parameters that control the algorithm. The ProcessorBehavioral subsystem writes character maps into a RAM for use as overlay labels.

```
modelname = 'PotHoleHDLDetector';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
```
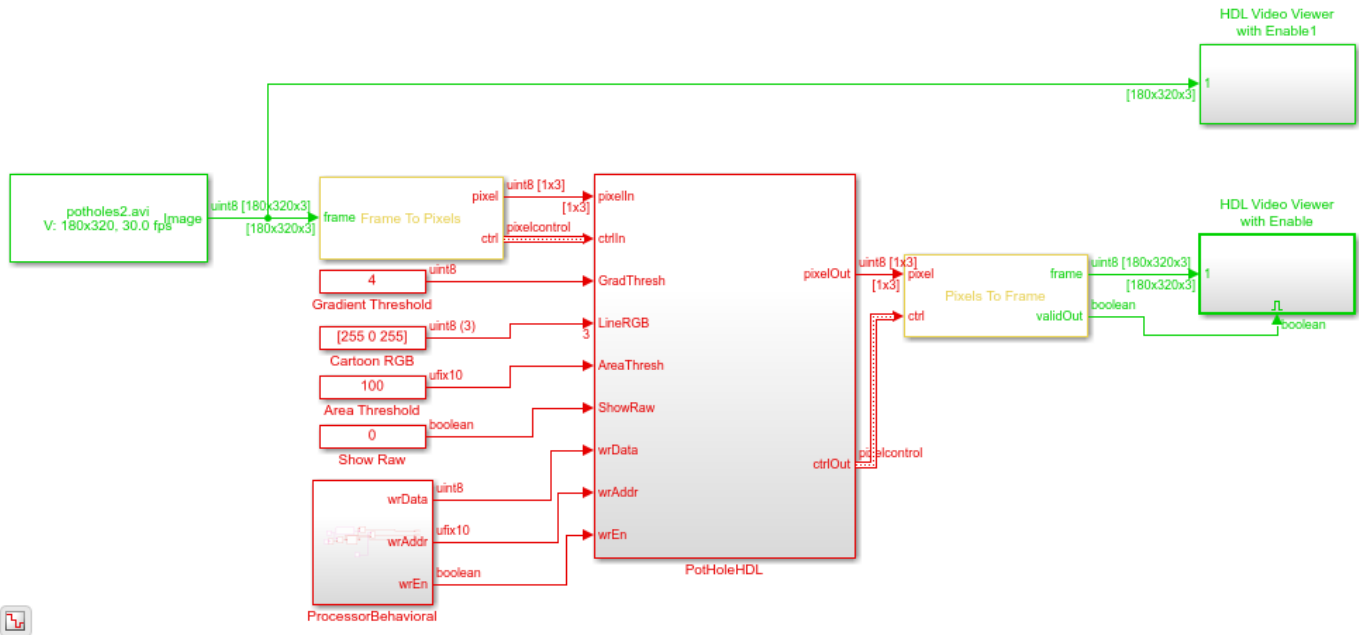
Overview of the FPGA Subsystem

The PotHoleHDL subsystem converts the RGB input video to intensity, then performs bilateral filtering and edge detection. The TrapezoidalMask subsystem selects the roadway area. Then the design applies a morphological close and calculates centroid coordinates for all potential potholes. The detector selects the largest pothole in each frame and saves the center coordinates. The Pixel Stream Aligner matches the timing of the coordinates with the input stream. Finally, the Fiducial31x31 and the Overlay32x32 subsystems apply alpha channel overlays on the frame to add a pothole center marker and a text label.

```
open_system([modelname '/PotHoleHDL'],'force');
```
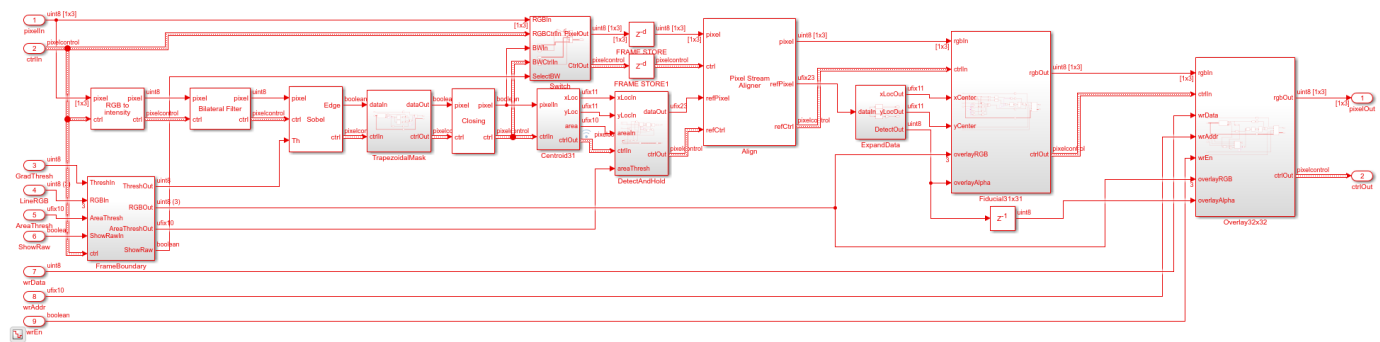

Input Parameter Values

The subsystem has four input parameters that can change while the system is running.

The gradient intensity parameter, Gradient Threshold, controls the edge detection part of the algorithm.

The Cartoon RGB parameter changes the color of the overlays, that is, the fiducial marker and the text.

The Area Threshold parameter sets the minimum number of marked pixels in the detection window in order for it to be classified as a pothole. If this value is too low, then linear cracks and other defects that are not road hazards will be detected. If it is too high then only the largest hazards will be detected.

The final parameter, Show Raw, allows you to debug the system more easily. It toggles the displayed image on which the overlays are drawn between the RGB input video and the binary image that the detector sees. Set this parameter to 1 to see how the detector is working.

All of these parameters work best if changes are only allowed on video frame boundaries. The FrameBoundary subsystem registers the parameters only on a valid start of frame.

```
open_system([modelname '/PotHoleHDL/FrameBoundary'],'force');
```
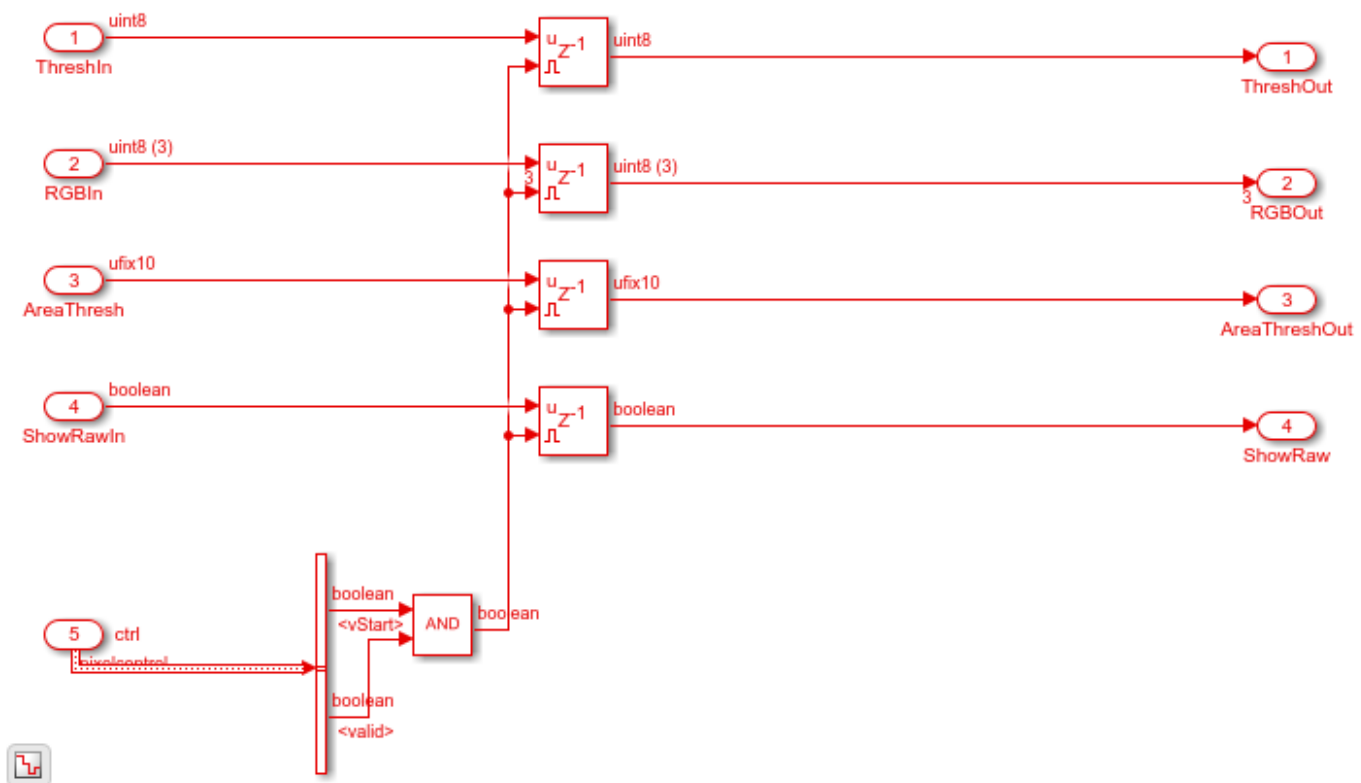

RGB to Intensity

The model splits the input RGB pixel stream so that a copy of the RGB stream continues toward the overlay blocks. The first step for the detector is to convert from RGB to intensity. Since the input data type for the RGB is `uint8`, the RGB to Intensity block automatically selects `uint8` as the output data type.

Bilateral Filter

The next step in the algorithm is to reduce high visual frequency noise and smaller road defects. There are many ways this can be accomplished but using a bilateral filter has the advantage of preserving edges while reducing the noise and smaller areas.

The Bilateral Filter block has parameters for the neighborhood size and two standard deviations, one for the spatial part of the filter and one for the intensity part of the filter. For this application a relatively large neighborhood of 9x9 works well. This model uses 3 and 0.75 for the standard deviations. You can experiment with these values later.

Sobel Edge Detection

The filtered image is then sent to the Sobel edge-detection block which finds the edges in the image and returns those edges that are stronger than the gradient threshold parameter. The output is a binary image. In your final application, this threshold can be set based on variables such as road conditions, weather, image brightness, etc. For this model, the threshold is an input parameter to the PotHoleHDL subsystem.

Trapezoidal Mask

From the binary edge image, you need to remove any edges that are not relevant to pothole detection. A good strategy is to use a mask that selects a polygonal region of interest and makes the area outside of that black. The model does not use a normal ROI block since that would remove the location context that you need later for the centroid calculation and labeling.

The order of operations also matters here because if you used the mask before edge detection, the edges of the mask would become strong lines that would result in false positives at the detector.

In the input video, the area in which the vehicle might encounter a pothole is limited to the roadway immediately in front of it and a trapezoidal section of roadway ahead. The exact coordinates depend on the camera mounting and lens. This example uses fixed coordinates for left-side top, right-side top, left-side bottom, and right-side bottom corners of the area. For this video, the top and bottom of the trapezoidal area are not parallel so this is not a true trapezoid.

The mask consists of straight lines between the corners, connecting left,right and top,bottom.

```
     ltc---rtc
    /         \
   /           \
  /             \
lbc------------rbc
```
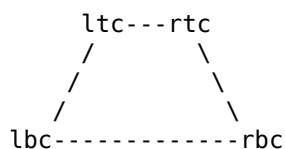
This example uses `polyfit` to determine a straight-line fit from corner to corner. For ease of implementation, the design calls `polyfit` with the vertical direction as the independent variable. This usage calculates x = f(y) instead of the more usual y = f(x). Using `polyfit` this way allows you to use a y-direction line counter as the input address of a lookup table of x-coordinates of the start (left) and end (right) of the area of interest on each line.

The lookup table is typically implemented in a BRAM in an FPGA, so it should be addressed with 0-based addressing. The model converts from MATLAB 1-based addressing to 0-based addressing just before the LUTs. To further reduce the size of the lookup table, the address is offset by the starting line of the trapezoid. In order to get good synthesis results, match typical block RAM registering in FPGAs by using a register after the lookup table. This register also adds some modest pipelining to the design.

For the 320x180 image:

```
raster = [320,180];
ltc = [155, 66];
lbc = [  1,140];
```

```
rtc = [155, 66];
rbc = [285,179];

% fit to x = f(y) for convenient LUT indexing
abl = polyfit([lbc(2),ltc(2)],[lbc(1),ltc(1)],1);  % left side
abr = polyfit([rbc(2),rtc(2)],[rbc(1),rtc(1)],1);  % right side
leftxstart = max(1,round((ltc(2):rbc(2))*abl(1)+abl(2)));
rightxend  = min(raster(1),round((ltc(2):rbc(2))*abr(1)+abr(2)));
startline  = min(ltc(2),rtc(2));
endline    = max(lbc(2),rbc(2));
% correct to zero-based addressing
leftxstart = leftxstart - 1;
rightxend = rightxend - 1;
startline = startline - 1;
endline = endline - 1;

open_system([modelname '/PotHoleHDL/TrapezoidalMask'],'force');
```
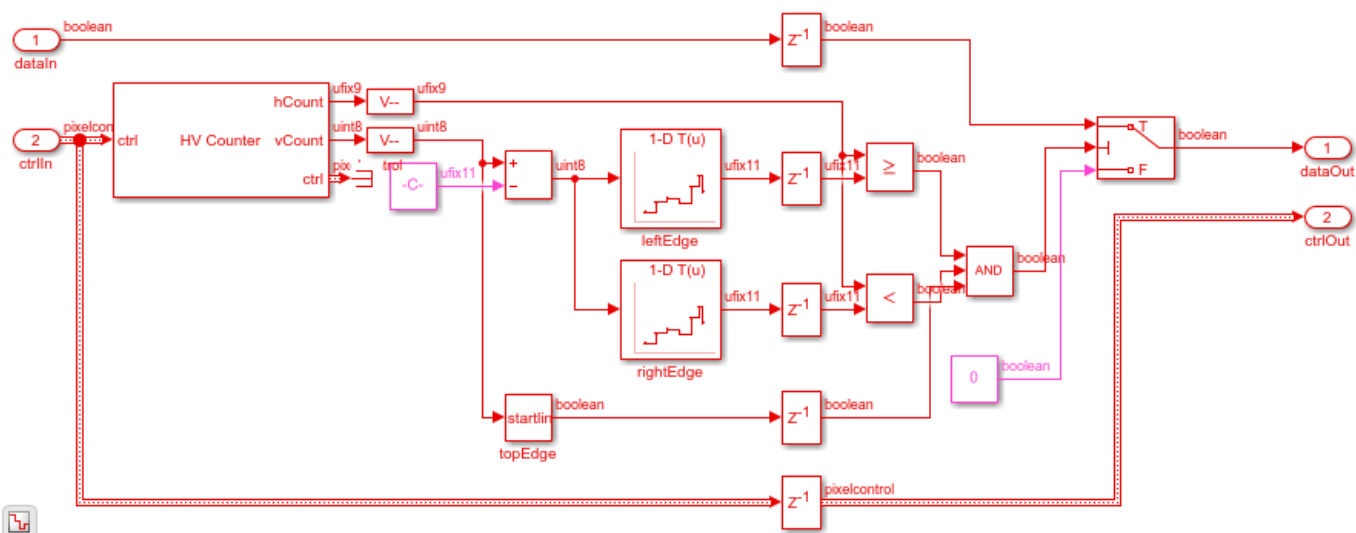

Morphological Closing

Next the design uses the Morphological Closing block to remove or close in small features. Closing works by first doing dilation and then erosion, and helps to remove small features that are not likely to be potholes. Specify a neighborhood on the block mask that determines how small or large a feature you want to remove. This model uses a 5x5 neighborhood, similar to a disk, so that small features are closed in.

Centroid

The centroid calculation finds the center of an active area. The design continuously computes the centroid of the marked area in each 31x31 pixel region. It only stores the center coordinates when the detected area is larger than an input parameter. This is a common difference between hardware and software systems: when designing hardware for FPGAs it is often easier to compute continuously but only store the answer when you need it, as opposed to calling functions as-needed in software.

For a centroid calculation, you need to compute three things from the region of the image: the weighted sum of the pixels in the horizontal direction, the weighted sum in the vertical direction, and the overall sum of all the pixels which corresponds to the area of the marked portion of the region.

The Line Buffer selects regions of 31x31 pixels, and returns them one column at a time. The algorithm uses the column to compute vertical weights, and total weights. For the horizontal weights, the design combines the columns to obtain a 31x31 kernel. You can choose the weights depending on what you want "center" to mean. This example uses -15:15 so that the center of the 31x31 region is (0,0) in the computed result.

The Vision HDL Toolbox blocks force the output data to zero when the output is not valid, as indicated in the pixelcontrol bus output. While not strictly required, this behavior makes testing and debugging much easier. To accomplish this behavior for the centroid results, the model uses Switch blocks with a Constant block set to 0.

Since you want the center of the detected region to be relative to the overall image coordinate system, add the horizontal and vertical pixel count to the calculated centroid.

open_system([modelname '/PotHoleHDL/Centroid31'],'force');
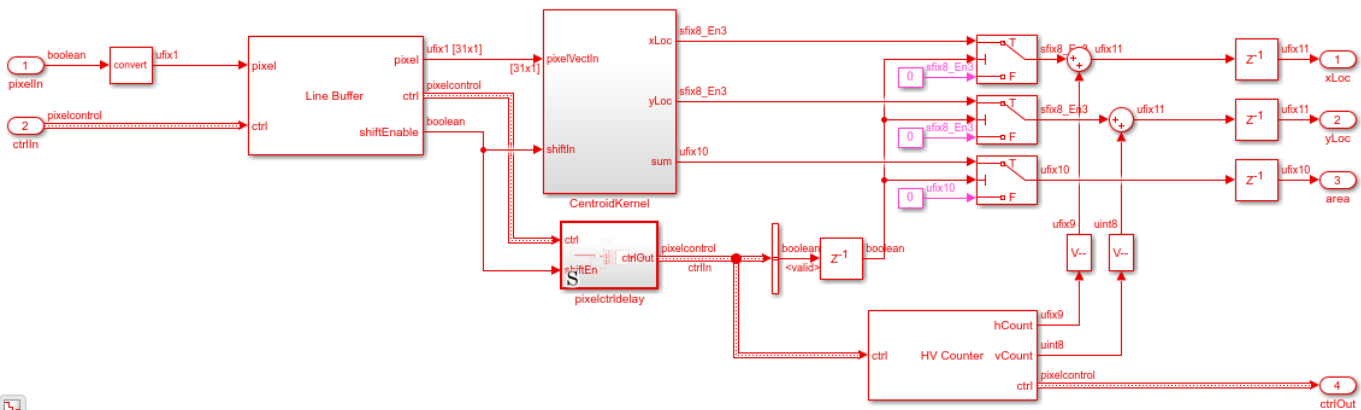
open_system([modelname '/PotHoleHDL/Centroid31/CentroidKernel'],'force');
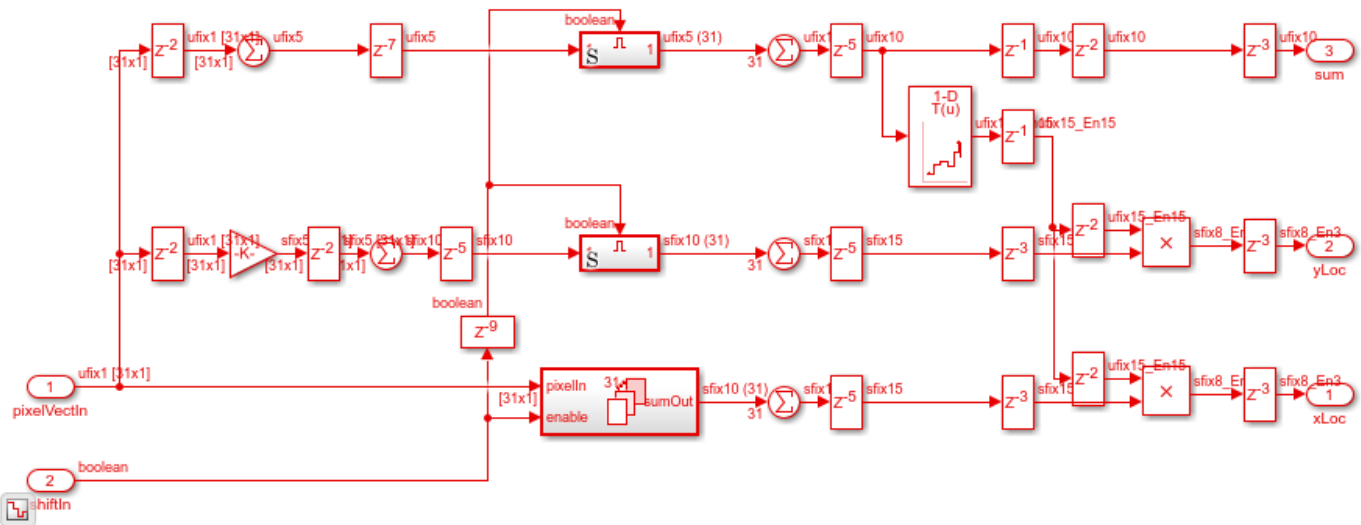
Detect and Hold

The detector operates on the total area sum from the centroid. The detector itself is very simple: compare the centroid area value to the threshold parameter, and find the largest area that is larger

than the threshold. The model logic compares a stored area value to the current area value and stores a new area when the input is larger than the currently stored value. By using > or >= you can choose the earliest value over the threshold or the latest value over the threshold. The model stores the latest value because later values are closer to the camera and vehicle. When the detector stores a new winning area value, it also updates the X and Y centroid values that correspond to that area. These coordinates are then passed to the alignment and overlay parts of the subsystem.

To pass the X, Y, and valid indication to the alignment algorithm, pack the values into one 23-bit word. The model unpacks them once they are aligned in time with the input frames for overlay.

```
open_system([modelname '/PotHoleHDL/DetectAndHold'],'force');
```
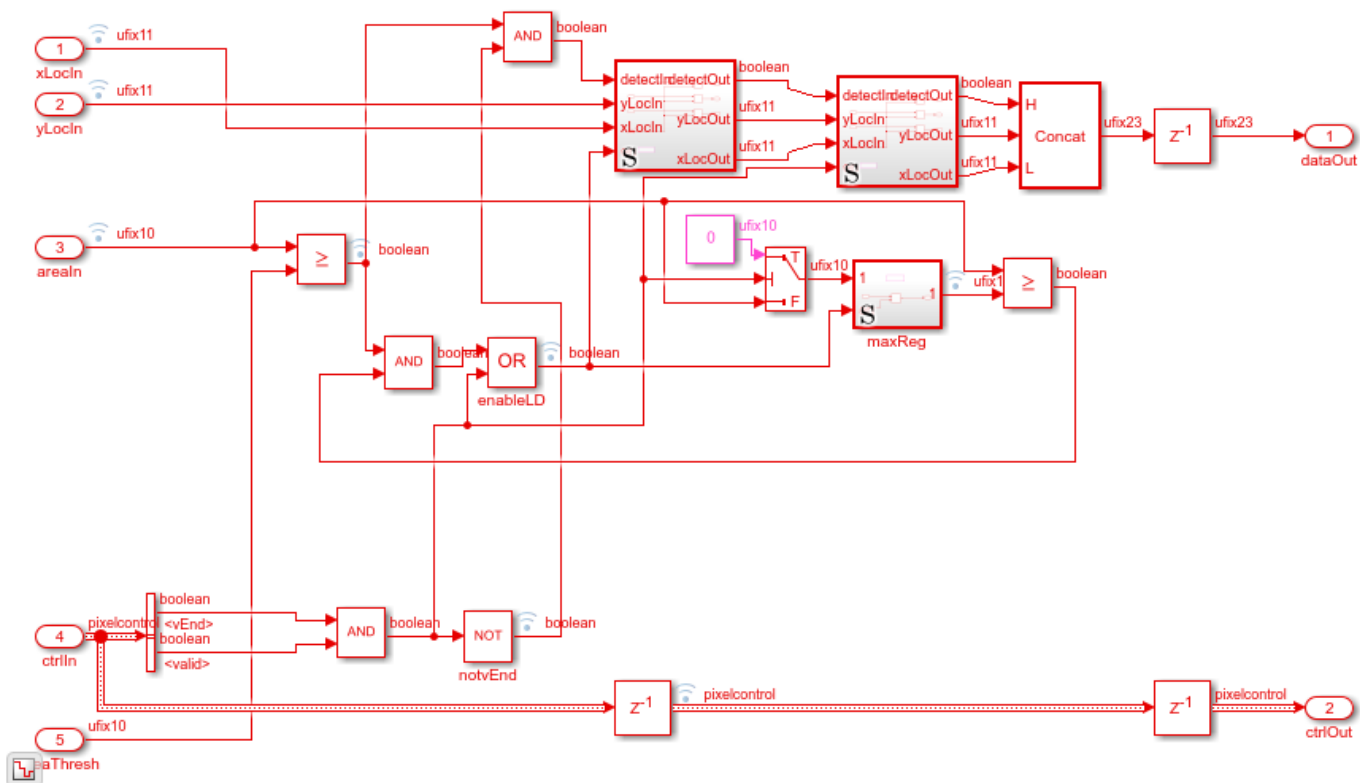

Pixel Stream Aligner

The Pixel Stream Aligner block takes the streaming information from the detector and sends it and the original RGB pixel stream to the overlay subsystems. The aligner compensates for the processing delay added by all the previous parts of the detection algorithm, without having to know anything about the latency of those blocks. If you later change a neighborhood size or add more processing, the aligner can compensate. If the total delay exceeds the **Maximum number of lines** parameter of the Pixel Stream Aligner block, adjust the parameter.

Fiducial Overlay

The fiducial marker is a square reticle represented as a 31-element array of 31-bit fixed-point numbers. This representation is convenient because a single read returns the whole word of overlay pixels for each line.

The diagram shows the overlay pattern by converting the fixed-point data to binary. This pattern can be anything you wish within the 31x31 size in this design.

```
load fiducialROM31x31.mat
crosshair = bin(fiducialROM);
crosshair(crosshair=='0') = ' ' %change '0' to space for better display


crosshair =

  31x31 char array

    '               1               '
    '               1               '
    '               1               '
    '               1               '
    '       1111111111111111111111   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1                       1   '
    '       1                       1   '
    '       1                       1   '
    '1111111111111           111111111111'
    '       1                       1   '
    '       1                       1   '
    '       1                       1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1           1           1   '
    '       1111111111111111111111   '
    '               1               '
    '               1               '
    '               1               '
    '               1               '
```

The fiducial overlay subsystem has a horizontal and vertical counter with a set of four comparators that uses the center of the detected area as the center of the region for the marker. The marker data is used as a binary switch that turns on alpha channel overlay. The alpha value is a fixed transparency parameter applied as a gain on the binary Detect signal when it is unpacked, in the ExpandData subsystem.

```
open_system([modelname '/PotHoleHDL/Fiducial31x31'],'force');
```

Character Overlay

The character font ROM for the on-screen display stores data in a manner similar to the fiducial ROM described above. Each 16-bit fixed-point number represents 16 consecutive horizontal pixels. The character maps are 16x16.

Since the character data would typically be written by a CPU in ASCII, the simplest way is to store the character data under 8-bit ASCII addresses in a dual-port RAM. The font ROM stores ASCII characters 33 ("!") to 122 ("z"). The design offsets the address by 33.

The font ROM was constructed from a public domain fixed width font with a few edits to improve readability. As in the fiducial marker, the character ROM data is used as a binary switch that turns on alpha channel overlay. The character alpha value is a fixed transparency parameter applied as a gain on the Detect signal when it is unpacked, in the ExpandData subsystem.

To visualize the character B in the font ROM, display it in binary.

```
load charROM16x16.mat
letterB = bin(charROM16x16(529:544)); % character array
letterB(letterB=='0')=' ' % remove '0' chars for better display


letterB =

  16x16 char array

    '                '
    '   111111111    '
    '   11111111111  '
    '   111     111  '
    '   111      111 '
    '   111      111 '
```

```
'   111     111    '
'   1111111111     '
'   111111111      '
'   111     111    '
'   111         111'
'   111         111'
'   111         111'
'   111        1111'
'   11111111111    '
'   111111111      '
```

```
open_system([modelname '/PotHoleHDL/Overlay32x32'],'force');
```
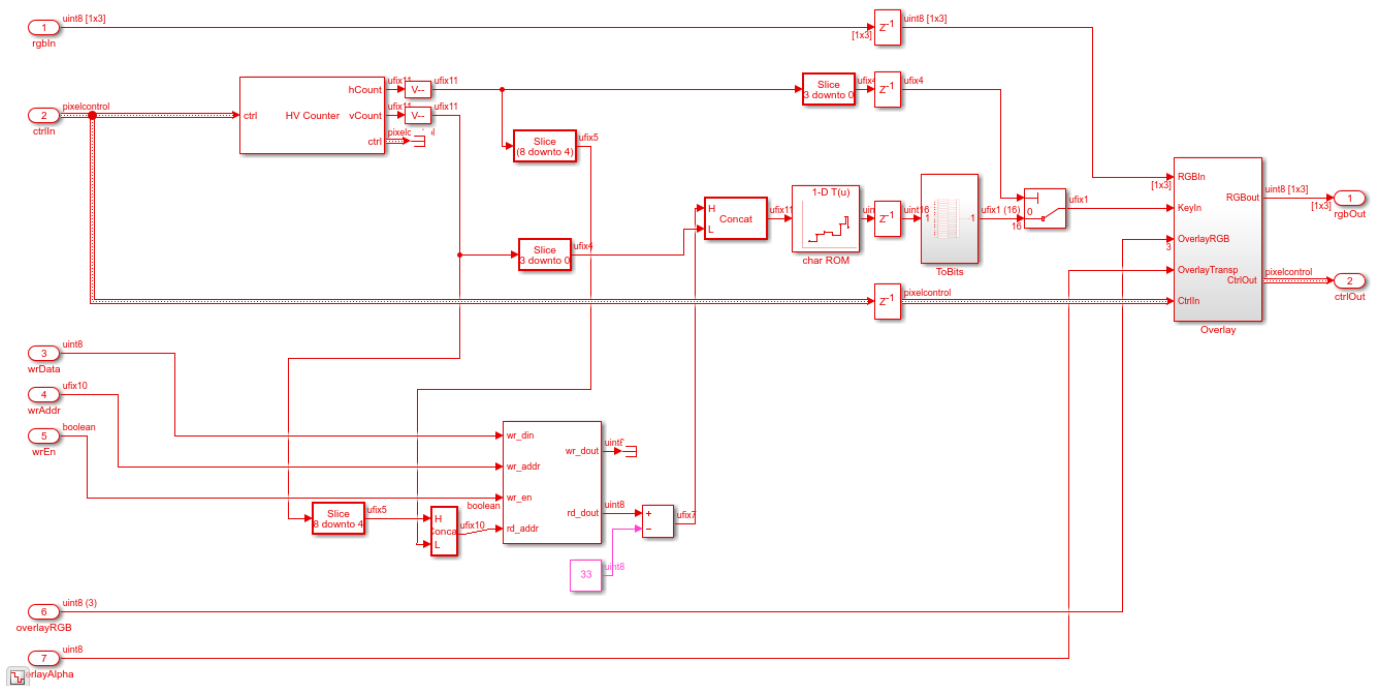

Viewing Detector Raw Image

When you work with a complicated algorithm, viewing intermediate steps in the processing can be very helpful for debugging and exploration. In this model, you can set the boolean Show Raw parameter to 1 (`true`) to display the result of morphological closing of the binary image, with the overlay of the detected results. To convert the binary image for use with the 8-bit RGB overlay, the model multiplies the binary value by 255 and uses that value on all three color channels.

HDL Code Generation

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('PotHoleHDLDetector/PotHoleHDL')
```

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

```
makehdltb('PotHoleHDLDetector/PotHoleHDL')
```

The part of this model that you can implement on an FPGA is the part between the Frame To Pixels and Pixels To Frame blocks. That is the subsystem called PotHoleHDL, which includes all the elements of the detector.

Simulation in an HDL Simulator

Now that you have HDL code, you can simulate it in your HDL Simulator. The automatically generated test bench allows you to prove that the Simulink simulation and the HDL simulation match.

Synthesis for an FPGA

You can also synthesize the generated HDL code in an FPGA synthesis tool, such as Xilinx Vivado. In a Virtex-7 FPGA (xc7v585tffg1157-1), the design achieves a clock rate of over 150 MHz.

The utilization report shows that the bilateral filter, pixel stream aligner, and centroid functions consume most of the resources in this design. The bilateral filter requires the most DSPs. The centroid implementation is quite efficient and uses only two DSPs. Centroid calculation also requires a reciprocal lookup table and so uses a large number of LUTs as memory.

Name	Slice LUTs (364200)	Slice Registers (728400)	F7 Muxes (182100)	Slice (91050)	LUT as Logic (364200)	LUT as Memory (111000)	LUT Flip Flop Pairs (364200)	Block RAM Tile (795)	DSPs (1260)
PotHoleHDL	18619	24540	92	7778	17012	1607	11170	305	92
u_Align (Align)	674	1065	90	519	646	28	149	96	0
u_Bilateral_Filter (Bilateral_...	6218	13879	0	3211	6070	148	5804	50	85
u_Centroid31 (Centroid31)	8836	6542	1	2794	7480	1356	3808	30.5	2
u_Closing (Closing)	1034	1057	0	400	998	36	619	8	0
u_Color_Space_Converte...	63	148	0	53	58	5	41	0	3
u_DetectAndHold (DetectA...	0	56	0	17	0	0	0	0	0
u_Edge_Detector (Edge_De...	569	926	0	282	544	25	376	2	2
u_Fiducial31x31 (Fiducial31...	266	163	0	147	262	4	82	0	0
u_FrameBoundary (FrameB...	77	86	0	85	77	0	2	0	0
u_Overlay32x32 (Overlay3...	495	178	1	193	490	5	103	1.5	0

Going Further

This example shows one possible implementation of an algorithm for detecting potholes. This design could be extended in the following ways :

- The gradient threshold could be computed from the average brightness using a gray-world model.
- The trapezoidal mask block could be made "steerable" by looking at the vehicle wheel position and adjusting the linear fit for the sloping sides of the mask.
- The detector could be made more robust by looking at the average brightness of the RGB or intensity image relative to the surrounding pavement since potholes are typically darker in intensity than the surrounding area.
- The visual frequency spectrogram of the pothole could also be used to look for specific types of surfaces in potholes.
- The detection area threshold value could be computed using average intensity in the trapezoidal roadway region.

- Multiple potholes could be detected in one frame by storing the top N responses rather than only the maximum detected response. The fiducial marker subsystem would need to be redesigned slightly to allow for overlapping markers.

Conclusion

This model shows how a pothole detection algorithm can be implemented in an FPGA. Many useful parts of this detector can be reused in other applications, such as the centroid block and the fiducial and character overlay blocks.

References

[1] Koch, Christian, and Ioannis Brilakis. "Pothole detection in asphalt pavement images." *Advanced Engineering Informatics* 25, no. 3 (2011): 507-15. doi:10.1016/j.aei.2011.01.002.

[2] Omanovic, Samir, Emir Buza, and Alvin Huseinovic. "Pothole Detection with Image Processing and Spectral Clustering." 2nd International Conference on Information Technology and Computer Networks (ICTN '13), Antalya, Turkey. October 2013.

Buffer Bursty Data Using Pixel Stream FIFO Block

This example shows how to interface with bursty pixel streams, such as those from DMA and Camera Link® sources, using the Pixel Stream FIFO block.

Overview

The DMACameraSourceHDL.slx system is shown below.

Buffer Bursty Data Using Pixel Stream FIFO Block
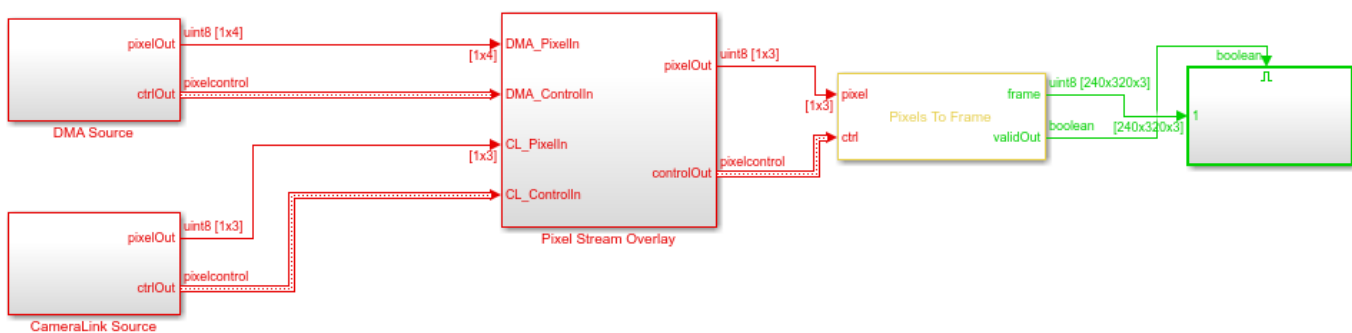
Copyright 2017 The MathWorks, Inc.

There are two pixel input streams - DMA source and Camera Link source. Input data for both sources is loaded from a `.mat` file, in the `InitFcn` callback. The DMA source models a non-contiguous stream of data arriving from off-chip memory. The pixels arrive in short bursts of random length, with random gaps between bursts. This can occur when there is contention on the DMA source and so it is not possible to stream pixels continuously from off-chip memory. The Camera Link source models the case when the camera is streaming an image of a lower resolution than the maximum permitted by the pixel clock and therefore will leave regular gaps between valid pixels. This spacing allows for streaming of multiple resolution images using a common clock, via strobing of validIn.

The Camera Link source models the incoming video stream from the sensor. The DMA source models a video stream from a frame buffer in which previous frame data has been processed in order to produce an alpha channel, allowing for blending of previous frame data with the current stream.

The **Pixel Stream Overlay** subsystem is shown in the diagram below. You can generate HDL code from this subsystem.

2-85
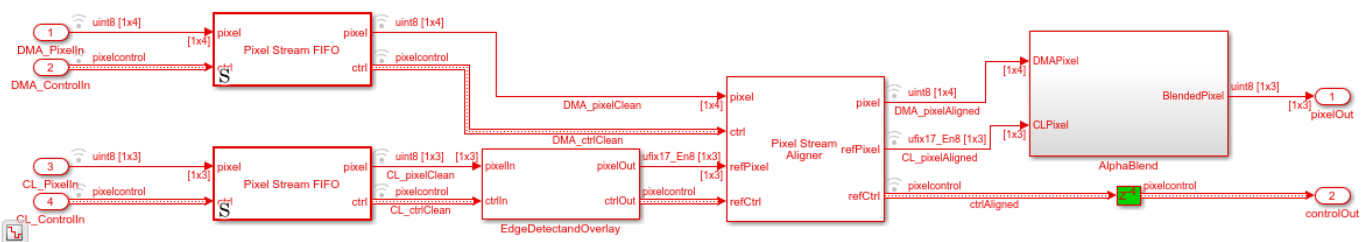
There are four main processing stages in the model - buffering of input data to remove burstiness, edge detection and overlay on Camera Link stream, alignment of pixel streams, and alpha blending of DMA stream onto Camera Link stream.

Pixel Stream Buffering

The Pixel Stream FIFO blocks buffer the input data as it is streamed into the model. The Pixel Stream FIFO is a masked subsystem. Looking into the Pixel Stream FIFO, we can see that it consists of a Memory Controller, Read and Write counters and two RAMs. One RAM stores the incoming pixel stream, and the other stores the incoming control signal stream. Once a full line has been buffered in RAM, the line is output continuously, removing any bursty behavior seen at input.
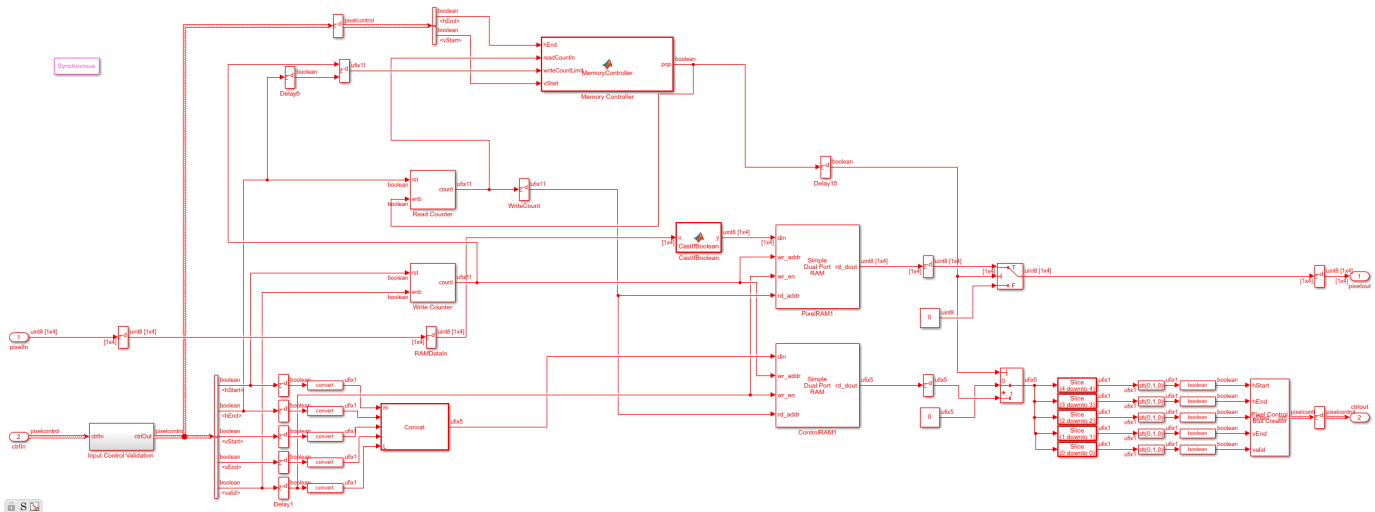
This waveform illustrates the difference in the pixel control signals after the Pixel Stream FIFO. The input `valid` signal, `DMA_ControlIn(5)`, shows short bursts of valid pixels, while the output `valid`, `DMA_ctrlClean(5)`, shows a continuous line of valid pixels. The total cycles in each line, shown by the time between `hStart` assertions, remains the same.
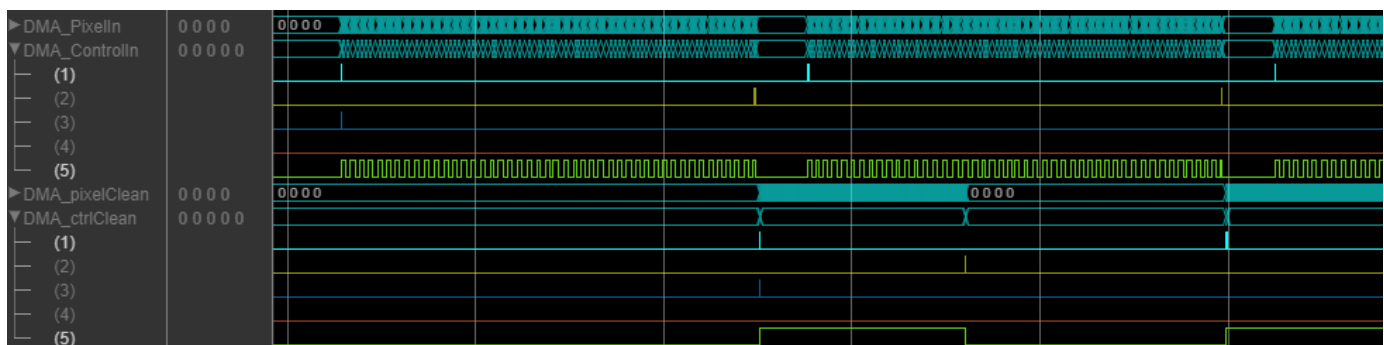
Edge Detection and Overlay on Camera Link Stream

To further differentiate the pixel streams, the Camera Link stream has an edge detection and overlay section. The pixel stream is first pre-processed by the Bilateral Filter block. This block smooths the image while preserving edges, and so it is a good choice for noise suppression prior to edge detection. The Edge Detector block detects edges using the Sobel method. The edges are then thinned using a [2x2] erosion operation. The thinned edge image is overlaid onto the original Camera Link image.
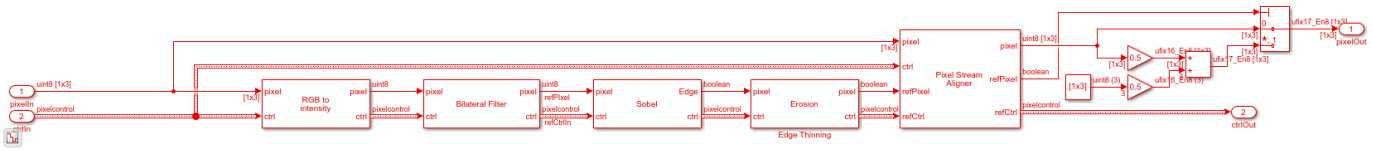
Pixel Stream Alignment

The Camera Link and DMA pixel streams must now be aligned to account for algorithmic delay in the data path. Aligning the pixel streams is straightforward using the Pixel Stream Aligner block.

Alpha Blending

The DMA input stream is a [1x4] vector whereas the Camera Link input is a [1x3] vector. The extra column in the DMA input is used to store the alpha channel information. The alpha channel represents the amount by which each of the pixels from the DMA source should be blended with the incoming Camera Link stream.
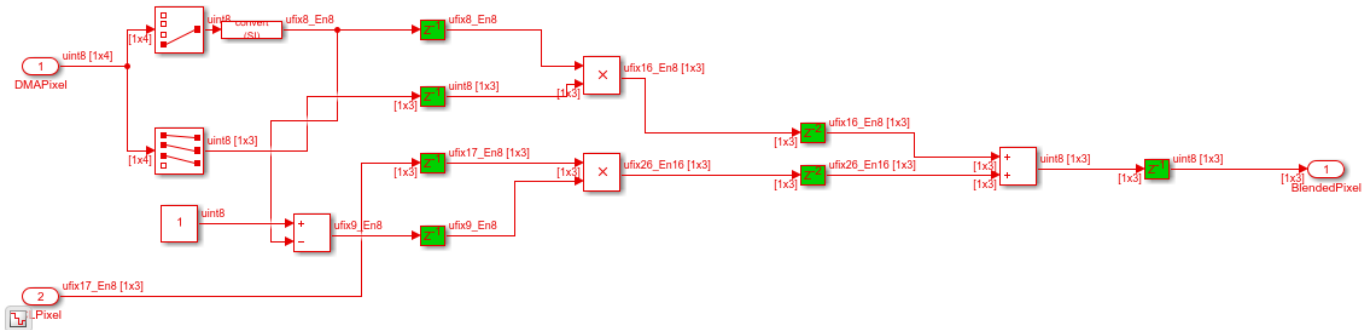
Results of the Simulation

The output video stream shows the DMA stream alpha blended onto the Camera Link input. The magenta colored overlay indicates the edges detected in the incoming Camera Link stream.
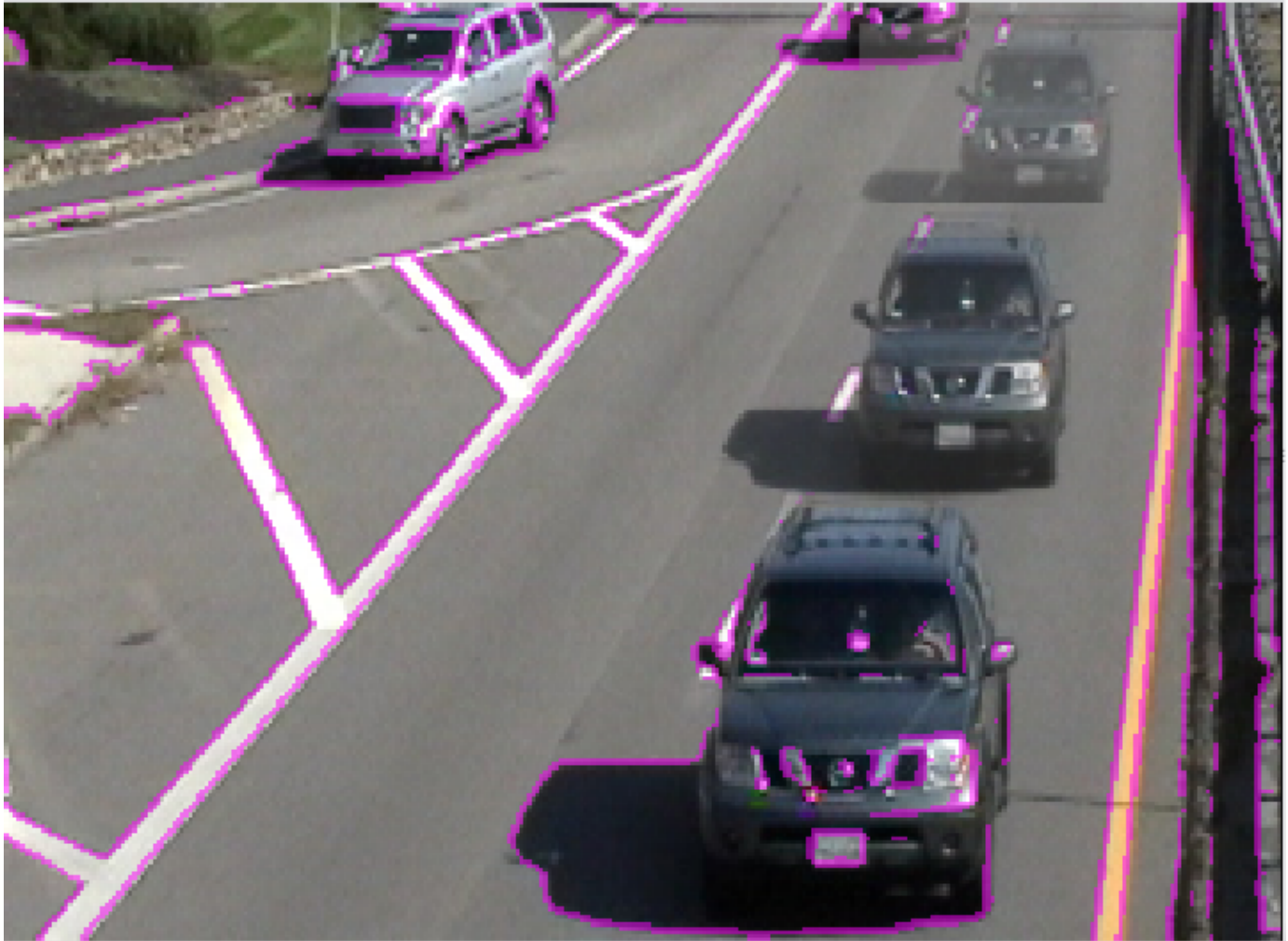
Generate HDL Code and Verify Its Behavior

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('DMACameraSourceHDL/Pixel Stream Overlay');
```

To generate an HDL test bench, use the following command:

```
makehdltb('DMACameraSourceHDL/Pixel Stream Overlay');
```

Using the Line Buffer to Create Efficient Separable Filters

This example shows how to design and implement a separable image filter, which uses fewer hardware resources than a traditional 2-D image filter.

Traditional image filters operate on a region of pixels and compute the resulting value of one central pixel using a two-dimensional filter kernel which is a matrix that represents the coefficients of the filter. Each coefficient is multiplied with its corresponding pixel and the result is summed to form the value. The region is then moved by one pixel and the next value is computed.

A separable filter is simple in concept: if the two-dimensional filter kernel can be factored into a horizontal component and a vertical component, then each direction can be computed separately using one-dimensional filters. This factorization can only be done for certain types of filter kernels. These kernels are called separable since the parts can be separated. Deciding which kernels are separable and which are not is easy using linear algebra in MATLAB. Mathematically, the two 1-D filters convolve to equal the original 2-D filter kernel, but a separable filter implementation often saves hardware resources.

Introduction

The SeparableFilterHDL.slx system is shown below. The SeptFiltHDL subsystem contains the separable filter, and also an Image Filter block implementation of the equivalent 2-D kernel as a reference.

```
modelname = 'SeparableFilterHDL';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
```
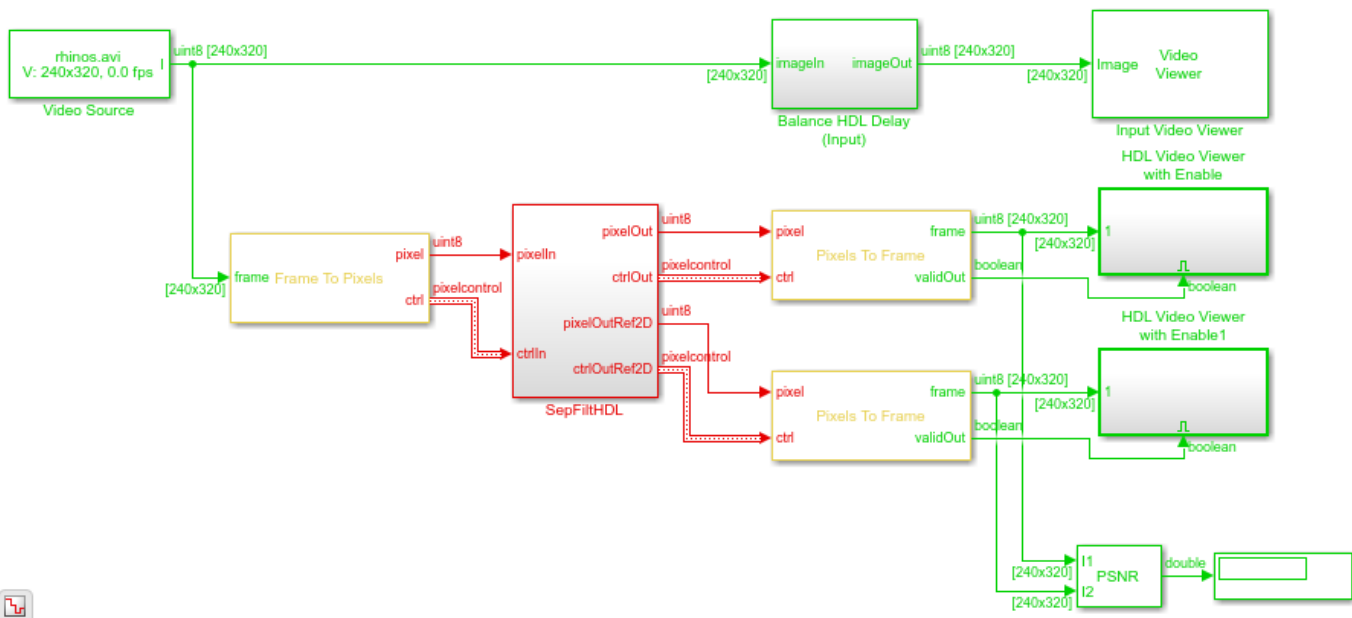
Determine Separable Filter Coefficients

Start by deciding what the purpose of your filter will be and compute the kernel. This example uses a Gaussian filter of size 5x5 with a standard deviation of 0.75. This filter has a blurring effect on images and is often used to remove noise and small details before other filtering operations, such as edge detection. Notice that the Gaussian filter kernel is circularly symmetric about the center.

```
Hg = fspecial('gaussian',[5,5],0.75)
```

```
Hg =

    0.0002    0.0033    0.0081    0.0033    0.0002
    0.0033    0.0479    0.1164    0.0479    0.0033
    0.0081    0.1164    0.2831    0.1164    0.0081
    0.0033    0.0479    0.1164    0.0479    0.0033
    0.0002    0.0033    0.0081    0.0033    0.0002
```

To check if the kernel is separable, compute its rank, which is an estimate of the number of linearly independent rows or columns in the kernel. If `rank` returns 1, then the rows and columns are related linearly and the kernel can be separated into its horizontal and vertical components.

```
rankHg = rank(Hg)
```

```
rankHg =

     1
```

To separate the kernel, use the `svd` function to perform singular value decomposition. The `svd` function returns three matrices, `[U,S,V]`, such that `U*S*V'` returns the original kernel, `Hg`. Since the kernel is rank 1, `S` contains only one non-zero element. The components of the separated filter are the first column of each of `U` and `V`, and the singular value split between the two vectors. To split the singular value, multiply both vectors with the square root of `S`. You must reshape `V` so that `Hh` is a horizontal, or row, vector.

For more information on filter separability, refer to the links at the bottom of this example.

```
[U,S,V]=svd(Hg)
Hv=abs(U(:,1)*sqrt(S(1,1)))
Hh=abs(V(:,1)'*sqrt(S(1,1)))
```

```
U =

   -0.0247   -0.9912   -0.1110    0.0673   -0.0000
   -0.3552    0.0737   -0.6054   -0.0431    0.7071
   -0.8640   -0.0301    0.4988    0.0619   -0.0000
   -0.3552    0.0737   -0.6054   -0.0431   -0.7071
   -0.0247   -0.0754    0.0761   -0.9939    0.0000
```

```
S =

    0.3793         0         0         0         0
         0    0.0000         0         0         0
```

```
       0          0    0.0000          0          0
       0          0          0    0.0000          0
       0          0          0          0    0.0000


V =

  -0.0247     0.1742     0.8521    -0.4929          0
  -0.3552     0.5980    -0.0716     0.1053     0.7071
  -0.8640    -0.4937     0.0199    -0.0968     0.0000
  -0.3552     0.5980    -0.0716     0.1053    -0.7071
  -0.0247    -0.1031     0.5131     0.8518    -0.0000


Hv =

   0.0152
   0.2188
   0.5321
   0.2188
   0.0152


Hh =

   0.0152     0.2188     0.5321     0.2188     0.0152
```

You can check your work by reconstructing the original kernel from these factored parts and see if they are the same to within floating-point precision. Compute the check kernel, Hc, and compare it to the original Hg using a tolerance.

```
Hc = Hv * Hh;
equalTest = all(Hc(:)-Hg(:) < 5*max(eps(Hg(:))))


equalTest =

  logical

   1
```

This result proves that Hv and Hh can be used to recreate the original filter kernel.

Fixed-Point Settings

For HDL code generation, you must set the filter coefficients to fixed-point data types. When picking fixed-point types, you must consider what happens to separability when you quantize the kernel.

First, quantize the entire kernel to a nominal data type. This example uses a 10-bit fixed-point number. Let the fixed-point tools select the best fraction length to represent the kernel values. Do the same conversion for the horizontal and vertical component vectors.

```
Hgfi = fi(Hg,0,10);
Hvfi = fi(Hv,0,10);
Hhfi = fi(Hh,0,10);
```

In this case, the best-precision 10-bit answer for Hg has 11 fractional bits, while Hv and Hh use only 10 fractional bits. This result makes sense since Hv and Hh are multiplied and added together to make the original kernel.

Now you must check if the quantized kernel is still rank 1. Since the `rank` and `svd` functions do not accept fixed-point types, you must convert back to doubles. This operation does not quantize the results, as long as the fixed-point type is smaller than 53 bits, which is the effective mantissa size of doubles.

```
rankDouble = rank(double(Hgfi))


rankDouble =

     3
```

This result shows that quantization can have a dramatic effect on the separability: since the rank is no longer 1, the quantized filter does not seem to be separable. For this particular filter kernel, you could experiment with the quantized word-length and discover that 51 bits of precision are needed in order for the rank function to return 1 after quantization. Actually, this result is overly conservative because of quantization of near-zero values within the `rank` function.

Instead of expanding the fixed-point type to 51 bits, add a tolerance argument to the `rank` function to limit the quantization effects.

```
rankDouble2048 = rank(double(Hgfi),1/2048)


rankDouble2048 =

     1
```

This result shows that the quantized kernel is rank 1 within an 11-bit fractional tolerance. So, the 11-bit separated coefficients are acceptable after all.

Another quantization concern is whether the fixed-point filter maintains flat field brightness. The filter coefficients must sum to 1.0 to maintain brightness levels. For a normalized Gaussian filter such as this example, the coefficient sum is always 1.0, but this sum can be moved away from 1.0 by fixed-point quantization. It can be critical in video and image processing to maintain exactly 1.0, depending on the application. If you imagine a chain of filters, each one of which raises the average level by around 1%, then the cumulative error can be large.

```
sumHg = sum( Hg(:) )
sumHgfi = sum( Hgfi(:) )


sumHg =

    1.0000


sumHgfi =

     1
```

```
        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Unsigned
          WordLength: 15
       FractionLength: 11
```

In this case, the sums of the double-precision `Hg` and the fixed-point `Hgfi` are indeed 1.0. If maintaining brightness levels to absolute limits is important in your application, then you might have to manually adjust the filter coefficient values to maintain a sum of 1.

Finally, check that the combination of the quantized component filters still compares to the quantized kernel. By default, the `fi` function uses full precision on the arithmetic expression. Use convergent rounding since there are some coefficient values very near the rounding limit.

```
Hcfi = fi(Hvfi * Hhfi,0,10,'fimath',fimath('RoundingMethod','Convergent'));
equalTest = all( Hcfi(:)==Hgfi(:) )
```

```
equalTest =

  logical

   1
```

This result confirms that the fixed-point, separated coefficients achieve the same filter as the 2-D Gaussian kernel.

Implementing the Separable Filter

To see the separable filter implementation, open the Separable Filter subsystem that is inside the SepFiltHDL subsystem.

```
open_system([modelname '/SepFiltHDL/Separable Filter'],'force');
```
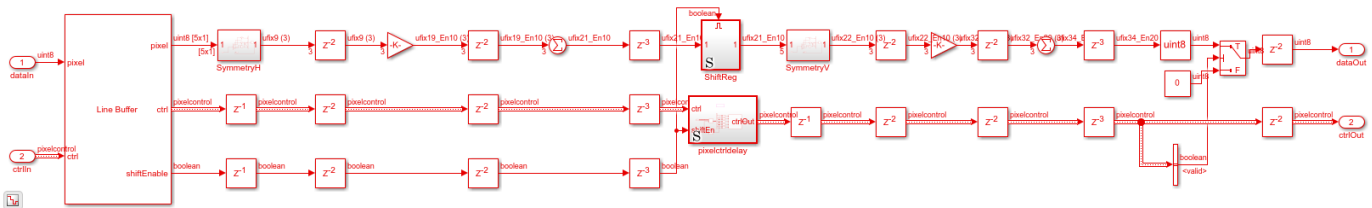

This subsystem selects vertical and horizontal vectors of pixels for filtering, and performs the filter operation.

The Line Buffer outputs a column of pixels for every time step of the filter. The Line Buffer also pads the edges of the image. This model uses **Padding method**: `Constant`, with a value of 0. The shiftEnable output signal is normally used to control a horizontal shift register to compile a 2-D pixel kernel. However, for a separable filter, you want to work in each direction separately. This model uses the output pixel column for the vertical filter, and uses the shiftEnable signal later to construct the horizontal pixel vector.

The separated horizontal and vertical filters are symmetric, so the model uses a pre-adder to reduce the number of multipliers even further. After the adder, a Gain block multiplies the column of pixels by the `Hv` vector. The Gain parameter is set to `Hv` and the parameter data type is `fixdt(0,10)`. The resulting output type in full-precision is `ufix18_En10`. Then a Sum block completes the vertical

filter. The Sum block is configured in full-precision mode. The output is a scalar of `ufix21_En10` type.

There are many pipelining options you could choose, but since this design is simple, manual pipelining is quick and easy. Adding delays of 2 cycles before and after the Gain multipliers ensures good speed when synthesized for an FPGA. A delay of 3 cycles after the Sum allows for it to be sufficiently pipelined as well. The model balances these delays on the pixelcontrol bus and the shiftEnable signal before going to the horizontal dimension filter.

The best way to create a kernel-width shift register is to use a Tapped Delay block, which shifts in a scalar and outputs the register values as a vector. For good HDL synthesis results, use the Tapped Delay block inside an enabled subsystem, with the Synchronous marker block.

The output of the Tapped Delay subsystem is a vector of 5 horizontal pixels ready for the horizontal component of the separable filter. The model implements a similar symmetric pre-add and Gain block, this time with `Hh` as the parameter. Then, a Sum block and similar pipelining complete the horizontal filter. The final filtered pixel value is in the full-precision data type `ufix34_En20`.

Many times in image processing you would like to do full-precision or at least high-precision arithmetic operations, but then return to the original pixel input data type for the output. This subsystem returns to the original pixel type by using a Data Type Conversion block set to `uint8`, with `Nearest` rounding and saturation.

The Vision HDL Toolbox blocks force the output data to zero when the output is not valid, as indicated in the pixelcontrol bus output. While not strictly required, this behavior makes testing and debugging much easier. To accomplish this behavior, the model uses a Switch block with a Constant block set to 0.

Resource Comparison

The separable 5x5 filter implementation uses 3 multipliers in the vertical direction and 3 multipliers in the horizontal direction, for a total of 6 multipliers. A traditional image filter usually requires 25 multipliers for a 5x5 kernel. However, the Image Filter block takes advantage of any symmetry in the kernel. In this example the kernel has 8-way and 4-way symmetry, so the Image Filter only uses 5 multipliers. In general there are savings in multipliers when implementing a separable filter, but in this case the 2-D implementation is similar.

The separable filter uses 4 two-input adders in each direction, 2 for the pre-add plus 2 in the Sum, for a total of 8. The Image Filter requires 14 adders total, with 10 pre-add adders and 4 final adders. So there is a substantial saving in adders.

The Image Filter requires 25 registers for the shift register, while the separable filter uses only 5 registers for the shift register. Each adder also requires a pipeline register so that is 8 for the separable case and 14 for the traditional case. The number of multiplier pipeline registers scales depending on the number of multipliers.

The separable filter uses fewer adders and registers than the 2-D filter. The number of multipliers is similar between the two filters only because the 2-D implementation optimizes the symmetric coefficients.

Results of the Simulation

The resulting images from the simulation of the separable filter and the reference Image Filter are very similar. Using the fixed-point settings in this example, the difference between the separable filter

and the reference filter never exceeds one bit. This difference is a 0.1% difference or greater than 54 dB PSNR between the filtered images overall.

HDL Code Generation

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('SeparableFilterHDL/SepFiltHDL')
```

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

```
makehdltb('SeparableFilterHDL/SepFiltHDL')
```

The part of this model that you can implement on an FPGA is the part between the Frame To Pixels and Pixels To Frame blocks. The SepFiltHDL subsystem includes both the separable algorithm and the traditional 2-D implementation for comparison purposes.

Simulation in an HDL Simulator

Now that you have HDL code, you can simulate it in your HDL simulator. The automatically generated test bench allows you to prove that the Simulink simulation and the HDL simulation match.

Synthesis for an FPGA

You can also synthesize the generated HDL code in an FPGA synthesis tool, such as Xilinx Vivado. In a Virtex-7 FPGA (xc7v585tffg1157-1), the filter design achieves a clock rate of over 250 MHz.

The utilization report shows that the separable filter uses fewer resources than the traditional image filter. The difference in resource use is small due to the symmetry optimizations applied by the Image Filter block.

Utilization - utilization_1									? _ □ ⤢ ×

Name	Slice LUTs (364200)	Slice Registers (728400)	Slice (91050)	LUT as Logic (364200)	LUT as Memory (111000)	LUT Flip Flop Pairs (364200)	Block RAM Tile (795)	DSPs (1260)	Bonded IOB (600)
⊟ SepFiltHDL	1660	2729	661	1508	152	1174	8	7	
⊞ u_Image_Filter (Image_Filter)	854	1500	370	793	61	623	4	4	
⊞ u_Separable_Filter (Separa...	806	1198	333	715	91	536	4	3	

utilization_1

Going Further

The filter in this example is configured for Gaussian filtering but other types of filters are also separable, including some that are very useful. The mean filter, which has a kernel with coefficients that are all `1/N`, is always separable.

```
Hm = ones(3)./9
rank(Hm)
```

Hm =

```
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111


ans =

    1
```

Or the Sobel edge-detection kernel:

```
Hs = [1 0 -1; 2 0 -2; 1 0 -1]
rank(Hs)


Hs =

    1     0    -1
    2     0    -2
    1     0    -1


ans =

    1
```

Or gradient kernels like this:

```
Hgrad = [1 2 3; 2 4 6; 3 6 9]
rank(Hgrad)


Hgrad =

    1     2     3
    2     4     6
    3     6     9


ans =

    1
```

Separability can also be applied to filters that do not use multiply-add, such as morphological filters where the operator is min or max.

Conclusion

You have used linear algebra to determine if a filter kernel is separable or not, and if it is, you learned how to separate the components using the svd function.

You explored the effects of fixed-point quantization and learned that it is important to work with precise values when calculating rank and singular values. You also learned about the importance of maintaining DC gain. Finally you learned why separable filters can be implemented more efficiently and how to calculate the savings.

References

[1] Eddins, S. "Separable convolution". Steve on Image Processing (October 4, 2006).

[2] Eddins, S. "Separable convolution: Part 2". Steve on Image Processing (November 28, 2006).

Image Pyramid for FPGA

This example shows how to generate multi-level image pyramid pixel streams from an input stream. This model derives multiple pixel streams by downsampling the original image in both the horizontal and vertical directions, using Gaussian filtering. This type of filter avoids aliasing artifacts.

Image pyramid is used in many image processing applications such as image compression, object detection and recognition using techniques such as convolutional neural network (CNN) or aggregate channel features (ACF). Image pyramid is also similar to scale-space representation.

The example model takes a 240p video input and produces three output streams: 160x120, 80x60, and 40x30.

```
modelname = 'ImagePyramidHDL';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'SimulationCommand','Update');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
```
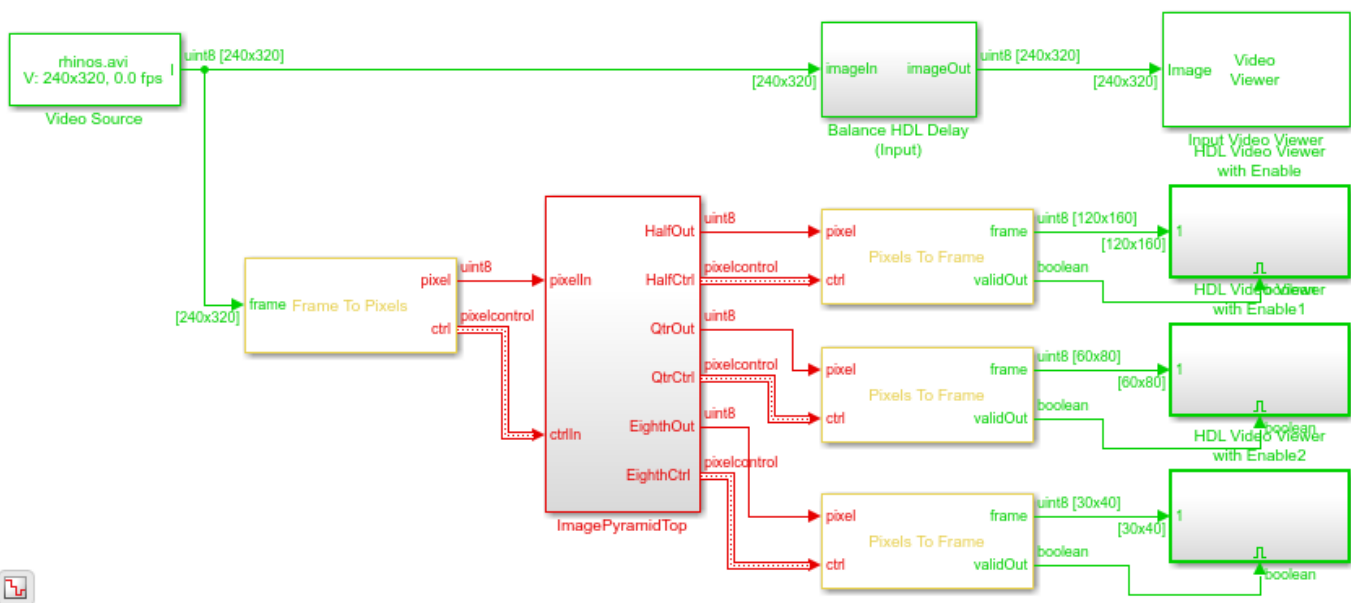

Each level of the pyramid contains a Line Buffer block and a downsampling filter.

```
open_system([modelname '/ImagePyramidTop/ResamplingPyramidFilter'],'force');
```
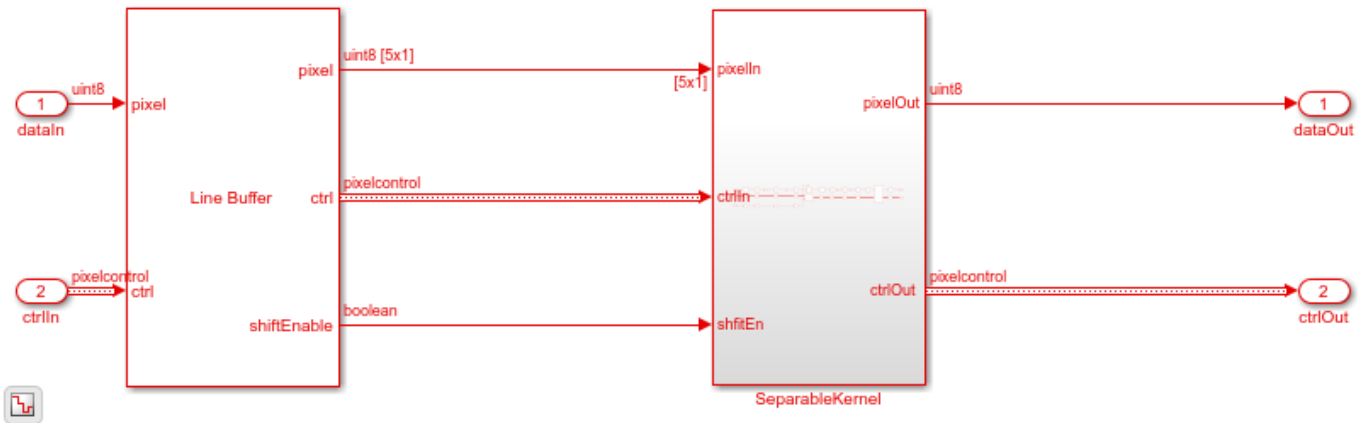
Filter Coefficients

The approximate Gaussian filter coefficients in [1] have been used in a number of image pyramid implementations. These coefficients are given by:

```
format long
Hh = [1 4 6 4 1]./16;
Hv = Hh';
Hg = Hv*Hh


Hg =

  Columns 1 through 3

   0.003906250000000   0.015625000000000   0.023437500000000
   0.015625000000000   0.062500000000000   0.093750000000000
   0.023437500000000   0.093750000000000   0.140625000000000
   0.015625000000000   0.062500000000000   0.093750000000000
   0.003906250000000   0.015625000000000   0.023437500000000

  Columns 4 through 5

   0.015625000000000   0.003906250000000
   0.062500000000000   0.015625000000000
   0.093750000000000   0.023437500000000
   0.062500000000000   0.015625000000000
   0.015625000000000   0.003906250000000
```

The results are similar to but not exactly the same as the Gaussian kernel with a 1.0817797 standard-deviation. So, Hg is an approximate Gaussian kernel.

```
Hf = fspecial('gaussian',5,1.0817797)


Hf =

  Columns 1 through 3

   0.004609023214619   0.016606534868404   0.025458671096979
   0.016606534868404   0.059834153028525   0.091728830511040
```

```
    0.025458671096979    0.091728830511040    0.140625009648116
    0.016606534868404    0.059834153028525    0.091728830511040
    0.004609023214619    0.016606534868404    0.025458671096979

Columns 4 through 5

    0.016606534868404    0.004609023214619
    0.059834153028525    0.016606534868404
    0.091728830511040    0.025458671096979
    0.059834153028525    0.016606534868404
    0.016606534868404    0.004609023214619
```

The filter, Hg, is obviously separable, since it was constructed from horizontal and vertical vectors. Therefore, a separable filter implementation is a good choice. Many of the coefficient values are powers of two or a combination of only two powers of two. These values mean that the filter implementation can replace multiplication with shift and add techniques such as canonical signed digit (CSD). Each vector in the separable representation is also symmetric, so the filter implementation uses a symmetry pre-adder to further reduce the number of operations.

Downsampling

After low-pass filtering with the approximate Gaussian filter above, the model then downsamples the pixel stream by two in both the horizontal and vertical directions. This is primarily accomplished by alternating the valid signal every other pixel. The model also recreates the other `pixelcontrol` bus signals.

The model includes horizontal and vertical counters that compare the number of output pixels and lines with the mask parameters for active pixels and lines. The model uses these counts to recreate the end of line (hEnd) and end of frame (vEnd) signals.

After downsampling once, the `pixelcontrol` bus valid signal alternates high and then low every other pixel. After the second downsample, it alternates with a pattern of one valid pixel followed by three non-valid pixels. In some applications, you may want to collect all the valid pixels into a continuously valid period of time. The Pixel Stream FIFO block, used between downsample stages, produces continuous valid pixels for each line.

Each ResamplingPyramidFilter subsystem accepts parameters for the output frame size. These numbers must be integers, and a factor of two smaller than the input image. If the input number of pixels per line is odd rather than even, then round down to the next integer. For example, if the input size is 25 pixels per line, the requested output size must be 12 pixels per line.

Going Further

The Gaussian filter kernel used in traditional image pyramid is not the only low-pass filter that could be used. Using an edge-preserving low-pass filter, such as a bilateral filter, with different kernel sizes, would preserve more detail in the pyramid.

It is sometimes helpful to compute the difference between two levels of an image pyramid. This algorithm is called a Laplacian pyramid. The smaller level is upsampled to same size as the larger level, and filtered. The filter is usually a scaled version of the same approximate Gaussian filter used in this model. The difference between layers represents the information lost in the downsampling process. A Laplacian pyramid can be used for applications including coring for noise removal, compositing images taken at different times or with different focal lengths, and many others.

A potential limitation of this model is that there is fairly high latency between the output streams. This latency occurs because the second and third levels depend on the output from the previous level. This could be avoided by creating parallel filters operating on more lines. This example implements a 5-by-5 filter that stores 5 lines at each level. A lower latency parallel implementation requires 13 lines of storage for a two-level filter or 103 lines for a three-level filter. This is not generally a cost-effective trade-off.

On FPGAs, line buffer memories are typically implemented using block RAMs. Smaller memories can be implemented in the FPGA fabric, and are known as distributed RAMs. Your synthesis tool chooses block or distributed RAM depending on the resources of your device. As the line size becomes smaller due to downsampling, distributed RAMs can be more efficient. In this example, the Line Buffer blocks in each level reserve space for up to 2k pixels per line. This size is the default size for the Line Buffer, and accommodates up to 1080p format video. To target distributed RAMs, specify a small power of two for the **Line buffer size** parameter. In this example, you could set the line buffer sizes of the three levels to 256, 128, and 64.

References

[1] Burt, P., and E. Adelson. "The Laplacian Pyramid as a Compact Image Code."*IEEE Transactions on Communications* 31, no. 4 (April 1983): 532-40.

FPGA Implementation of Stereo Disparity using Semi-Global Block Matching

This example shows how to compute disparity between left and right stereo camera images using the Semi-Global Block Matching algorithm. This algorithm is suitable for implementation on an FPGA.

Distance estimation is an important measurement for applications in Automated Driving and Robotics. A cost-effective way of performing distance estimation is by using stereo camera vision. With a stereo camera, depth can be inferred from point correspondences using triangulation. Depth at any given point can be computed if the disparity at that point is known. Disparity measures the displacement of a point between two images. The higher the disparity, the closer the object.

This example computes disparity using the Semi-Global Block Matching (SGBM) method, similar to the `disparity` function. The SGBM method is an intensity-based approach and generates a dense and smooth disparity map for good 3D reconstruction. However, it is highly compute-intensive and requires hardware acceleration using FPGAs or GPUs to obtain real-time performance.

The example model presented here is FPGA-hardware compatible, and can therefore provide real-time performance.

Introduction

Disparity estimation algorithms fall into two broad categories: local methods and global methods. Local methods evaluate one pixel at a time, considering only neighboring pixels. Global methods consider information that is available in the whole image. Local methods are poor at detecting sudden depth variation and occlusions, and hence global methods are preferred. Semi-global matching uses information from neighboring pixels in multiple directions to calculate the disparity of a pixel. Analysis in multiple directions results in a lot of computation. Instead of using the whole image, the disparity of a pixel can be calculated by considering a smaller block of pixels for ease of computation. Thus, the Semi-Global Block Matching (SGBM) algorithm uses block-based cost matching that is smoothed by path-wise information from multiple directions.

Using the block-based approach, this algorithm estimates approximate disparity of a pixel in the left image from the same pixel in the right image. More information about Stereo Vision is available here. Before going into the algorithm and implementation details, two important parameters need to be understood: Disparity Levels and Number of Directions.

Disparity Levels: Disparity levels is a parameter used to define the search space for matching. As shown in figure below, the algorithm searches for each pixel in the Left Image from among D pixels in the Right Image. The D values generated are D disparity levels for a pixel in Left Image. The first D columns of Left Image are unused because the corresponding pixels in Right Image are not available for comparison. In the figure, w represents the width of the image and h is the height of the image. For a given image resolution, increasing the disparity level reduces the minimum distance to detect depth. Increasing the disparity level also increases the computation load of the algorithm. At a given disparity level, increasing the image resolution increases the minimum distance to detect depth. Increasing the image resolution also increases the accuracy of depth estimation. The number of disparity levels are proportional to the input image resolution for detection of objects at the same depth. This example supports disparity levels from 8 to 128 (both values inclusive). **The explanation of the algorithm refers to 64 disparity levels.** The models provided in this example can accept input images of any resolution.
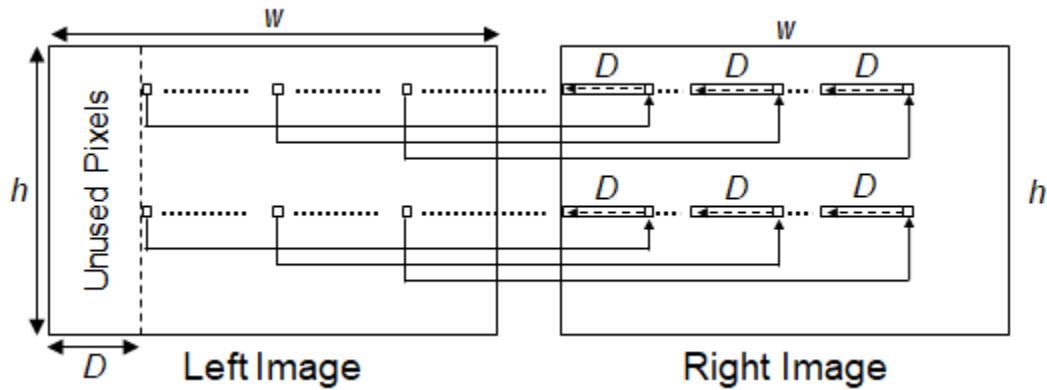
Number of Directions: In the SGBM algorithm, to optimize the cost function, the input image is considered from multiple directions. In general, accuracy of disparity result improves with increase in number of directions. This example analyzes five directions: left-to-right (A1), top-left-to-bottom-right (A2), top-to-bottom (A3), top-right-to-bottom-left (A4), and right-to-left (A5).
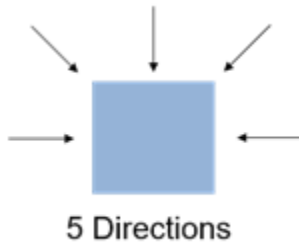
SGBM Algorithm

The SGBM algorithm takes a pair of rectified left and right images as input. The pixel data from the raw images may not have identical vertical coordinates because of slight variations in camera positions. Images need to be rectified before performing stereo matching to make all epi-polar lines parallel to the horizontal axis and match vertical coordinates of each corresponding pixel. For more details on rectification, please see `rectifyStereoImages` function. The figure shows a block diagram of the SGBM algorithm, using five directions.
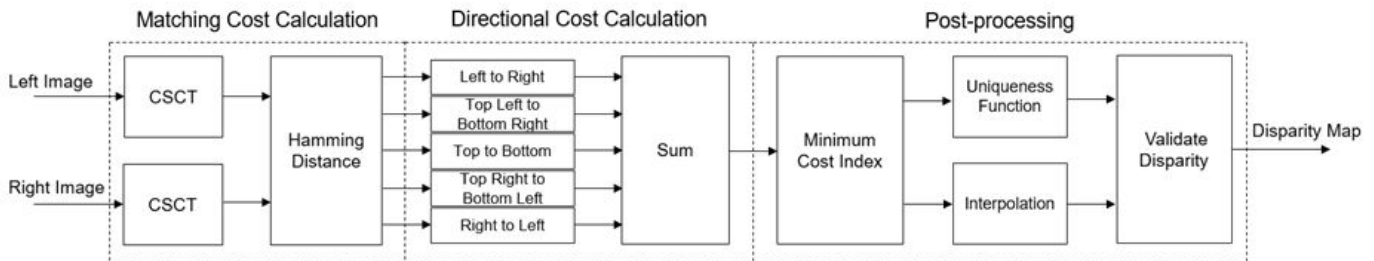
The SGBM algorithm implementation has three major modules: Matching Cost Calculation, Directional Cost Calculation and Post-processing.

Many methods have been explored in the literature for computing matching cost. This example implementation uses the census transform as explained in [2]. This module can be divided into two steps: Center-Symmetric Census Transform (CSCT) of left and right images and Hamming Distance computation. First, the model computes the CSCT on each of the left and right images using a sliding window. For a given pixel, a 9-by-7 pixel window is considered around it. CSCT for the center pixel in that window is estimated by comparing the value of each pixel with its corresponding center-symmetric counterpart in the window. If the pixel value is larger than its corresponding center-symmetric pixel, the result is 1, otherwise the result is 0. The figure shows an example 9-by-7 window. The center pixel number is 31. The 0th pixel is compared to the 62nd pixel (blue), the 1st pixel is compared to the 61st pixel (red), and so on, to generate 31 results. Each result a single bit output and the result of the whole window is arranged as a 31-bit number. This 31-bit number is the CSCT output for each pixel in both images.

0	1	2	3					
		31						
				59	60	61	62	

In the Hamming Distance module, the CSCT outputs of the left and right images are pixel-wise XOR'd and set bits are counted to generate the matching cost for each disparity level. To generate D disparity levels, D pixel-wise Hamming distance computation blocks are used. The matching cost for D disparity levels at a given pixel position, p, in the left image is computed by computing the Hamming distance with *(p to D+p)* pixel positions in the right image. The matching cost, *C(p,d)*, is computed at each pixel position, p, for each disparity level, d. The matching cost is not computed for pixel positions corresponding to the first D columns of the left image.

The second module of SGBM algorithm is directional cost estimation. In general, due to noise, the matching cost result is ambiguous and some wrong matches could have lower cost than correct ones. Therefore additional constraints are required to increase smoothness by penalizing changes of neighboring disparities. This constraint is realized by aggregating 1-D minimum cost paths from multiple directions. It is represented by aggregated cost from r directions at each pixel position, *S(p,d)*, as given by

$$S(p, d) = \sum_r L_r(p, d)$$

The 1-D minimum cost path for a given direction, *L_r(p,d)*, is computed as shown in the equation.

$$L_r(p, d) = C(p, d) + min(L_r(p - r, d), L_r(p - r, d - 1) + P1, L_r(p - r, d + 1) + P1, \min_i L_r(p - r, i) + P2) - \min_k L_r(p - r, k)$$

where

$L_r(p, d)$ = *current cost of pixel p and disparity d in direction r*

$C(p, d)$ = *matching cost at pixel p and disparity d*

$$L_r(p-r, d-1) = previous\ cost\ of\ pixel\ in\ r\ direction\ at\ disparity\ d-1$$

$$L_r(p-r, d+1) = previous\ cost\ of\ pixel\ in\ r\ direction\ at\ disparity\ d+1$$

$$\min_i L_r(p-r, i) = minimum\ cost\ of\ pixel\ in\ r\ direction\ for\ previous\ computation$$

$$P1,\ P2 = penalty\ for\ discontinuity$$

As mentioned earlier, this example uses five directions for disparity computation. Propagation in each direction is independent. The resulting disparities at each level from each direction are aggregated for each pixel. Total cost is the sum of the cost calculated for each direction.

The third module of SGBM algorithm is Post-processing. This module has three steps: minimum cost index calculation, interpolation, and a uniqueness function. Minimum cost index calculation finds the index corresponding to the minimum cost for a given pixel. Sub-pixel quadratic interpolation is applied on the index to resolve disparities at the sub-pixel level. The uniqueness function ensures reliability of the computed minimum disparity. A higher value of the uniqueness threshold marks more disparities unreliable. As a last step, the negative disparity values are invalidated and replaced with -1.

HDL Implementation

The figure below shows the overview of the example model. The blocks leftImage and rightImage import a stereo image pair as input to the algorithm. In the Input subsystem, the Frame To Pixels block converts input images from the leftImage and rightImage blocks to a pixel stream and accompanying control signals in a `pixelcontrol` bus. The pixel stream is passed as input to the SGBMHDLAlgorithm subsystem which contains three computation modules described above: matching cost calculation, directional cost calculation, and post-processing. The output of the SGBMHDLAlgorithm subsystem is a disparity value pixel stream. In the Output subsystem, the Pixels To Frame block converts the output to a matrix disparity map. The disparity map is displayed using the Video Viewer block.

```
modelname = 'SGBMDisparityExample';
open_system(modelname);
set_param(modelname,'SampleTimeColors','off');
set_param(modelname,'Open','on');
set_param(modelname,'SimulationCommand','Update');
set(allchild(0),'Visible','off');
```

FPGA Implementation of Stereo Disparity using SGBM
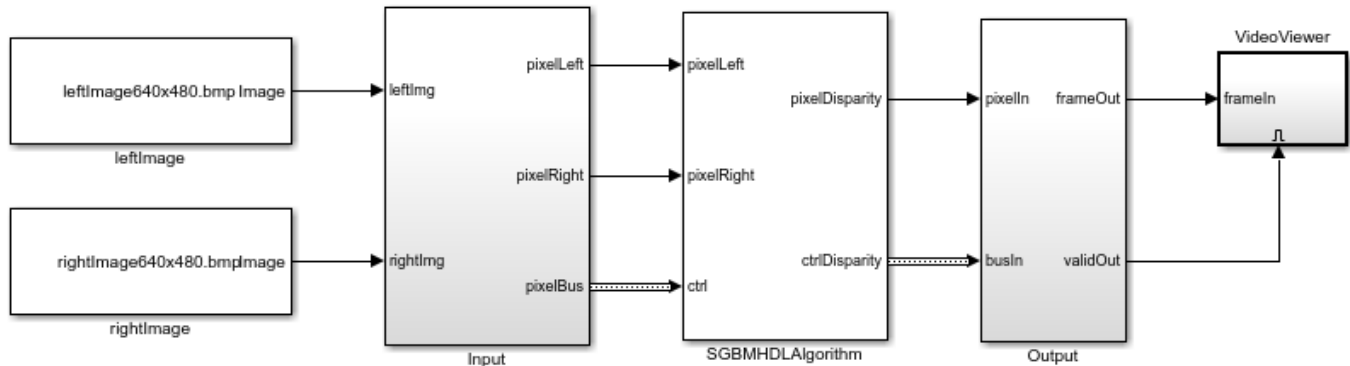
How to Use This Model?
Step 1 : Provide the path to rectified input images in the **leftImage** and **rightImage** blocks.
Step 2 : Run simulation and observe the output in the **VideoViewer**. The result is exported to the workspace with variable names dispMap and dispMapValid.

Parameter Selection:
Step 1 : Double-click the **SGBMHDLAlgorithm** block.
Step 2 : Enter Disparity Levels - an integer value between 8 and 128 [Default=64]
Step 3 : Enter a Uniqueness Threshold - an integer value between 5 and 15 [Default=5]

Matching Cost Calculation

The matching cost calculation is again separated into two parts: CSCT computation and Hamming distance calculation. CSCT is calculated on each 9-by-7 pixel window by aligning each group of pixels for comparison using Tapped Delay blocks, For Each Subsystem blocks and buffers. The input pixels are padded with zeros to allow CSCT computation for the corner pixels. The resulting stream of pixels is passed to ctLogic subsystem. Figure below shows ctLogic subsystem which uses the Tapped Delay block to generate a group of pixels. The pixels are buffered for `imgColSize` cycles, where *imgColSize* is the number of pixels in an image line. A group of pixels that is aligned for comparison is generated from each row. The For Each block and Logical Operator block replicate the comparison logic for each pixel of the input vector size. To implement a 9-by-7 window, the model uses four such For Each blocks. The result generated by each For Each block is a vector which is further concatenated to form a vector of size 31-bits. After Bit Concat is used, the output data type is `uint5`. CSCT and zero-padding operations are performed separately on the left and right input images and the results are passed to the Hamming Distance subsystem.

```
open_system('SGBMDisparityExample/SGBMHDLAlgorithm/MatchingCost/CensusTransform/ctLogic','force')
```
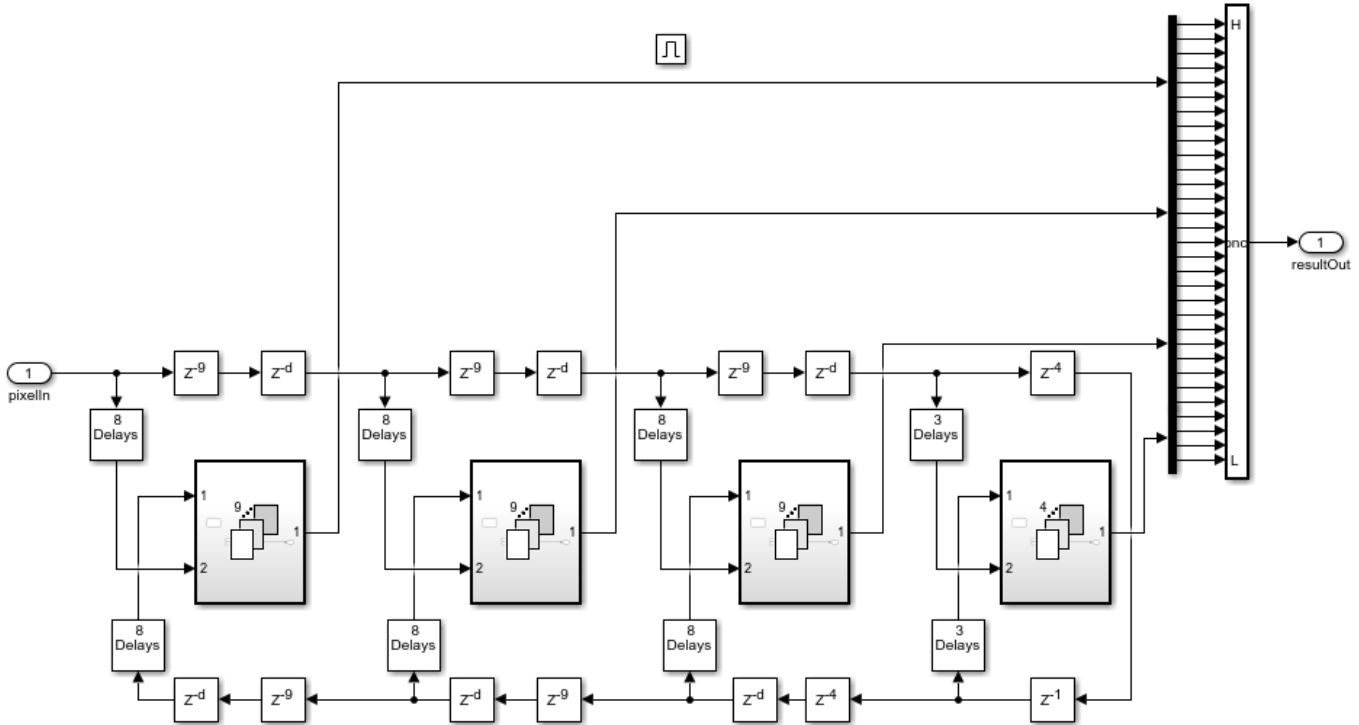
In the Hamming Distance subsystem, the 65th result of the left CSCT is XOR'd with the 65th to 2nd results of the right CSCT. The set bits are counted to obtain Hamming distance. This distance must be calculated for each disparity level. The right CSCT result is passed to the enabledTappedDelay subsystem to generate a group of pixels which is then XOR'd with the left CSCT result using For Each block. The For Each block also counts the set bits in the result. The For Each block replicates the Hamming distance calculation for each disparity level. The result is a vector, with 64 disparity levels corresponding to each pixel. This vector is the Matching Cost, and it is passed to the Directional Cost subsystem.

```
open_system('SGBMDisparityExample/SGBMHDLAlgorithm/MatchingCost/HammDistA','force');
```
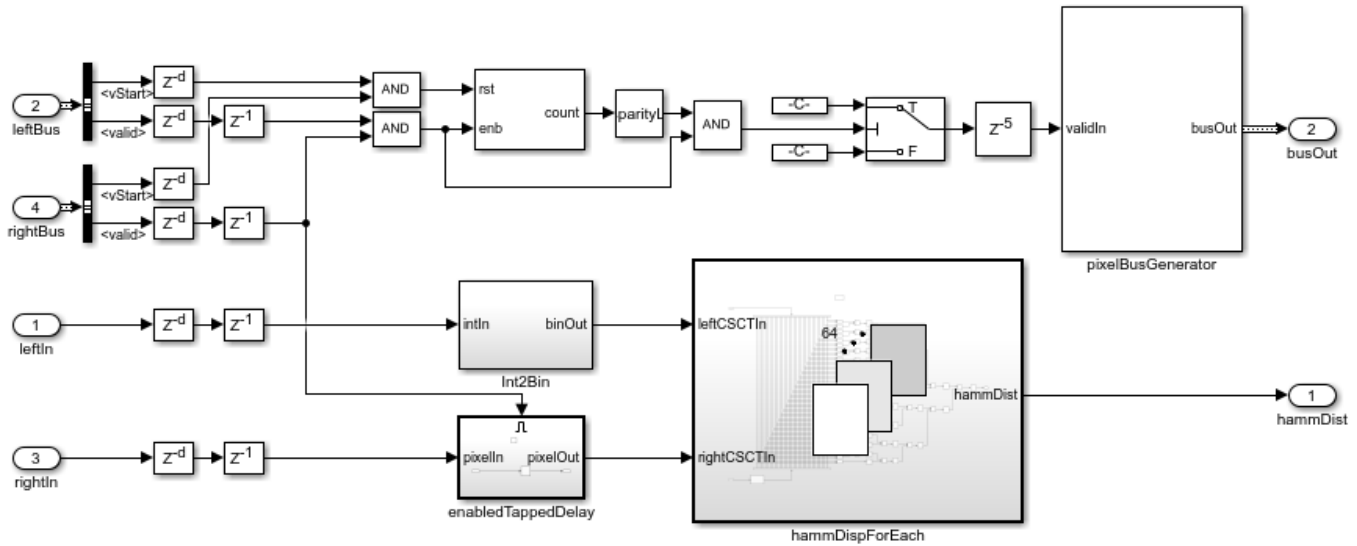
Directional Cost Calculation

The Directional Cost subsystem computes disparity at each pixel in multiple directions. The five directions used in the example are left-to-right (A1), top-left-to-bottom-right (A2), top-to-bottom (A3), top-right-to-bottom-left (A4), and right-to-left (A5). As the cost aggregation at each pixel in each direction is independent of each other, all five directions are implemented concurrently.

Each directional analysis is investigating the previous cost value with respect to the current cost value. The value of previous cost required to compute the current cost for each pixel depends on the direction under consideration. The figure below shows the position of the previous cost with respect to the current cost under computation, for all five directions.
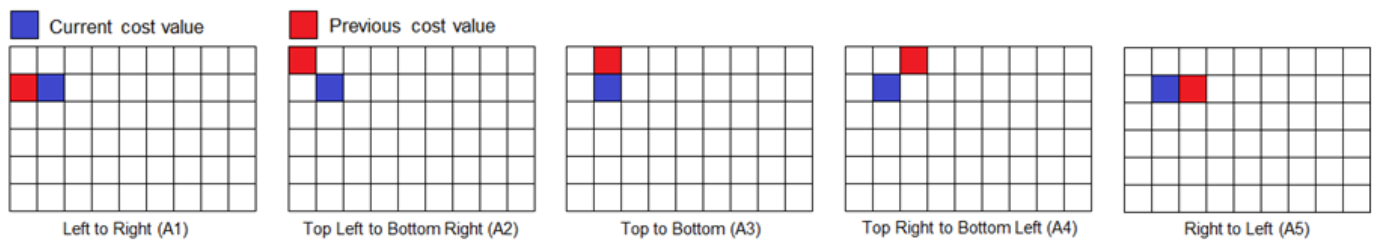
In the figure above, the blue box indicates the position of the current pixel for which current cost values are computed. The red box indicates the position of the previous cost values to be used for current cost computation. For A1, the current cost becomes the previous cost value for the next computation when traversing from left to right. Thus, the current cost value should be immediately fed back to compute the next current cost, as described in [3]. For A2, when traversing from left to right, current cost value should be used as the previous cost value after *imgColSize+1* cycles. Current cost values are hence buffered for cycles equal to *imgColSize+1* and then fed back to compute the next current cost.

Similarly, for A3 and A4, the current cost values are buffered for cycles equal to *imgColSize* and *imgColSize-1*, respectively. However, for A5, when traversing from left to right, the previous cost value is not available. Thus, the direction of traversal to compute A5 is reversed. This adjustment is

done by reversing the input pixels of each row. The current cost value then becomes the previous cost value for the next current cost computation, similar to A1.

The 1-D minimum cost path computes the current cost at d disparity position, using the Matching Cost value, the previous cost values at disparities $d-1$, d, and $d+1$, and the minimum of the previous cost values. The figure below shows the minimum cost path subsystem, which computes the current cost at a disparity position for a pixel.

open_system('SGBMDisparityExample/SGBMHDLAlgorithm/DirectionalCost/LeftToRight/lrSubsystem/minCos
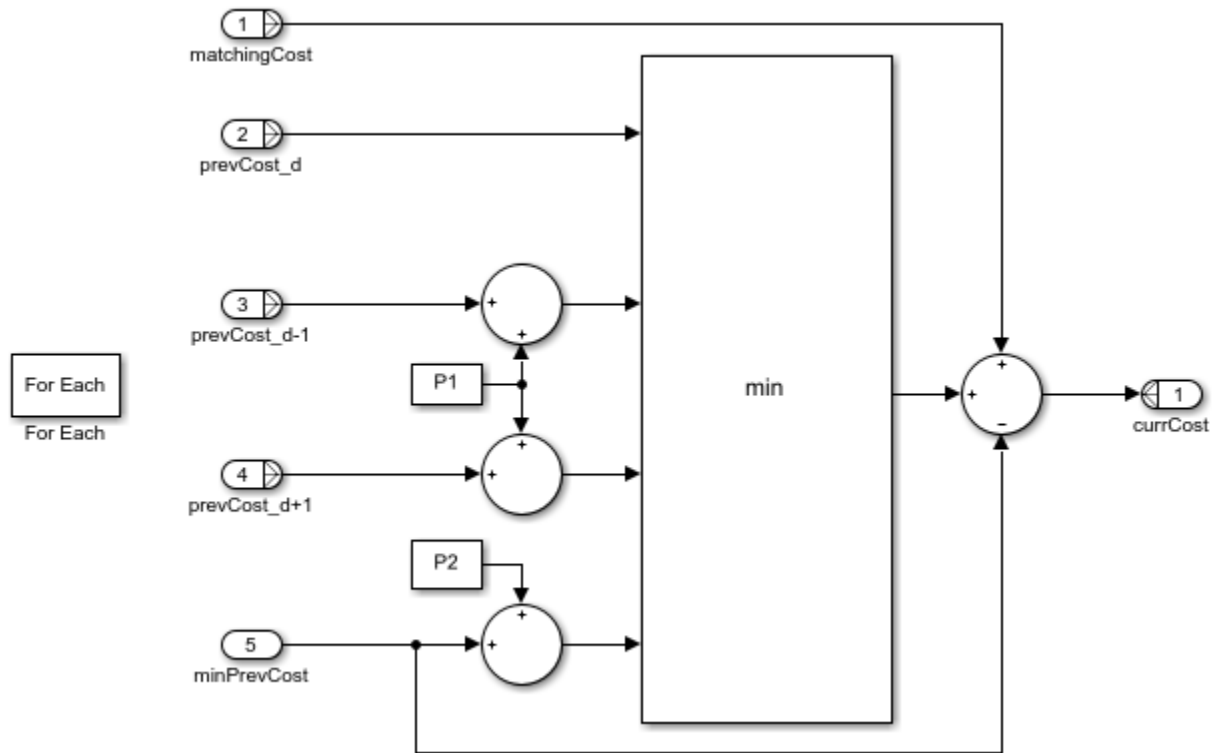
The For Each block is used to replicate the minimum cost path calculation for each disparity level, for each direction. The figure below shows the implementation of A1 for 64 disparity levels. As shown in the figure, 64 minimum cost path calculations are generated as represented by minCostPath subsystem. The matching cost is an input from the Hamming Distance subsystem. The current cost computed by the minCostPath subsystem is immediately fed back to itself as the previous cost values, for the next current cost computation. Thus, values for *prevCost_d* are now available. Values for *prevCost_d-1* are obtained by shifting the 1st to 63rd fed-back values to the 2nd to 64th positions. The d-1 subsystem contains a Selector block that shifts the position of the values, and fills in zero at the 1st position.

Similarly, values for *prevCost_d+1* position are obtained by shifting the 2nd to 64th feedback values to the 1st to 63rd position and inserting a zero at the 64th position. The current cost computed is also passed to the min block to compute the minimum value from the current cost values. This value is fed back to the *minPrevCost* input of the minCostPath subsystem. The next current cost is then computed by using the current cost values, acting as previous cost values, in the next cycle for A1. Since the minimum cost of disparity levels from the previous set is immediately needed for the current set, this feedback path is the critical path in the design.

```
open_system('SGBMDisparityExample/SGBMHDLAlgorithm/DirectionalCost/LeftToRight/lrSubsystem','forc
```
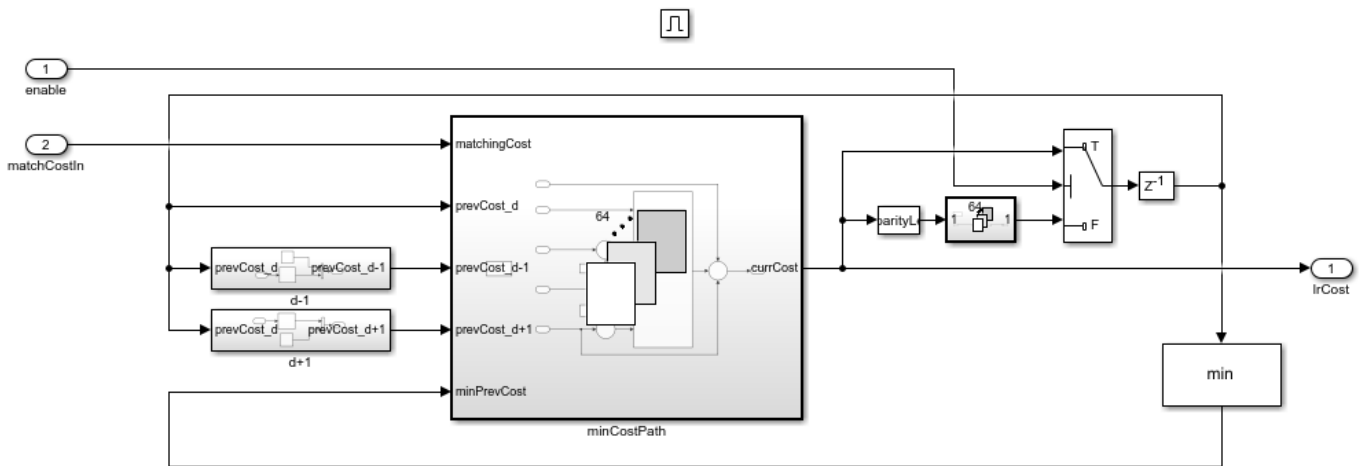
The current cost computations for A2, A3, and A4 are implemented in the same manner. Since the current cost value is not immediately required for these directions, there is a buffer in both feedback paths. This buffer prevents this feedback path from being the critical path in the design. The figure below shows the A3 implementation with a buffer in the feedback paths.

```
open_system('SGBMDisparityExample/SGBMHDLAlgorithm/DirectionalCost/TopToBottom/tbSubsystem','forc
```
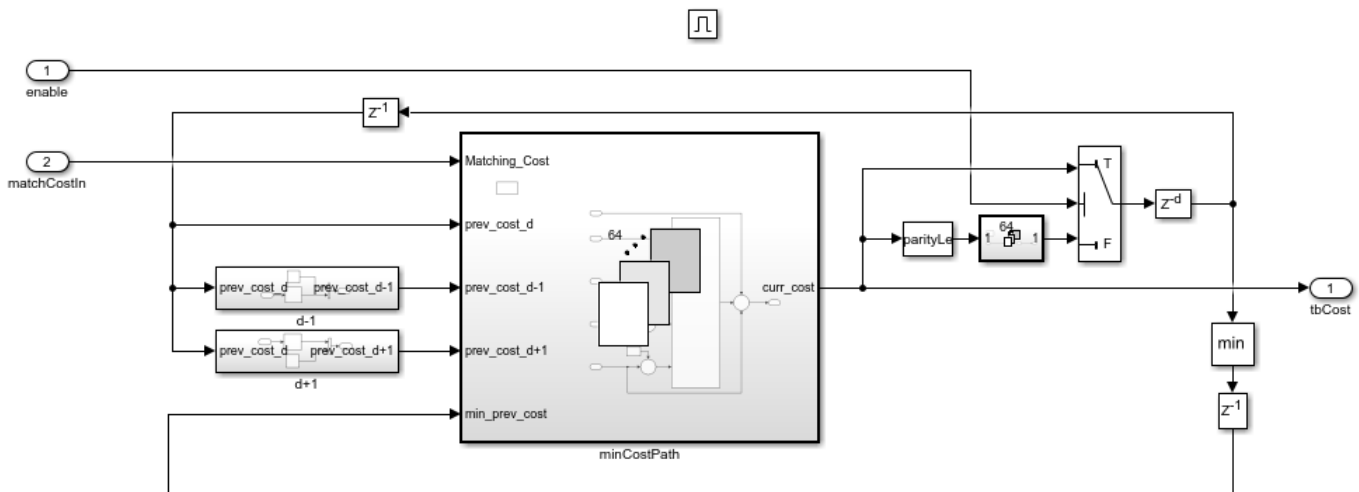
The current cost calculation for A5 has additional logic to reverse the rows at input and again reverse the rows at output to match the pixel positions for the total cost calculation. A single buffer of *imgColSize* cycles achieves this reversal. Since all directions are calculated concurrently, the time required to reverse the rows must be compensated for on the other paths. Delay equivalent to *2*imgColSize* cycles is introduced in the other four directions. To optimize resources, instead of buffering 64 values of matching cost for each pixel, the 31-bit result of CSCT is buffered. A separate Hamming Distance module is then required to compute matching cost for A5. This design reduces on-chip memory usage. The rows are reversed after the CSCT computation and matching cost is calculated using a separate Hamming Distance module that provides the Matching Cost input to A5. Also, dataAligner subsystem is used to remove data discontinuity for each row before passing it to Hamming Distance subsystems. This helps easy synchronization of data at time of aggregation. The

current cost obtained from all five directions at each pixel are aggregated to obtain the total cost at each pixel. The total cost is passed to the Post-processing subsystem.

Post-Processing

In the post-processing subsystem, the index of the minimum cost is calculated at each pixel position from 64 disparity levels by using Min blocks in a tree architecture. The index value obtained is the disparity of each pixel. Along with minimum cost index computation, the minimum cost value at the computed index, and the cost values at *index-1* and *index+1* are also computed. The Minimum_Cost_Index subsystem implements tree architecture to compute a minimum value from 128 values. 64 disparity values are padded with 64 more values to make a vector of 128 values. Minimum value is computed from this vector with 128 values. In case, a vector with 128 values is available no value is padded to a vector or in other words, vector is passed directly for minimum value calculation. Variant Subsystem, Variant Model is used to select between logic using variant subsystem variables. Sub-pixel quadratic interpolation is then applied to the index to resolve disparity at sub-pixel level. Also, a uniqueness function is applied to the index calculated by min blocks, to ensure reliable disparity results. As a last step, invalid disparities are identified and replaced with -1.

Model Parameters

The model presented here takes disparity levels and uniqueness threshold as input parameters as shown in figure. Disparity levels is an integer value from 8 to 128 with the default value of 64. Higher value of disparity level reduces the minimum distance detected. Also, for larger input image size larger disparity level helps better detection of depth of object. The uniqueness threshold must be a positive integer value, between 0 and 100 with a typical range from 5 to 15. Lower value of uniqueness threshold marks more disparities reliable. The default value of uniqueness threshold is 5.
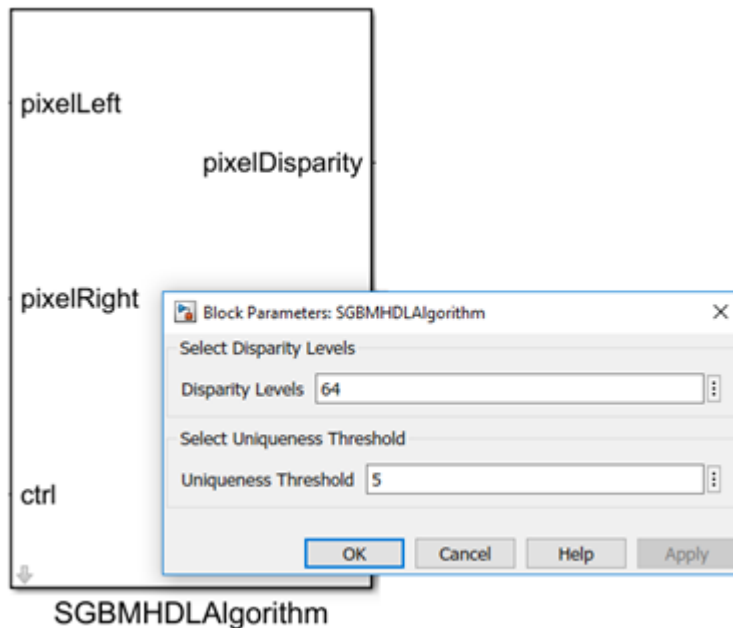
Simulation and Results

The example model can be simulated by specifying a path for the input images in the leftImage and rightImage blocks. The example uses sample images of size 640-by-480 pixels. The figure shows a sample input image and the calculated disparity map. The model exports these calculated disparities

and a corresponding valid signal to the MATLAB workspace, using variable names `dispMap` and `dispMapValid` respectively. The output disparity map is 576-by-480 pixels, since the first 64 columns are unused in the disparity computation. The unused pixels are padded with 0 in Output subsystem to generate output image of size 640-by-480 as shown in Video Viewer. A disparity map with colorbar is generated using the commands shown below. Higher disparity values in the result indicate that the object is nearer to the camera and lower disparity values indicate farther objects.

```
dispMapValid = find(dispMapValid == 1);
disparityMap = (reshape(dispMap(dispMapValid(1:imgRowSize*imgColSize),:),imgColSize,imgRowSize))
figure(); imagesc(disparityMap);
title('Disparity Map');
colormap jet; colorbar;
```


leftImage640x480

rightImage640x480

Video Viewer

Disparity Map
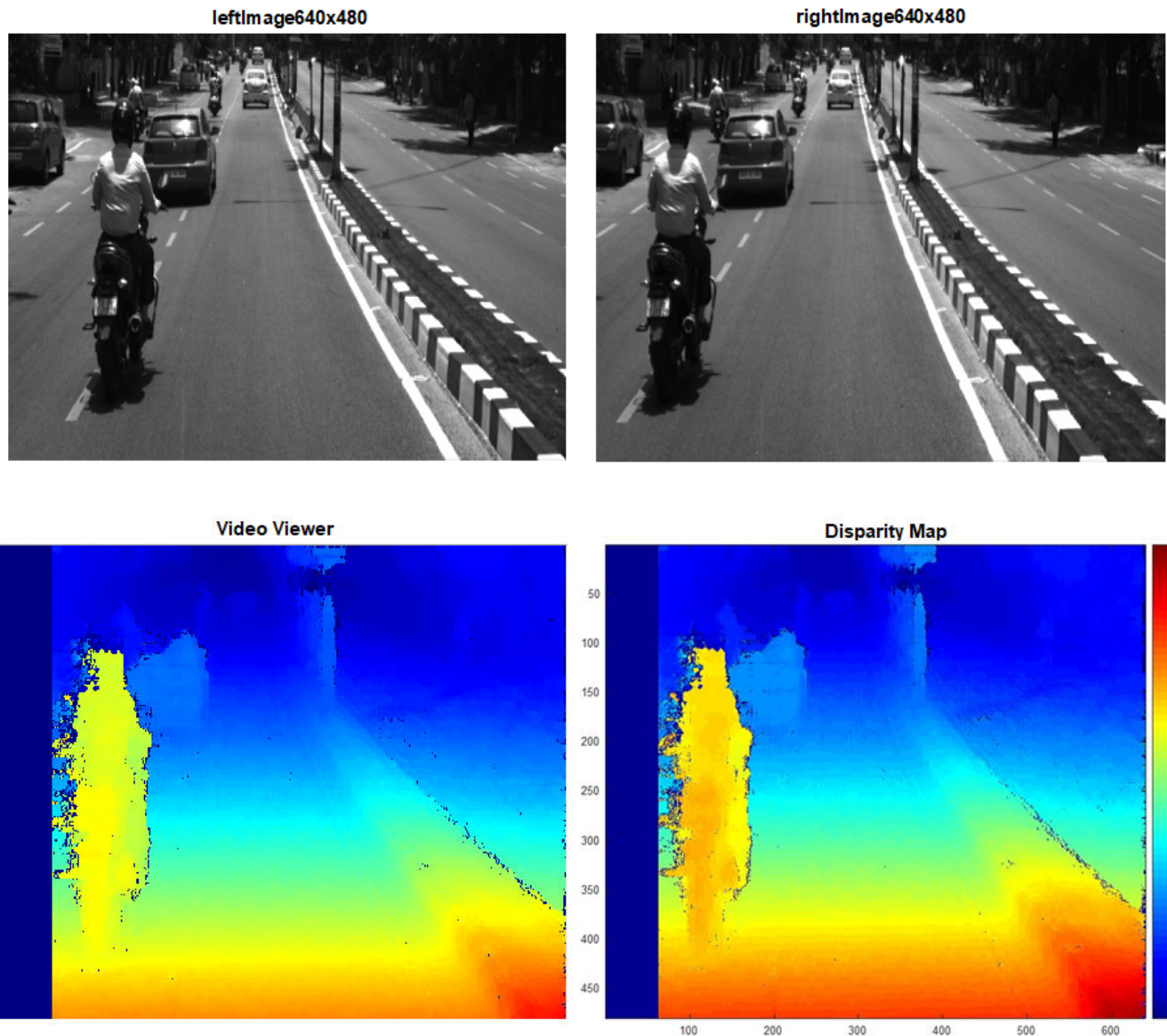
The example model is compatible to generate HDL code. You must have an HDL Coder™ license to generate HDL code. The design was synthesized for the Intel® Arria® 10 GX (115S2F45I1SG) FPGA.

The table below shows resource utilization for three disparity level at different image resolutions. Considering one pair of stereo input images as a frame, the algorithm throughput is estimated by finding the number of clock cycles required for processing the current frame before the arrival of next frame. The core algorithm throughput, without overhead of buffering input and output data, is the maximum operating frequency divided by the minimum cycles required between input frames. For example, for 128 disparity levels and 1280-by-720 image resolution, the minimum cycles to process the input frame is 938,857 clock cycles/frame. The maximum operating frequency obtained for algorithm with 128 disparity levels is 61.69 MHz, the core algorithm throughput is computed as 65 frames per second.

```
% ================================================================================
% |Disparity Levels       ||       64      ||       96      ||       128          |
% ================================================================================
% |Input Image Resolution ||    640 x 480   ||    960 x 540   ||    1280 x 720     |
% |ALM Utilization        ||   45,613 (11%) ||   64,225 (15%) ||    85,194 (20%)   |
% |Total Registers        ||    49,232      ||    64,361      ||    85,564         |
% |Total Block Memory Bits || 3,137,312 (6%) || 4,599,744 (9%) || 11,527,360 (21%) |
% |Total RAM Blocks        ||    264 (10%)   ||    409 (16%)    ||    741 (28%)     |
% |Total DSP Blocks        ||     65 (4%)    ||     97 (6%)     ||    129 (8%)      |
% ================================================================================
```

References

[1] Hirschmuller H., Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information, International Conference on Computer Vision and Pattern Recognition, 2005.

[2] Spangenberg R., Langner T., and Rojas R., Weighted Semi-Global matching and Center-Symmetric Census Transform for Robust Driver Assistance, Computer Analysis of Images and Patterns, 2013.

[3] Gehrig S., Eberli F., and Meyer T., A Real-Time Low-Power Stereo Vision Engine Using Semi-Global Matching, International Conference on Computer Vision System, 2009.

Stereo Image Rectification

This example shows how to implement stereo image rectification for a calibrated stereo camera pair. The example model is FPGA-hardware compatible and provides real-time performance. This example compares its results with the Computer Vision Toolbox™ `rectifyStereoImages` function.

Introduction

A stereo camera is a camera system with two or more lenses with a separate image sensor for each lens. They are used for distance estimation, making 3-D pictures, and stereoviews. Camera lenses distort images, and it is difficult to align two cameras to be perfectly parallel. So, the raw images from a pair of stereo cameras must be rectified. Stereo image rectification projects images onto a common image plane in such a way that the corresponding points in the two stereo images have the same row coordinates. This image projection corrects the images to appear as if the two cameras are parallel.

The algorithm used in this example performs distortion removal and alignment correction in a single system.

Stereo Image Rectification Algorithm

The stereo image rectification algorithm uses a reverse mapping technique to map the pixel locations of the output rectified image to the pixels in the input camera image. The diagram shows the four stages of the algorithm.
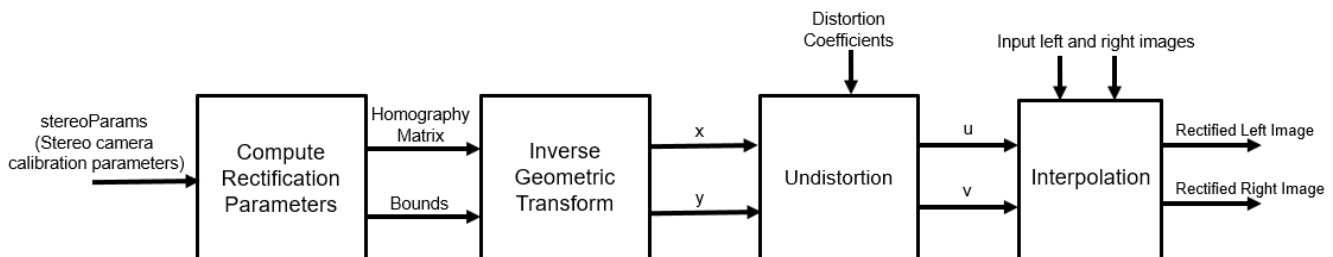
Compute Rectification Parameters: This stage computes rectification parameters from input stereo camera calibration parameters. These calibration parameters include camera intrinsics, rotation matrices, translation matrices, and distortion coefficients (radial and tangential). This stage returns a homography matrix for each camera, and the output bounds. The output bounds are needed to compute the integer pixel coordinates of the output rectified image, and the homography matrices are needed to transform integer pixel coordinates in the output rectified image to corresponding coordinates of the undistorted image.

Inverse Geometric Transform: An inverse geometric transformation translates a point in one image plane onto another image plane. In stereo image rectification, this operation maps integer pixel coordinates in the output rectified image to the corresponding coordinates of the input camera image by using the homography matrix, *H*. If *(p,q)* is an integer pixel coordinate in the rectified output image and *(x,y)* is the corresponding coordinate of the undistorted image, then this equation describes the transformation.

$$[x \ \ y \ \ z]_{1X3} = [p \ \ q \ \ 1]_{1X3} * H_{3X3}^{-1}$$

where *H* is the homography matrix. To convert from homogeneous to cartesian coordinates, *x* is set to *x/z* and *y* is set to *y/z*.

Undistortion: Lens distortions are optical aberrations which may deform the images. There are two main types of lens distortions: radial and tangential distortions. Radial distortion occurs when light rays bend more near the edges of a lens than they do at its optical center. Tangential distortion occurs when the lens and the image plane are not parallel. For distortion removal, the algorithm maps the coordinates of the undistorted image to the input camera image by using distortion coefficients.

If *(u,v)* is the coordinate of the input camera image and *(x,y)* is the corresponding coordinate of the undistorted image, then this equation describes the undistortion operation.

$$u_{radial} = x(1 + k_1 r^2 + k_2 r^4) \ , \ u_{tangential} = 2p_1 xy + p_2(r^2 + 2x^2)$$

$$v_{radial} = y(1 + k_1 r^2 + k_2 r^4) \ , \ v_{tangential} = 2p_2 xy + p_1(r^2 + 2y^2)$$

where $r^2 = x^2 + y^2$.

k_1, k_2 are *radial distortion coefficients* and p_1, p_2 are *tangential distortion coefficients*.

$$u = u_{radial} + u_{tangential}$$

$$v = v_{radial} + v_{tangential}$$

Inverse geometric transformation and undistortion both contribute to an overall mapping between the coordinates of the output undistorted rectified image *(u,v)* and the coordinates of the input camera image.

Interpolation: Interpolation resamples the image intensity values corresponding to the generated coordinates. The example uses bilinear interpolation.
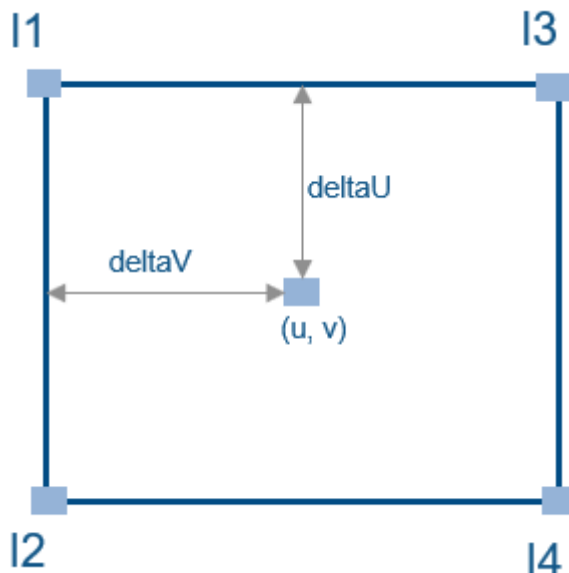
As shown in the diagram, *(u,v)* is the coordinate of the input pixel generated by the undistortion stage. *I1*, *I2*, *I3*, and *I4* are the four neighboring pixels, and *deltaU* and *deltaV* are the displacements of the target pixel from its neighboring pixels. This stage computes the weighted average of the four neighboring pixels by using this equation.

$$rectifiedPixel = I_1(1 - deltaU)(1 - deltaV) + I_2(deltaU)(1 - deltaV) + I_3(1 - deltaU)(deltaV) + I_4(deltaU)(deltaV)$$

HDL Implementation

The figure shows the top-level view of the StereoImageRectificationHDL model. The LeftInputImage and RightInputImage blocks import the stereo left and right images from files. The Frame To Pixels blocks convert these stereo image frames to pixel streams with `pixelcontrol` buses for input to the HDLStereoImageRectification subsystem. This subsystem performs the inverse geometric transform, undistortion, an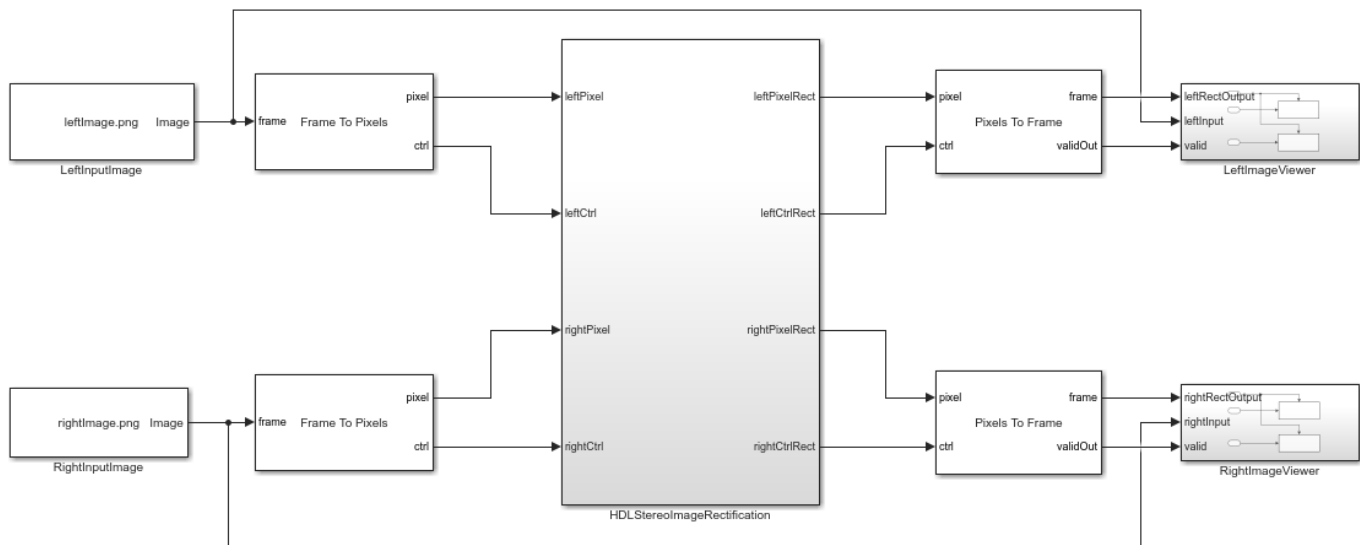d interpolation to generate the rectified output pixel values. The Pixels To Frame blocks convert the streams of output pixels back to frames. The LeftImageViewer and RightImageViewer subsystems display the input frames and their corresponding rectified outputs.

The InitFcn of the example model imports the stereo calibration parameters from a data file and computes the rectification parameters by calling `ComputeRectificationParams.m`. Alternatively, you can generate your own set of rectification parameters and provide them as mask parameters of the InverseGeometricTransform and Undistortion subsystems.

The HDLStereoImageRectification subsystem generates a single `pixelcontrol` bus from the two input **ctrl** busses. The RectifiedCoordinateGeneration subsystem generates the row and column pixel coordinates of the output rectified and undistorted image. It uses two HDL counters to generate the row and column coordinates. The InverseGeometricTransform subsystems map these coordinates onto their corresponding row and column coordinates, *(x,y)*, of the distorted image. The Undistortion subsystems map the *(x,y)* coordinates to its corresponding coordinate *(u,v)* of the input camera image, using the distortion coefficients and stereo camera intrinsics.

The Interpolation subsystems store the pixel intensities of the input stereo images in a memory and calculate the addresses of the four neighbors of *(u,v)* required for interpolation. To calculate each rectified output pixel intensity, the subsystem reads the four neighbor pixel values and finds their weighted sum.
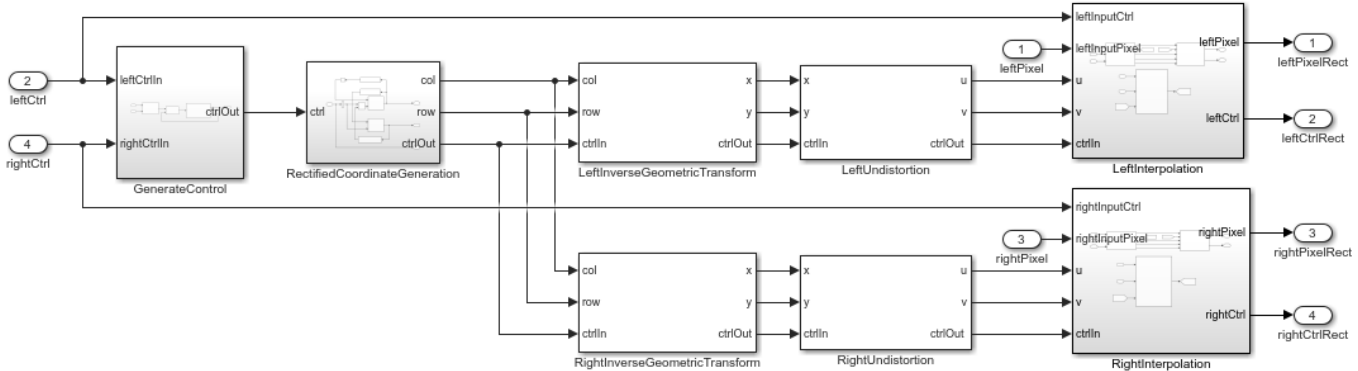
Inverse Geometric Transformation

The HDL implementation of inverse geometric transformation multiplies the coordinates *[row col 1]* with the inverse homography matrix. The inverse homography matrix (3-by-3) is a masked parameter of the InverseGeometricTransformation subsystem. ComputeRectificationParams.m, called in the InitFcn of the model, generates the homography matrix. The Transformation subsystem implements the matrix multiplication with Product blocks that multiply by each element of the homography matrix. The HomogeneousToCartesian subsystem converts the generated homogeneous coordinates, *[x y z]* back to the cartesian format, *[x y]* for further processing. The HomogeneousToCartesian subsystem uses a Reciprocal block configured to use the ShiftAdd architecture, and the **UsePipelines** parameter is set to 'on'. To see these parameters, right-click the block and select **HDL Code > HDL Block Properties**. Until this stage, the word length was allowed to grow with each operation. After the HomogeneousToCartesian subsystem, the word length of the coordinates is truncated to a size that still ensures precision and accuracy of the generated coordinates.
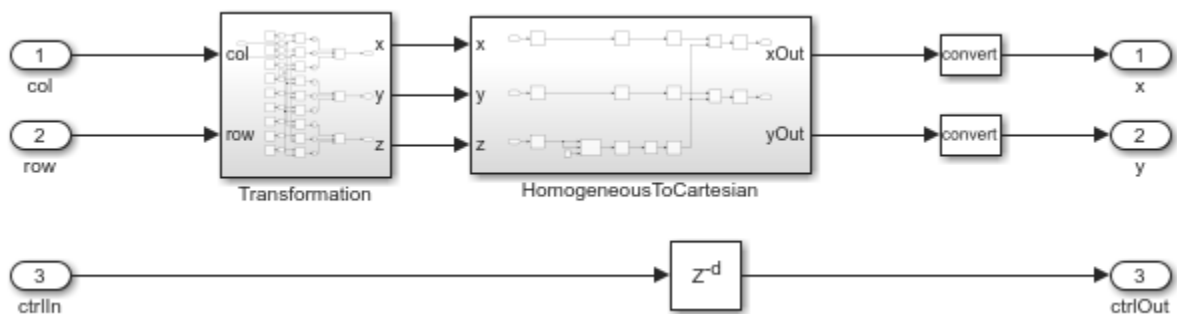
Undistortion

The HDL implementation of Undistortion takes the 3-by-3 camera intrinsic matrix, distortion coefficients *[k1 k2 p1 p2]*, and the reciprocal of *fx* and *fy* as masked parameters. ComputeRectificationParams.m, which is called in the InitFcn of the model, generates these parameters. The intrinsic matrix is defined as $\begin{bmatrix} fx & skew & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$.

The Undistortion subsystem implements the equations mentioned in the Stereo Image Rectification Algorithm section by using Sum, Product, and Shift arithmetic blocks. The word length is allowed to

grow with each operation, and then the Denormalization subsystem truncates the word length to a size that still ensures the precision and accuracy of the generated coordinates.
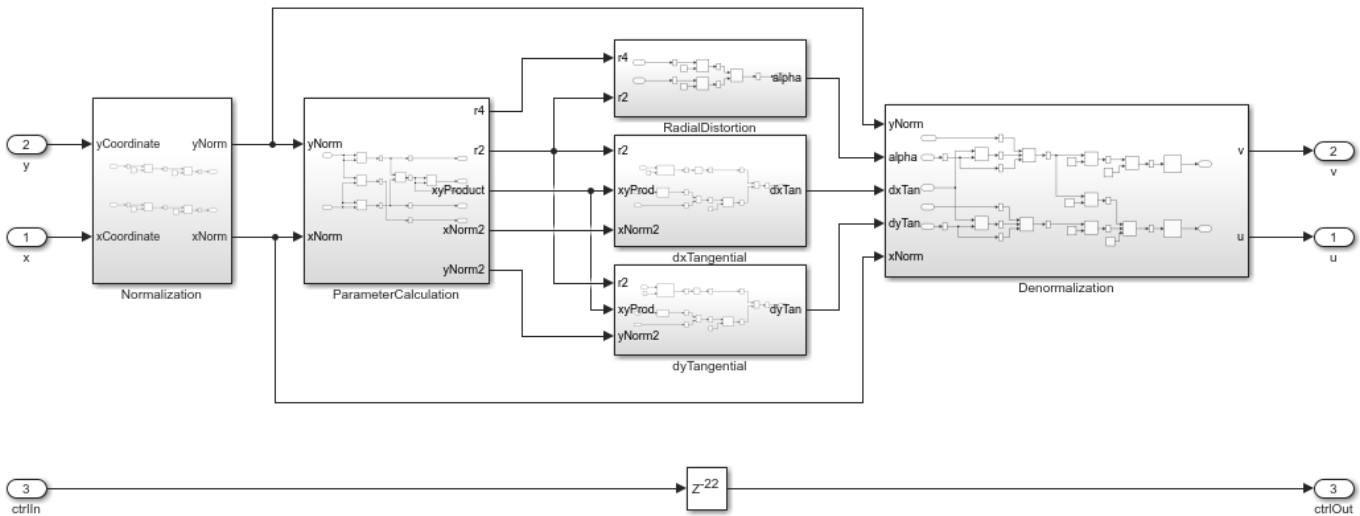
Interpolation

These sections describe the three components inside the Interpolation subsystem.
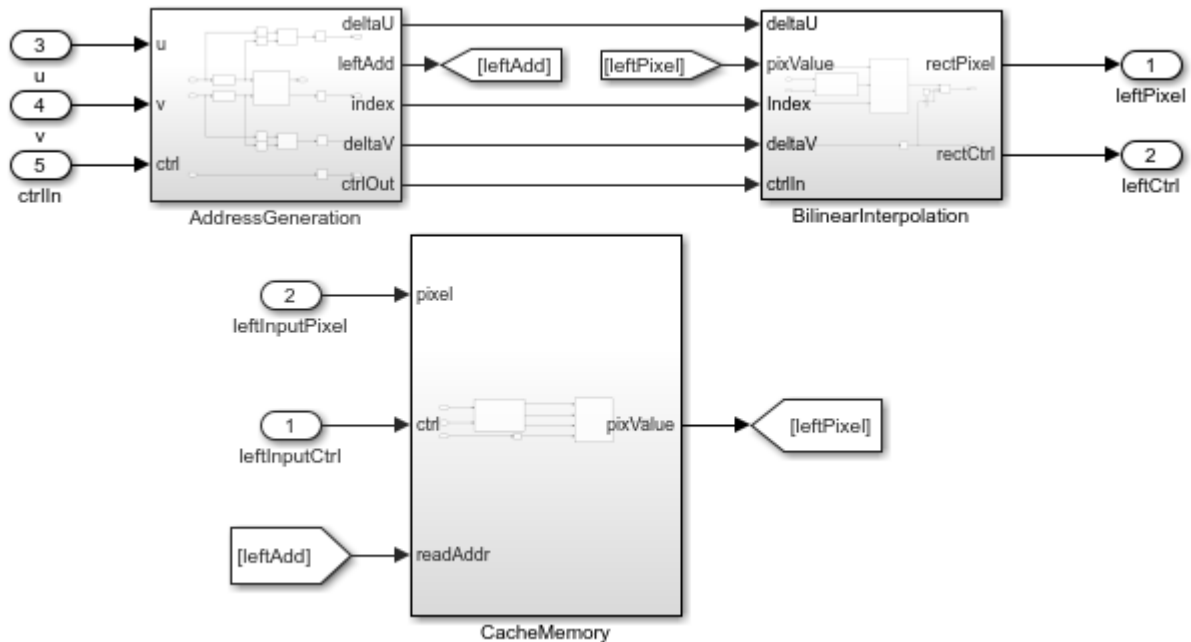
Address Generation

The AddressGeneration subsystem takes the mapped coordinate of the input raw image *(u,v)* as input. It calculates the displacement *deltaU* and *deltaV* of each pixel from its neighboring pixels. It also rounds the coordinates to the nearest integer toward negative infinity.

The AddressCalculation subsystem checks the coordinates against the bounds of the input images. If any coordinate is outside the image dimensions, is capped to the boundary value for further processing. Next, the subsystem calculates the index of the address of each of the four neighborhood pixels in the CacheMemory subsystem. The index represents the column of the cache. The index for each address is determined using the even and odd nature of the incoming column and row coordinates, as determined by the Extract Bits block.

```
% ==========================
% |Row   || Col   || Index ||
% ==========================
% |Odd   || Odd   ||   1   ||
% |Even  || Odd   ||   2   ||
% |Odd   || Even  ||   3   ||
% |Even  || Even  ||   4   ||
% ==========================
```

The address of the neighborhood pixels is generated using this equation:

$$Address = (\frac{Size\,of\,column}{2} * nR) + nC$$

where nR is the row coordinate, and nC is the column coordinate.

$$nR = \frac{row}{2} - 1 \text{, if row is even }, nR = \frac{row-1}{2} \text{, if row is odd}$$

$$nC = \frac{col}{2} \text{, if col is even }, nC = \frac{col+1}{2} \text{, if col is odd}$$

Once all the addresses and their corresponding indices are generated, they are vectorized using a Vector Concatenate block. The IndexChangeForMemoryAccess MATLAB Function block rearranges the addresses in increasing order of their indices. This operation ensures the correct fetching of the data from the CacheMemory block. The addresses are then given as an input to the CacheMemory block, and the *index*, *deltaU*, and *deltaV* are passed to the BilinearInterpolation subsystem.
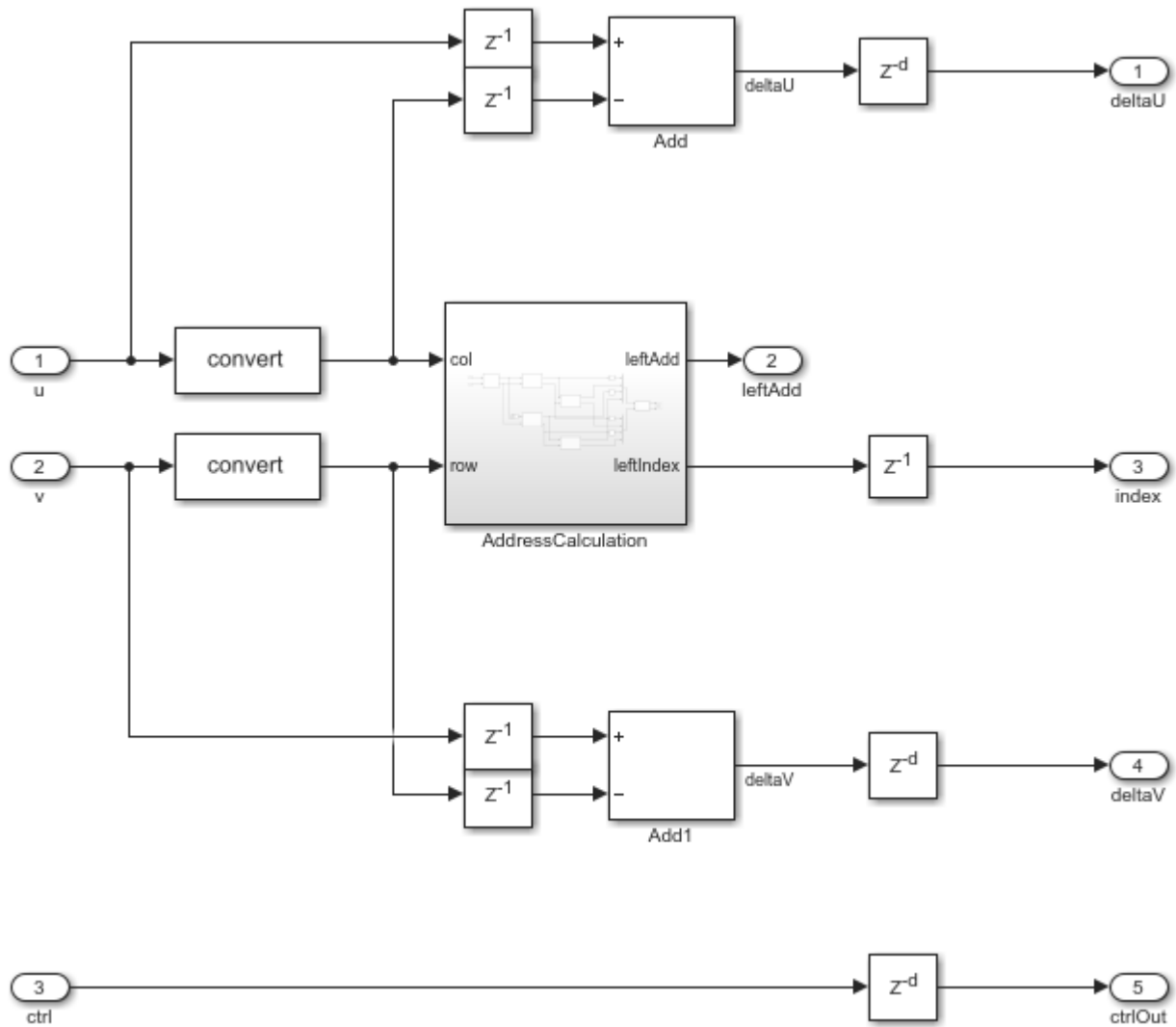
Cache Memory

The CacheMemory subsystem contains a Simple Dual Port RAM block. The input pixels are buffered to form [Line 1 Pixel 1 | Line 2 Pixel 1 | Line 1 Pixel 2 | Line 2 Pixel 2] in the RAM. This configuration enables the algorithm to read all four neighboring pixels in one cycle. The required size of the cache memory is calculated from the *offset* and *displacement* parameters in `ComputeRectificationParams.m` script. The *displacement* is the sum of *maximum deviation* and the *first row map*. The *first row map* is the maximum value of the input image row coordinate that corresponds to the first row of the output rectified image. *Maximum deviation* is the greatest difference between the maximum and minimum row coordinates for each row of the input image row map.

The WriteControl subsystem forms vectors of incoming pixels, and vectors of write enables and write addresses. The AddressGeneration subsystem provides a vector of read addresses. The vector of pixels returned from the RAM are passed to the BilinearInterpolation subsystem.
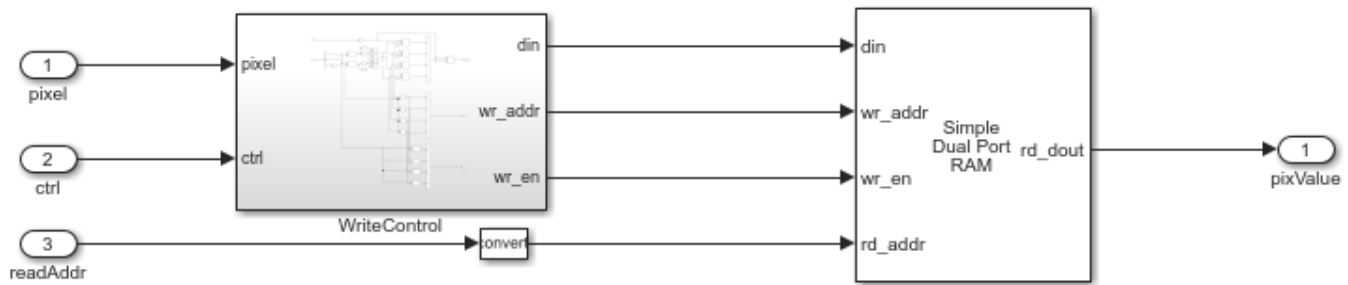
Bilinear Interpolation

The BilinearInterpolation subsystem rearranges the vector of read pixels from the cache to their original indices. Then, the BilinearInterpolationEquation block calculates a weighted sum of the neighborhood pixels by using the bilinear interpolation equation mentioned in the Stereo Image Rectification Algorithm section. The result of the interpolation is the value of the output rectified pixel.
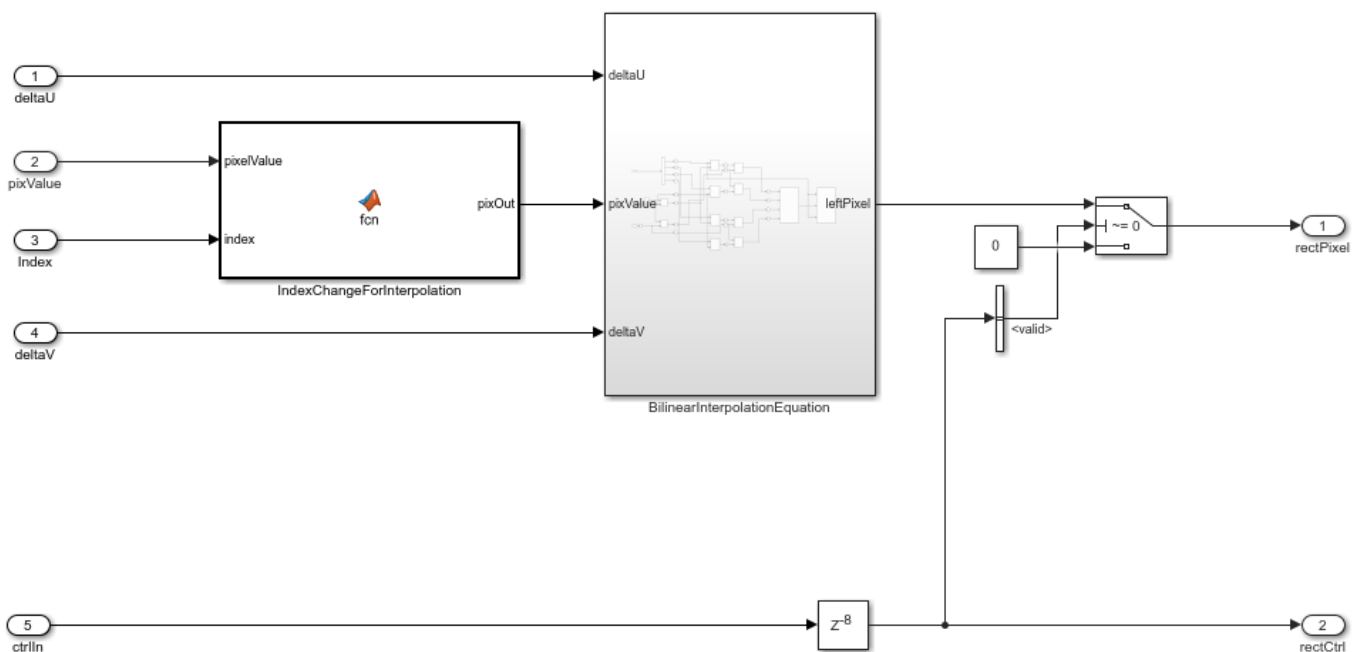
Simulation and Results

This example uses 960-by-1280 stereo images. The input pixels use the `uint8` data type. The example does not provide multipixel support. Due to the large frame sizes used in this example, simulation can take a relatively long time to complete.

The figure shows the left and right input images and the corresponding rectified output images. The results of the StereoImageRectificationHDL model match the output of the `rectifyStereoImages` function in MATLAB with an error of +/-1.
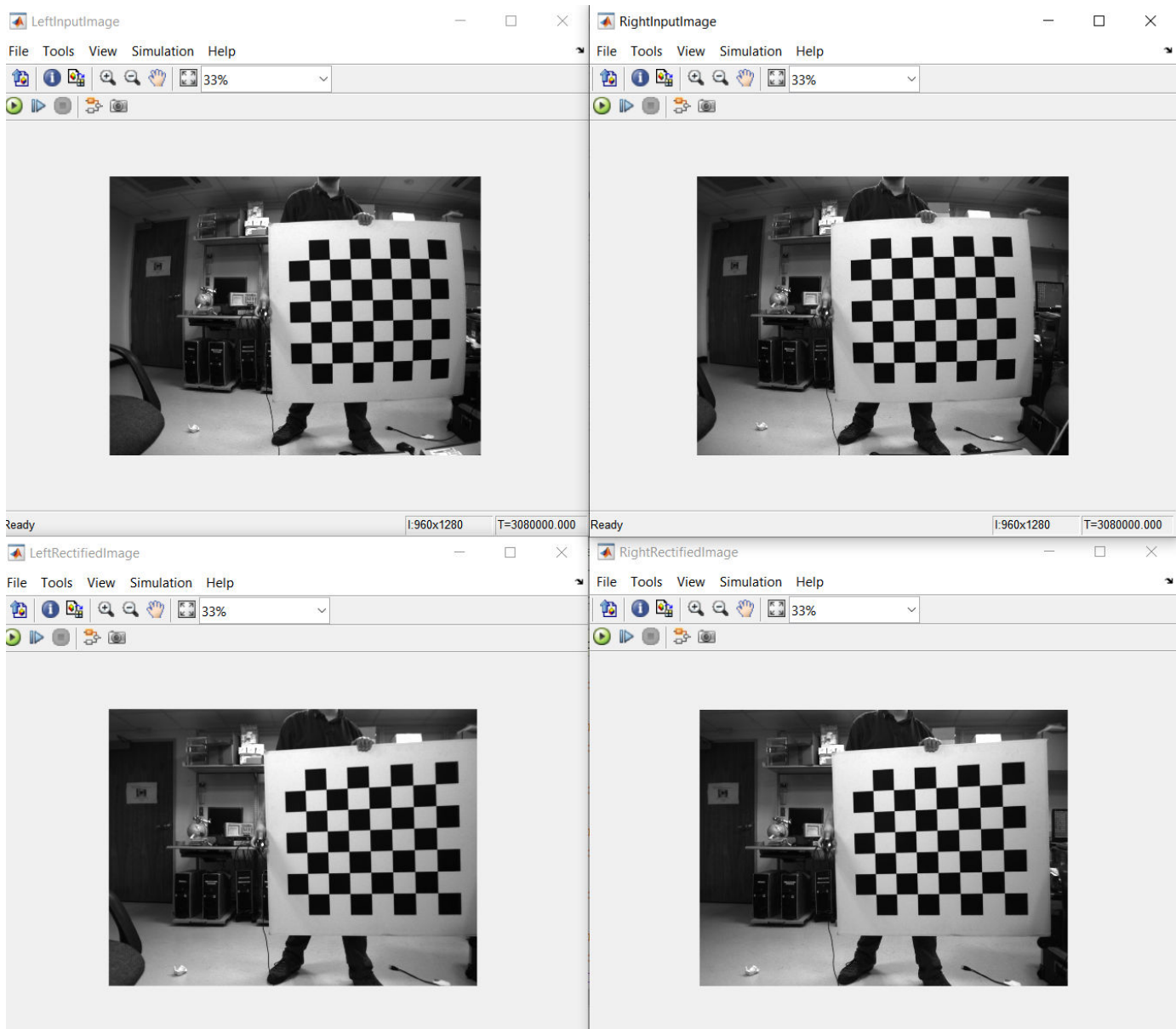
You can generate HDL code for the HDLStereoImageRectification subsystem. You must have an HDL Coder™ license to generate HDL code. This design was synthesized for the Intel® Arria® 10 GX (115S2F45I1SG) FPGA. The HDL design achieves a clock rate of over 150 MHz. The table shows the resource utilization for the subsystem.

```
% =============================================================
% |Model Name             || StereoImageRectificationHDL  ||
% =============================================================
% |Input Image Resolution ||       960 x 1280             ||
% |ALM Utilization        ||          9098                ||
% |Total Registers        ||          21349               ||
% |Total RAM Blocks       ||          333                 ||
% |Total DSP Blocks       ||          200                 ||
% =============================================================
```

References

[1] G. Bradski and A. Kaehler, Learning OpenCV : Computer Vision with the OpenCV Library. Sebastopol, CA: O'Reilly, 2008.

See Also

rectifyStereoImages

FPGA Implementation of Low Light Enhancement

This example shows how to enhance low-light images on an FPGA.

Low-light enhancement (LLE) is a pre-processing step for applications in autonomous driving, scientific data capture, and general visual enhancement. Images captured in low-light and uneven brightness conditions have low dynamic range with high noise levels. These qualities can lead to degradation of the overall performance of computer vision algorithms that process such images. This algorithm improves the visibility of the underlying features in an image.

The example model includes a floating-point frame-based algorithm as a reference, a simplified implementation that reduces division operations, and a streaming fixed-point implementation of the simplified algorithm that is suitable for hardware.

LLE Algorithm

This example performs LLE by inverting an input image and then applying a de-haze algorithm on the inverted image. After inverting the low-light image, the pixels representing non-sky region have low intensities in at least one color channel. This characteristic is similar to an image captured in hazy weather conditions [1]. The intensity of these dark pixels is mainly due to scattering, or airlight, so they provide an accurate estimation of the haze effects. To improve the dark channel in an inverted low-light image, the algorithm modifies the airlight image based on the ambient light conditions. The airlight image is modified using the dark channel estimation and then refined with a smoothing filter. To avoid noise from over-enhancement, the example applies non-linear correction to better estimate the airlight map. Although this example differs in its approach, for a brief overview of low-light image enhancement, see "Low-Light Image Enhancement" (Image Processing Toolbox)(Image Processing Toolbox).

The LLE algorithm takes a 3-channel low-light RGB image as input. This figure shows the block diagram of the LLE Algorithm.
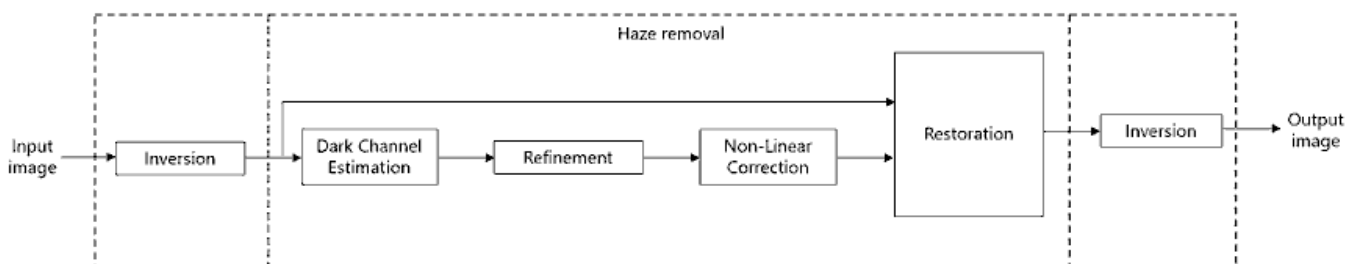
The algorithm consists of six stages.

1. *Scaling and Inversion*: The input image $I^c(x, y), c \in [r, g, b]$ is converted to range [0,1] by dividing by 255 and then inverting pixel-wise.

$$I_{scal}^c(x, y) = \frac{I^c(x, y)}{255}$$

$$I_{inv}^c(x, y) = 1 - I_{scal}^c(x, y)$$

2. *Dark Channel Estimation*: The dark channel is estimated by finding the pixel-wise minimum across all three channels of the inverted image [2]. The minimum value is multiplied by a haze factor, z, that represents the amount of haze to remove. The value of z is between 0 and 1. A higher value means more haze will be removed from the image.

$$I_{air}(x,y) = z \times \min_{c \, \epsilon \, [r,g,b]} I_{inv}^c(x,y)$$

3. *Refinement*: The airlight image from the previous stage is refined by iterative smoothing. This smoothing strengthens the details of the image after enhancement. This stage consists of five filter iterations with a 3-by-3 kernel for each stage. The refined image is stored in $I_{refined}(x,y)$. These equations derive the filter coefficients, h, used for smoothing.

$$I_{refined(n+1)}(x,y) = I_{refined(n)}(x,y) * h, \; n = [0,1,2,3,4] \; \& \; I_{refined(0)} = I_{air}$$

$$where \; h = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$Let \; I_{refined(5)}(x,y) = I_{refined}(x,y)$$

4. *Non-Linear Correction*: To reduce over-enhancement, the refined image is corrected using a non-linear correction equation shown below. The constant, m, represents the mid-line of changing the dark regions of the airlight map from dark to bright values. The example uses an empirically-derived value of $m = 0.6$.

$$I_{nlc}(x,y) = \frac{[I_{refined}(x,y)]^4}{[I_{refined}(x,y)]^4 + m^4}$$

5. *Restoration*: Restoration is performed pixel-wise across the three channels of the inverted and corrected image, I_{nlc}, as shown:

$$I_{restore}^c(x,y) = \frac{I_{scal}^c(x,y) - I_{nlc}(x,y)}{1 - I_{nlc}(x,y)}$$

6. *Inversion*: To obtain the final enhanced image, this stage inverts the output of the restoration stage, and scales to the range [0,255].

$$I_{enhanced}^c(x,y) = 255 \times (1 - I_{restore}^c)$$

LLE Algorithm Simplification

The scaling, non-linear correction, and restoration steps involve a divide operation which is not efficient to implement in hardware. To reduce the computation involved, the equations in the algorithm are simplified by substituting the result of one stage into the next stage. This substitution results in a single constant multiplication factor rather than several divides.

Dark channel estimation without scaling and inversion is given by

$$I_{air}(x,y) = \frac{z}{255} I'_{air}(x,y) \; where \; I'_{air}(x,y) = 255 - \min_{c \, \epsilon \, [r,g,b]} I^c(x,y)$$

The result of the iterative refinement operation on I_{air} is

$$I_{refined}(x,y) = \frac{z}{255} I'_{refined(5)}(x,y)$$

where

$$I'_{refined(n+1)}(x,y) = I'_{refined(n)}(x,y) * h, \ n = [0,1,2,3,4] \ \& \ I'_{refined(0)}(x,y) = I'_{air}(x,y)$$

Substituting $I_{refined}$ into the non-linear correction equation gives

$$I_{nlc}(x,y) = \frac{z^4[I'_{refined}(x,y)]^4}{z^4[I'_{refined}(x,y)]^4 + (255 \times m)^4}$$

Substituting I_{nlc} into the restoration equation gives

$$I^c_{restore}(x,y) = 1 - \frac{I^c(x,y)}{255} - \frac{I^c(x,y)}{255}\frac{z^4}{(255 \times m)^4}[I'_{refined}(x,y)]^4$$

Subtracting $I^c_{restore}$ from 1 and multiplying by 255 gives

$$I^c_{enhanced}(x,y) = I^c(x,y) \times \left(1 + \left[\frac{z}{255 \times m}I'_{refined}(x,y)\right]^4\right)$$

With the intensity midpoint, m, set to 0.6 and the haze factor, z, set to 0.9, the simplified equation is

$$I^c_{enhanced}(x,y) = I^c(x,y) \times \left(1 + \left[\frac{1}{170}I'_{refined}(x,y)\right]^4\right)$$

In the equation above, the factor multiplied with $I^c(x,y)$ can be called the Enhancement Factor. The constant $\frac{1}{170}$ can be implemented as a constant multiplication rather than a divide. Therefore, the HDL implementation of this equation does not require a division block.

HDL Implementation

The simplified equation is implemented for HDL code generation by converting to a streaming video interface and using fixed-point data types. The serial interface mimics a real video system and is efficient for hardware designs because less memory is required to store pixel data for computation. The serial interface also allows the design to operate independently of image size and format, and makes it more resilient to video timing errors. Fixed-point data types use fewer resources and give better performance on FPGA than floating-point types.

```
open_system('LLEExample');
```
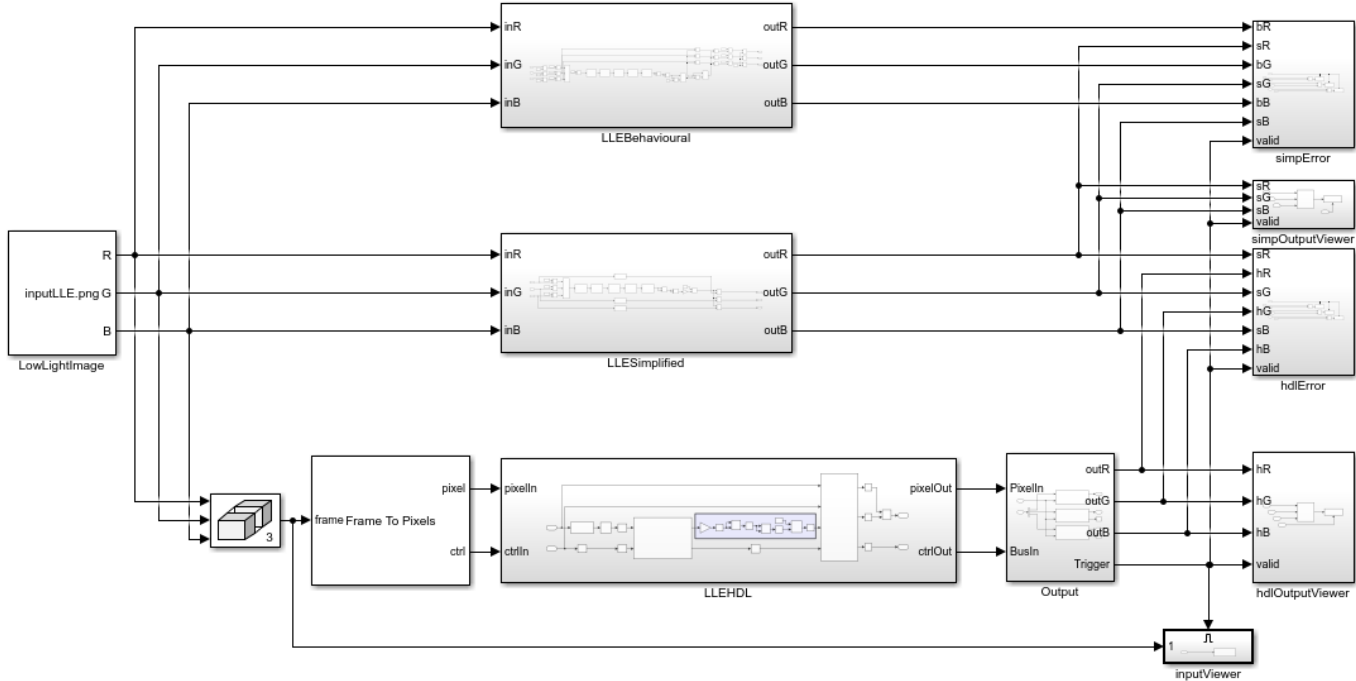
The location of the input image is specified in the *LowLightImage* block. The *LLEBehavioural* subsystem computes the enhanced image using the raw equations as described in the LLE Algorithm section. The *LLESimplified* subsystem computes the enhanced image using the simplified equations. The *simpOutputViewer* shows the output of the *LLESimplified* subsystem.

The *LLEHDL* subsystem implements the simplified equation using a steaming pixel format and fixed-point blocks from Vision HDL Toolbox. The *Input* subsystem converts the input frames to a pixel stream of `uint8` values and a `pixelcontrol` bus using the Frame To Pixel block. The *Output* subsystem converts the output pixel stream back to image frames for each channel using the Pixel To Frame block. The resulting frames are compared with result of the *LLESimplified* subsystem. The *hdlOutputViewer* subsystem and *inputViewer* subsystem show the enhanced output image and the low-light input image, respectively.

```
open_system('LLEExample/LLEHDL');
```
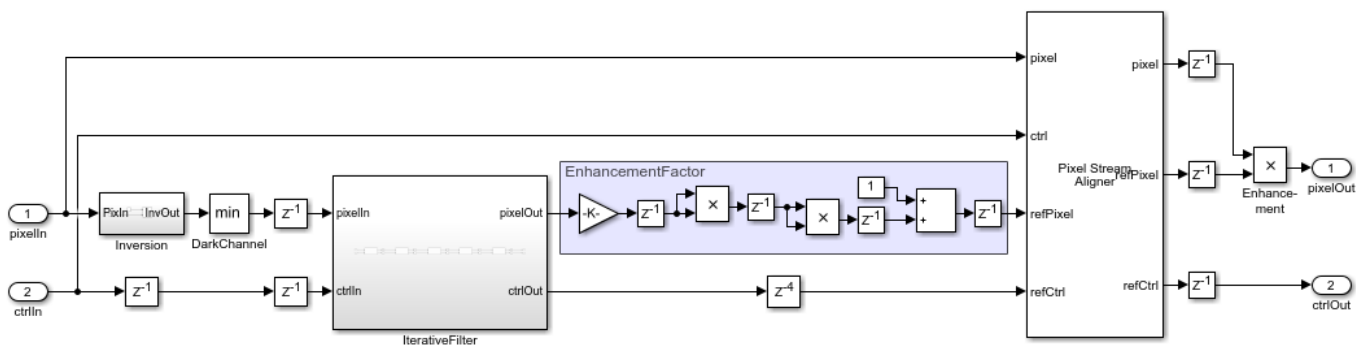

The *LLEHDL* subsystem inverts the input `uint8` pixel stream by subtracting each pixel from 255. Then the *DarkChannel* subsystem calculates the dark channel intensity minimum across all three

channels. The *IterativeFilter* subsystem smooths the airlight image using sequential Image Filter blocks. The bit growth of each filter stage is maintained to preserve the precision. The Enhancement Factor is calculated in EnhancementFactor area. The constant $\frac{1}{170}$ is implemented using a Gain block. The Pixel Stream Aligner block aligns the input pixel stream with the pipelined, modified stream. The aligned input stream is then multiplied by the modified pixel stream.

Simulation and Results

The input to the model is provided in the *LowLightImage* (Image From File) block. This example uses a 720-by-576 pixel input image with RGB channels. Both the input pixels and the enhanced output pixels use `uint8` data type. The necessary variables for the example are initialized in InitFcn callback.

The *LLEBehavioural* subsystem uses floating-point Simulink blocks to prototype the equations mentioned in the LLE Algorithm section. The *LLESimplified* subsystem implements the simplified equation in floating-point blocks, with no divide operation. The *LLEHDL* subsystem implements the simplified equation using fixed-point blocks and streaming video interface. The figure shows the input image and the enhanced output images obtained from the *LLESimplified* subsystem and the *LLEHDL* subsystem.
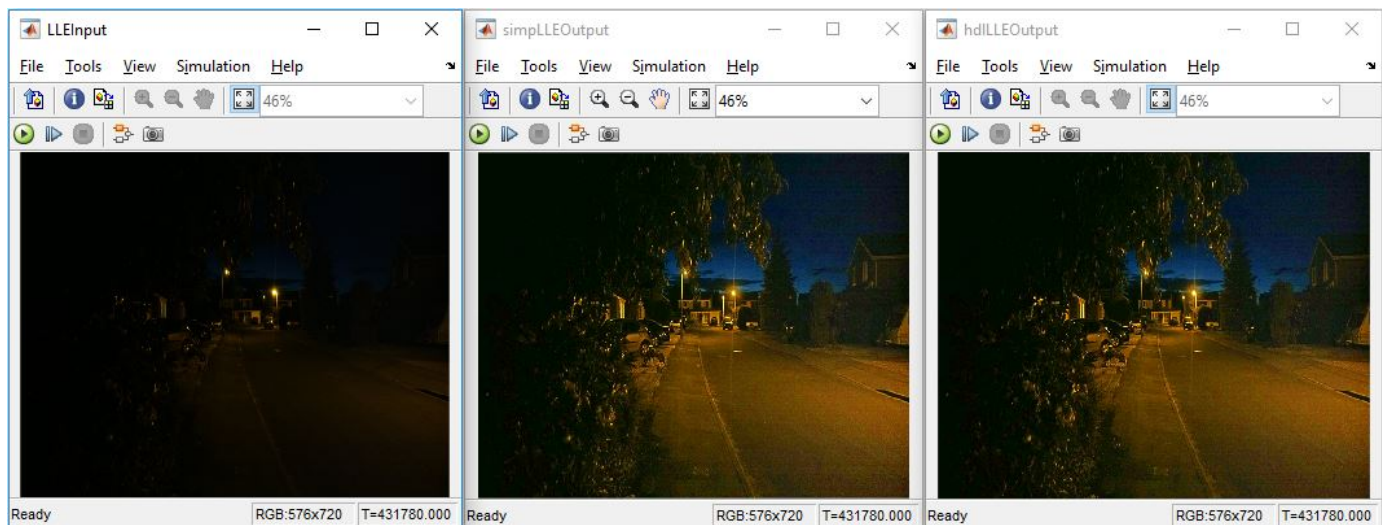
The accuracy of the result can be calculated using the percentage of error pixels. To compute the percentage of error pixels in the output image, the difference between the pixel value of the reference output image and the *LLEHDL* output image should not be greater than one, for each channel. The percent of pixel values that differ by more than 1 is computed for the three channels. The *simpError* subsystem compares the result of the *LLEBehavioural* subsystem with the result of the *LLESimplified* subsystem. The *hdlError* subsystem compares the result of the *LLEHDL* subsystem with the result of the *LLESimplified* subsystem. The error pixel count is displayed for each channel. The table shows the percentage of error pixels calculated by both comparisons.

	% error pixels		
Model Name / Channel	R	G	B
LLEBehavioural vs LLESimplified	0	0	0
LLESimplified vs LLEHDL	0.1649	0.1671	0.0477

You can generate HDL code for the LLEHDL subsystem. An HDL Coder™ license is required to generate HDL code. This design was synthesized for the Intel® Arria® 10 GX (115S2F45I1SG) FPGA. The table shows the resource utilization. The HDL design achieves a clock rate of over 250 MHz.

Model Name	LLEHDL
Input Image Resolution	720 x 576
ALM Utilization	2464
Total Registers	6643
Total Block Memory Bits	529,328
Total RAM Blocks	56
Total DSP Blocks	11

References

[1] X. Dong, G. Wang, Y. Pang, W. Li, and J. Wen, "Fast efficient algorithm for enhancement of low lighting video" IEEE International Conference on Multimedia and Expo, 2011.

FPGA Implementation of Contrast Limited Adaptive Histogram Equalization

This example shows how to implement a contrast-limited adaptive histogram equalization (CLAHE) algorithm using Simulink blocks. It uses the `adapthisteq` function from the Image Processing Toolbox™ as reference. The example model is FPGA-hardware compatible, and can provide real-time performance.

Introduction

Adaptive histogram equalization (AHE) is an image pre-processing technique used to improve contrast in images. It computes several histograms, each corresponding to a distinct section of the image, and uses them to redistribute the luminance values of the image. It is therefore suitable for improving the local contrast and enhancing the definitions of edges in each region of an image. However, AHE has a tendency to overamplify noise in relatively homogeneous regions of an image. A variant of adaptive histogram equalization called contrast-limited adaptive histogram equalization (CLAHE) prevents this effect by limiting the amplification.

CLAHE Algorithm
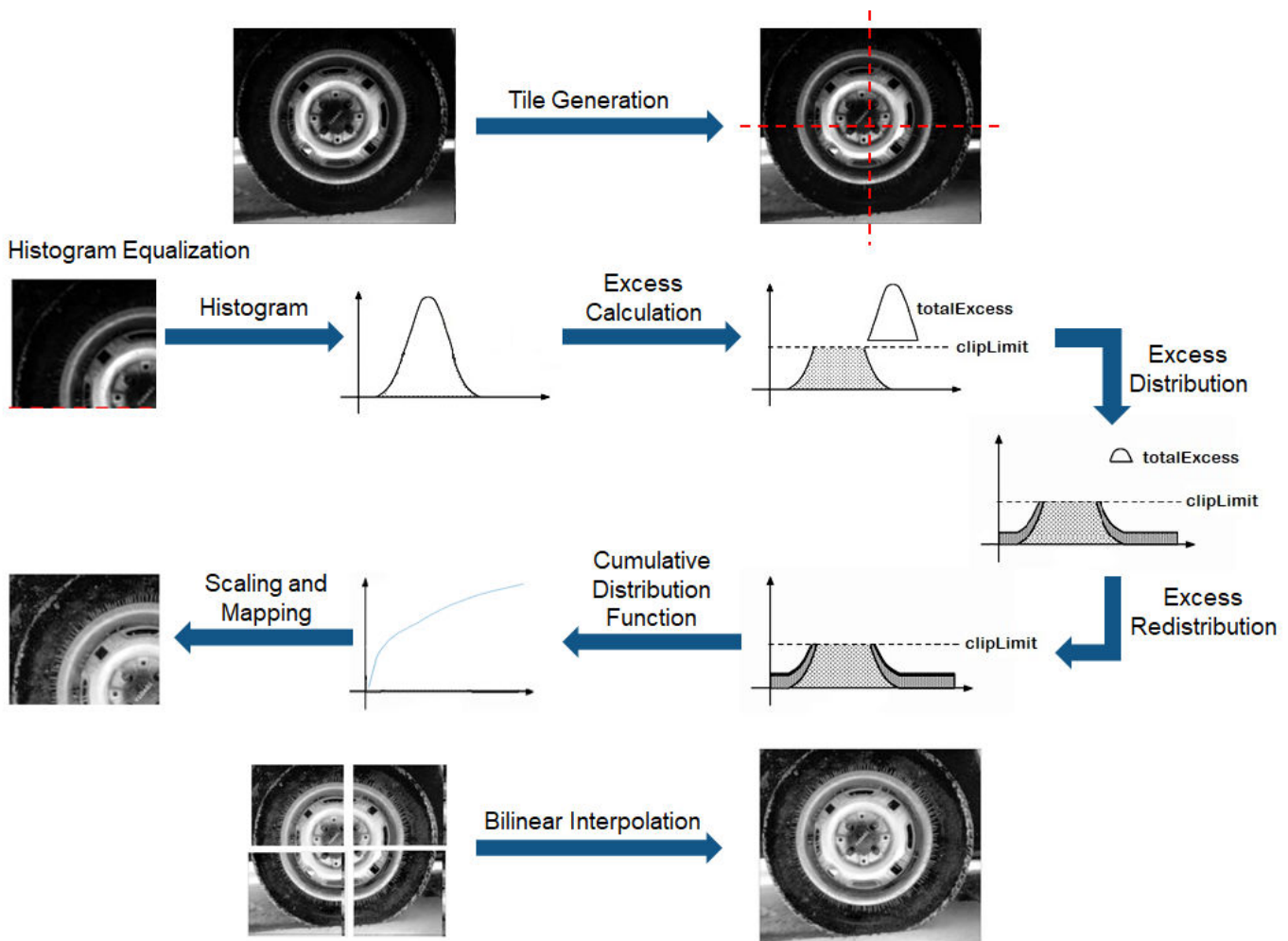
The CLAHE algorithm has three major parts: tile generation, histogram equalization, and bilinear interpolation. The input image is first divided into sections. Each section is called a tile. The input image shown in the figure is divided into four tiles. Histogram equalization is then performed on each tile using a pre-defined clip limit. Histogram equalization consists of five steps: histogram computation, excess calculation, excess distribution, excess redistribution, and scaling and mapping using a cumulative distribution function (CDF). The histogram is computed as a set of bins for each tile. Histogram bin values higher than the clip limit are accumulated and distributed into other bins. CDF is then calculated for the histogram values. CDF values of each tile are scaled and mapped using the input image pixel values. The resulting tiles are stitched together using bilinear interpolation, to generate an output image with improved contrast.

HDL Implementation
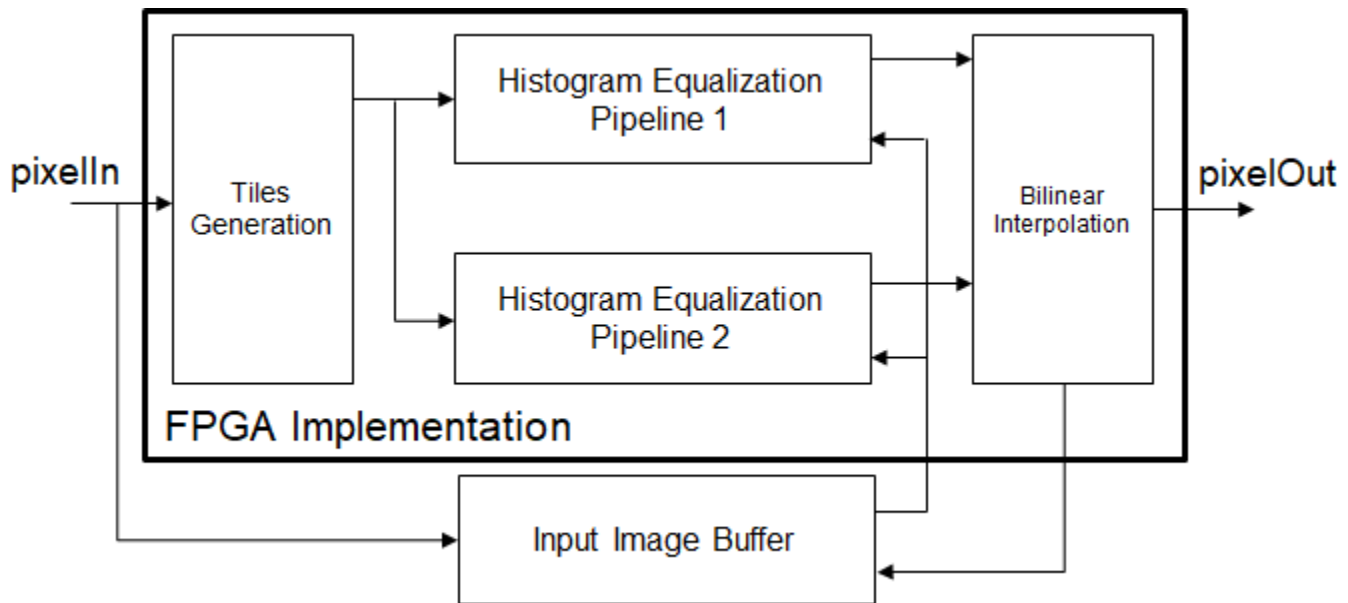
This figure shows the block diagram of the HDL implementation of the CLAHE algorithm. It consists of a tile generation block, a histogram equalization pipeline block, a bilinear interpolation block, and an input image buffer block. Tiles are generated by modifying the `pixelcontrol` bus of the pixel stream for the desired tile size. The pixel stream and the modified `pixelcontrol` bus are fed to the histogram equalization pipeline. Two histogram equalization pipelines are required to keep pace with the input data. They operate in ping-pong manner. Each pipeline contains histogram equalization modules equal to the number of tiles in the horizontal direction. The histogram equalization modules work in parallel to compute histogram equalization for each tile. The last stage in the histogram equalization module, scaling and mapping, needs the original input image data. This data is stored in an input image buffer block. The bilinear interpolation block generates addresses to read the input image pixel values from the memory. The input image pixel values from the image buffer block are given to the histogram equalization modules for mapping. Mapped values obtained from histogram equalization are scaled and used in the bilinear interpolation computation to reduce boundary artifacts.

```
modelname = 'CLAHEExample';
open_system(modelname,'force');
set_param(modelname,'SampleTimeColors','off');
set_param(modelname,'Open','on');
```

```
set_param(modelname,'SimulationCommand','Update');
set(allchild(0),'Visible','off');
```
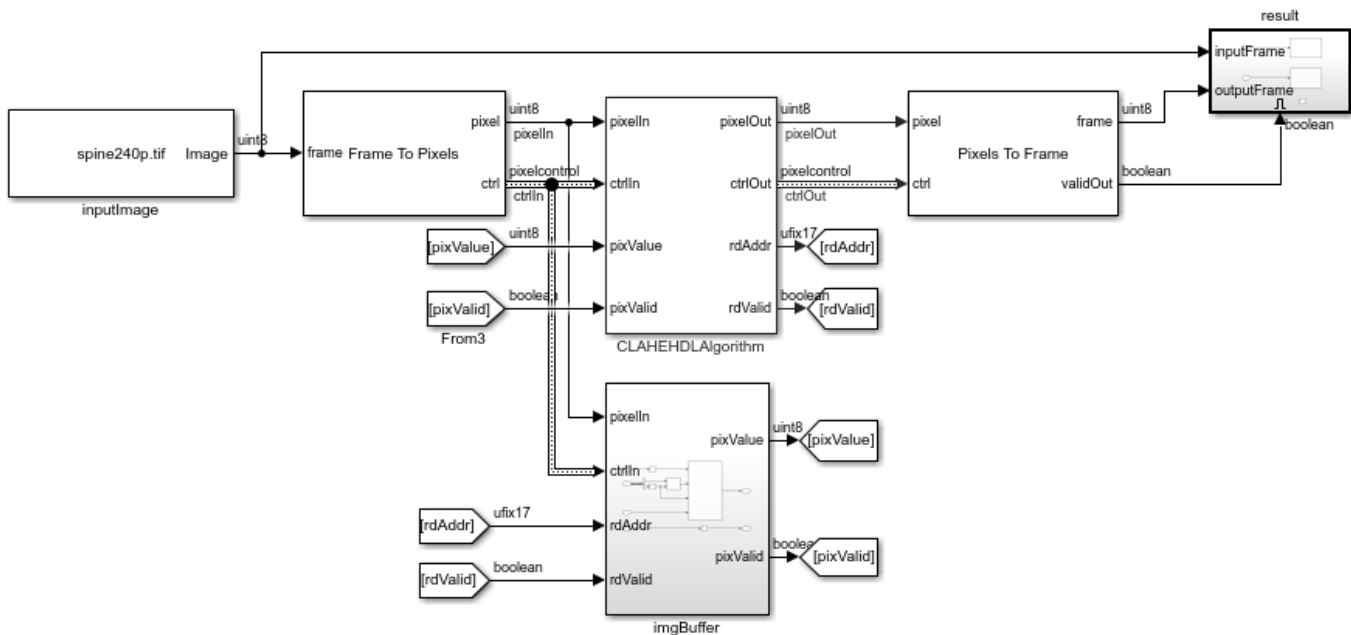

The figure shows the top level view of the `CLAHEExample` model. The input image path is specified in the inputImage block. The input image frame is converted to a pixel stream and `pixelcontrol` bus using a Frame To Pixels block. The pixel stream is passed to the CLAHEHDLAlgorithm subsystem for contrast enhancement and is also stored in the imgBuffer subsystem. While processing, the CLAHEHDLAlgorithm subsystem generates the address to read image data from the imgBuffer subsystem. The pixel value read from the imgBuffer subsystem is passed to CLAHEHDLAlgorithm for adjustment. The adjusted pixel values are given to the Pixels To Frame block and converted to a frame using the control signals. The Result subsystem shows the input image and output image once all the pixels in the frame have been received by the Pixels To Frame block.

Tile Generation

```
system = 'CLAHEExample/CLAHEHDLAlgorithm/tileGeneration';
open_system(system,'force');
```
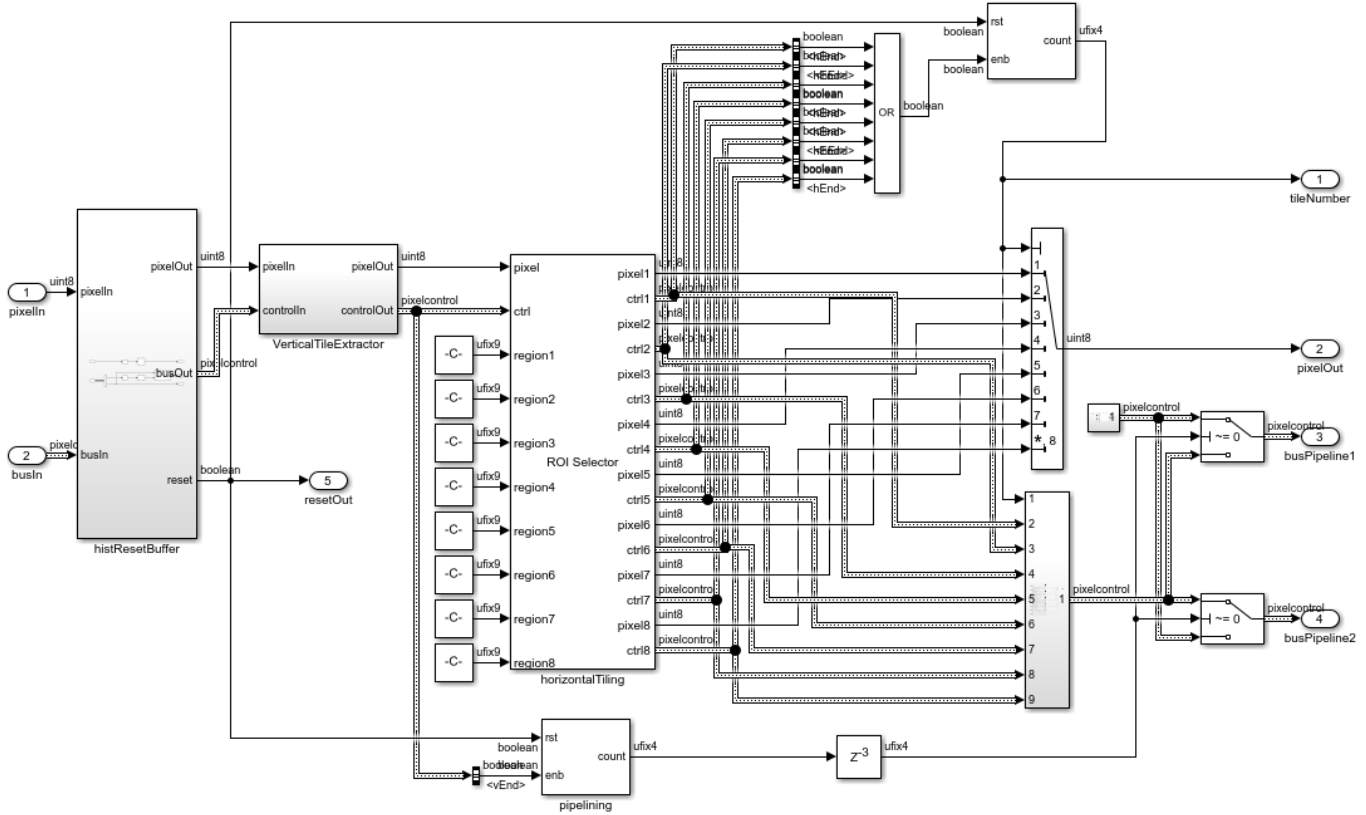
The figure shows the tile generation subsystem. The input image is divided into 8 tiles in both horizontal and vertical directions. Tiles are created by modifying the input `pixelcontrol` bus to select the pixels in each tile region. The VerticalTileExtractor subsystem extracts tiles in the vertical direction. The size of a vertical tile is computed by dividing the number of rows in the input image by the number of tiles in the vertical direction (8 in this example). This vertical tile is given to the ROI Selector block to generate 8 horizontal tiles and their corresponding `pixelcontrol` busses. The size of the horizontal tiles is computed by dividing the number of columns by the number of tiles in the horizontal direction (8 in this example). The pixel stream to the histogram equalization pipeline is controlled by diverting each vertical tile to an alternate pipe. The tile size calculated in either must be an even integer. If the input image does not divide into an integer number of even-sized tiles, pad the input image symmetrically.

Histogram Equalization Pipeline

```
system = 'CLAHEExample/CLAHEHDLAlgorithm/histoEqPipeline/';
subsystem = [system 'histPipe1'];
open_system(subsystem,'force');
```
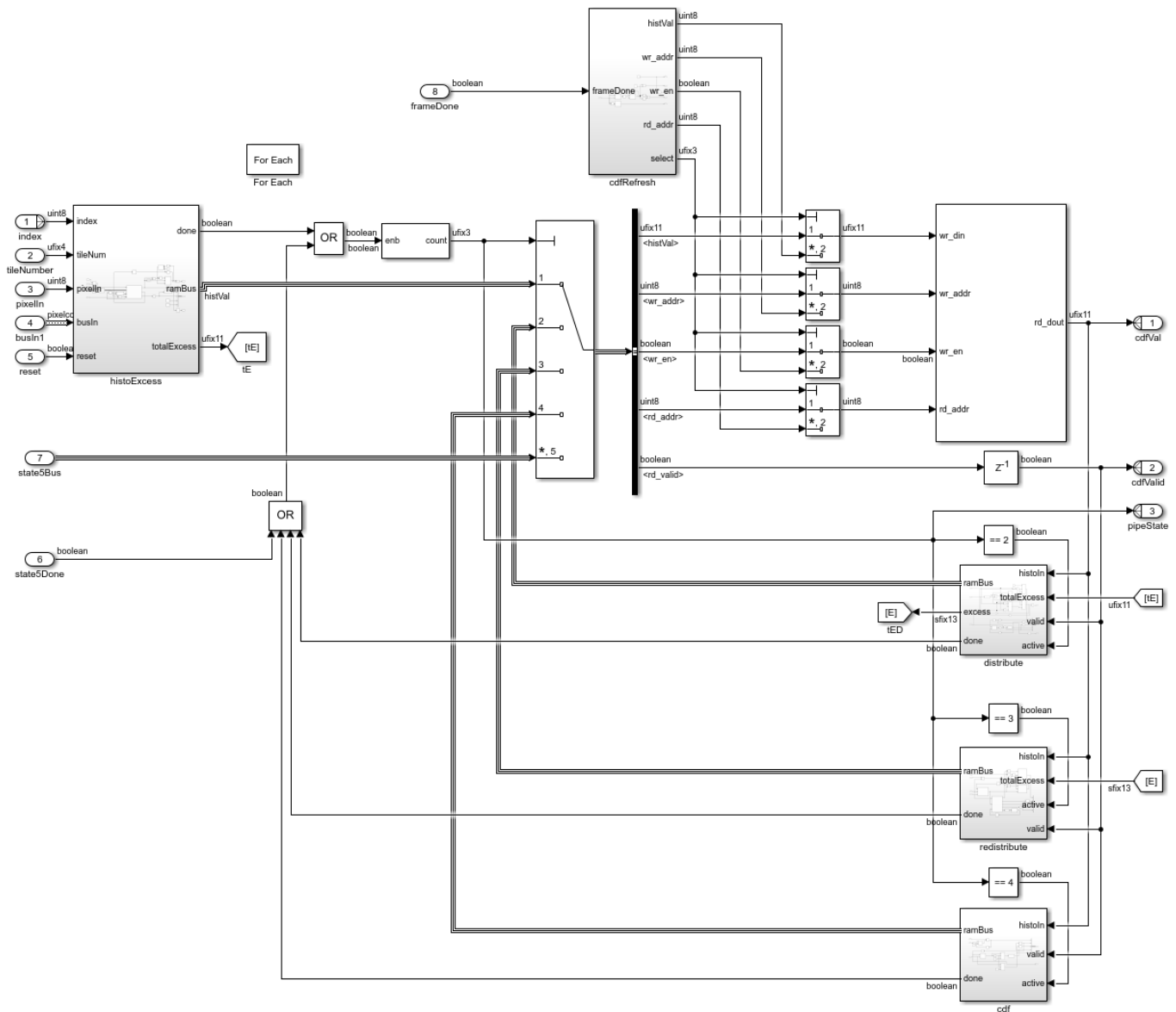
Two histogram equalization pipelines are used to keep pace with the streaming input pixels. Each histogram equalization pipeline consists of 8 histogram equalization modules corresponding to each tile in the horizontal direction. These modules are implemented by using a For Each subsystem. Each histogram equalization module is divided into five stages: histogram calculation, total excess calculation, total excess distribution, excess redistribution, cumulative distribution function, and mapping.

The first module of the histogram pipeline, histoExcess subsystem, performs histogram calculation and total excess calculation for each tile. To compute the histogram, the Histogram block is used. When the histogram is complete the block generates a **readRdy** signal. The subsystem then reads the histogram values and determines excess value from each bin by using clip limit value. The clip limit is computed from the normalized clip limit value specified using these equations.

$$minClipLimit = ceil(numPixInTile/numBins);$$

$$clipLimit = minClipLimit + round(normClipLimit * (numPixInTile - minClipLimit));$$

The excess value from each bin is accumulated to form total excess value. The previously computed histogram values are not changed during total excess calculation and are stored in a Simple Dual Port RAM memory block. The necessary control signals for the RAM block (ramBus) are generated by the histoExcess subsystem. The total excess value calculated in the histoExcess subsystem is used by the Distribute subsystem.

The Distribute subsystem computes two variables: average bin increment and upper limit. These values are computed from the total excess value by using these equations:

$$avgBinIncr = totalExcess/numBins;$$

$$upperLimit = clipLimit - avgBinIncr;$$

The Distribute subsystem then reads the value of each histogram bin from the RAM block. It updates the value at every bin based on these three conditions:

1   If the histogram value of a bin is greater than the clip limit, it is replaced with the clip limit.
2   If the histogram value of a bin is between the clip limit and the upper limit, the histogram value is replaced with the clip limit. The total excess value is reduced by the number of added pixels equal to (`clipLimit - histVal`).
3   If the histogram value of a bin is less than the upper limit, the histogram value is increased by the average bin increment. The total excess value is reduced by the average bin increment.

The adjusted histogram value is stored at the same address. The remaining total excess value is passed to the Redistribute subsystem as excess value.

```
system = 'CLAHEExample/CLAHEHDLAlgorithm/histoEqPipeline/';
subsystem = [system 'histPipe1/redistribute'];
open_system(subsystem,'force');
```
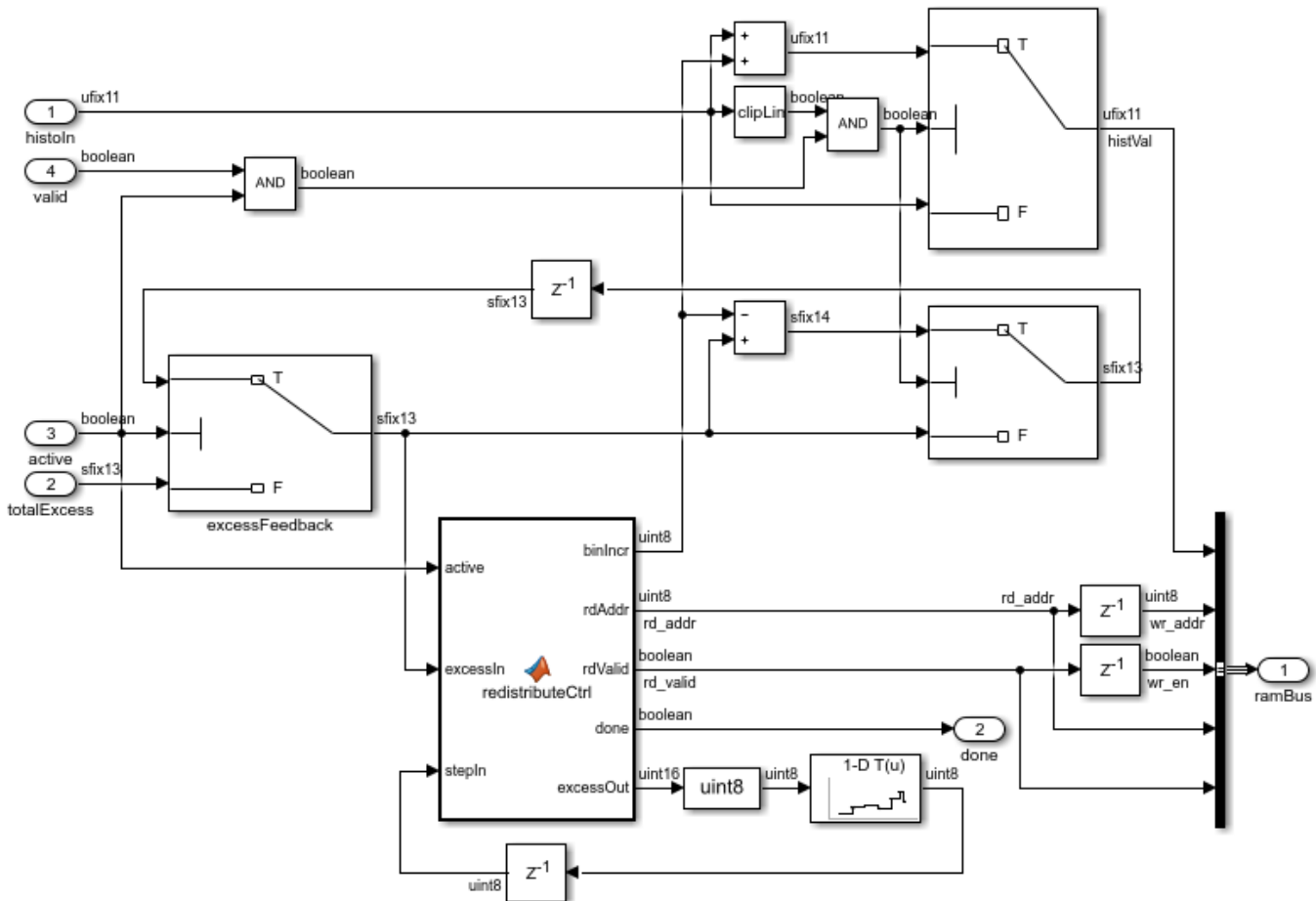
The Redistribute subsystem distributes spillover excess values to the histogram bins. It primarily uses two variables to distribute excess values: **binIncr** and **step**. **binIncr** specifies the value to be added to the histogram bins. **step** specifies the increment in the address counter used to fetch the histogram bin value. If the excess is greater than or equal to the number of bins, then **binIncr** is calculated by dividing the excess value by the number of bins, and **step** is set to 1. The divide is implemented by using a right-shift operation, since the number of bins is a power of 2.

If the excess is less than the number of bins, **binIncr** is set to 1 and **step** is calculated by dividing the number of bins by the excess value. The divide is computed by using a n-D Lookup Table block. The `redistributeCtrl` MATLAB Function generates the address for the RAM block by using the **step** value computed. When the address reaches the total number of bins, the **step** value is re-computed using the most recent excess value. Care is taken to not repeat the first bin as the start bin for redistribution. If the value of the histogram bin is less than the clip limit, it is increased by **binIncr**, and the same value is subtracted from the excess value. If the value of histogram bin is equal to the clip limit, no operation is performed and the value is written back to the same address. The MATLAB Function block repeats these bin adjustments until the excess value reaches 0.

The last stage of the histogram equalization pipeline is CDF calculation. The CDF subsystem computes the cumulative sum of the histogram bin values. The histogram values are read from the RAM block and added to the sum of the previous histogram bin values. It is then stored to the same address.

The five stages of the histogram equalization module can be considered as five states. The five states of histogram equalization module are sequential. Thus, a state counter is used to move from one state to another state. A counter value determines the state of the histogram equalization module. A Multiport Switch block is used with the state counter as the index value. The multi-port switch connects the ramBus from each state with the correct memory according to the index. The state counter is in state 1 in idle condition. When histoExcess finishes excess calculation it sets the **done** signal to 1 for one cycle, and the state counter moves to state 2. Similarly, the distribute subsystem, redistribute subsystem, and cdf subsystem generate done flags when their processing completes. These done flags increment the state counter to state 5, where it uses input image pixel values from the input image buffer block as addresses to read CDF values from the RAM. The address counter that reads the input image values is driven by the bilinear interpolation subsystem. The state counter is incremented by the bilinear interpolation subsystem when mapping for the respective pipeline is complete.

Bilinear Interpolation
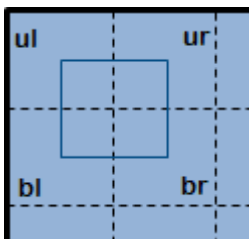
Bilinear interpolation is used to smooth edges when the tiles are stitched together. The figure shows how four tiles are used to compute pixel values in the output image. The each tile is divided into four parts. One part from each of the four tiles are grouped together to compute bilinear interpolation for that section of the image.

Interpolation uses this equation:

```
claheI(imgTileIdx{1}, imgTileIdx{2}) = ...
    (rowRevW .* (colRevW .* double(grayxform(imgPixVals,ulMapTile)) + ...
                 colW    .* double(grayxform(imgPixVals,urMapTile)))+ ...
     rowW    .* (colRevW .* double(grayxform(imgPixVals,blMapTile)) + ...
                 colW    .* double(grayxform(imgPixVals,brMapTile))))...
    /normFactor;
```

$$grayxform(imgPixVals, mapTile) = round(255 * mapTile(imgPixVals)/numPixInTile);$$

The bilinear interpolation equation uses the position of a pixel with respect to each tile and the intensity information at that position to compute a pixel value in the output image. The intensity information is obtained from the input image pixel values stored in the image buffer. For corner tiles, intensity values are replicated (mirrored). The intensity information at the respective position in each tile is extracted from the CDF function of the histogram equalization pipeline by using the input image pixel value at the same position. The `grayxform` function scales the values obtained from the CDF function. The result is then divided by the number of pixels in a tile, represented as *normFactor* in the equation.

```
system = 'CLAHEExample/CLAHEHDLAlgorithm/bilinearInterpolation';
open_system(system,'force');
```
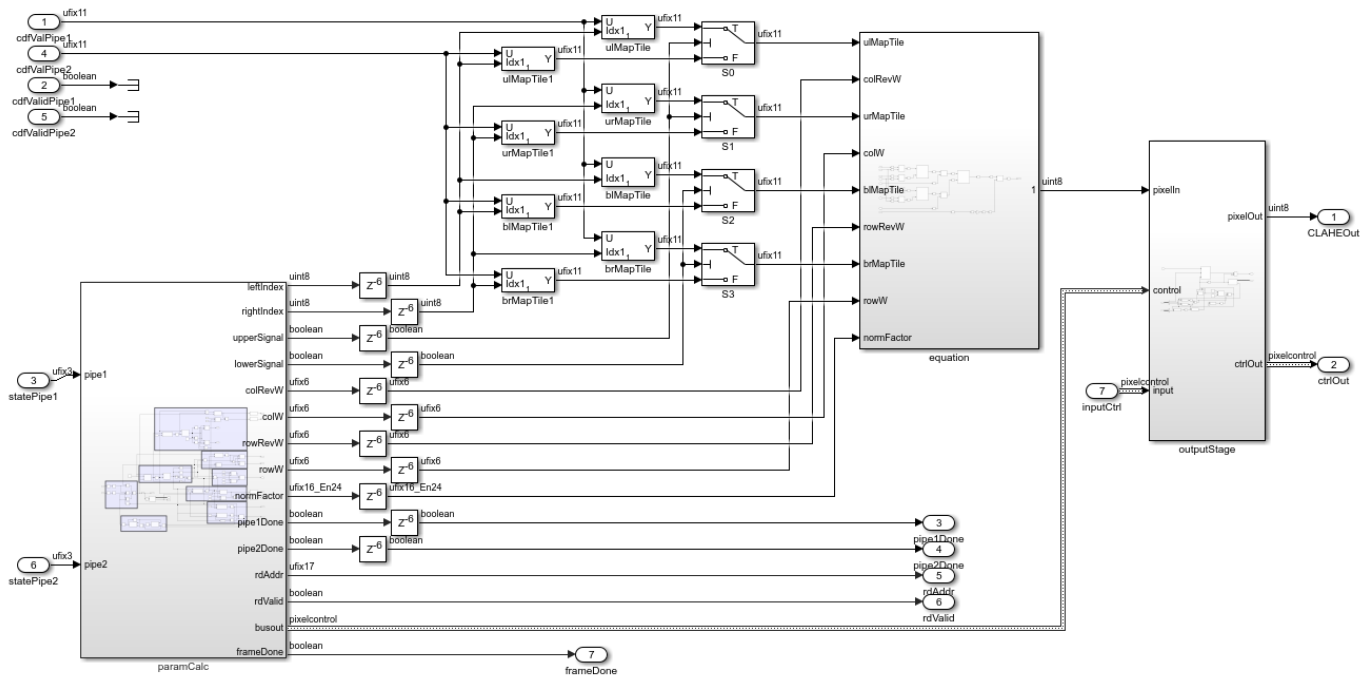
The figure shows the HDL implementation of the bilinear interpolation subsystem. When the histogram equalization pipeline reaches state 5, the paramCalc subsystem starts computing the read address for the imgBuffer subsystem. The pixel value read from the buffered image is the address for the RAM in the histogram equalization pipeline. CDF values are fetched from the read address for all the tiles from both the histogram equalization pipelines simultaneously. The required CDF values are selected and passed to the equation subsystem using Selector Switch blocks and Switch blocks. The Switch block selects which pipeline contains upper/lower tiles and the Selector Switch blocks select data corresponding to left/right tiles. The control signals for the Selector Switch and Switch blocks are generated in the paramCalc subsystem by using a read counter. Thus, intensity values at a pixel position for each tile are obtained from the image buffer. The bilinear interpolation equation also requires the pixel position and the total number of pixels in the tile. These parameters are also generated in the paramCalc subsystem. The equation subsystem is pipelined to optimize performance in hardware. The result is returned as a pixel stream with a `pixelcontrol` bus.
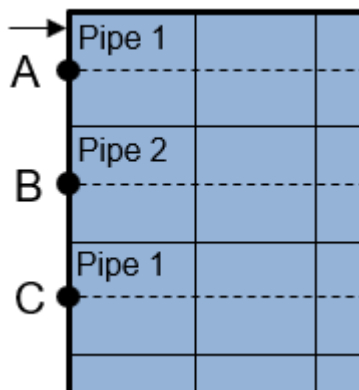
Bilinear interpolation of the output image is computed by traversing the rows from left to right. When all histogram equalization modules in the first pipeline have reached state 5, the paramCalc subsystem is enabled. The read addresses for the imgBuffer subsystem are computed until point A. Further computation of bilinear interpolation requires values from the histogram equalization modules of the second pipeline. When all histogram equalization modules in the second pipeline have reached state 5, the read address counter is again enabled and the bilinear interpolation output results are computed for pixel positions between point A and point B. Once the address counter reaches point B, results from first pipeline are no longer required. The `pipe1Done` signal is generated to change the state of the first histogram equalization pipeline modules back to state 1. Until this point, the tiles in the first pipeline are upper tiles and the tiles in the second pipeline are lower tiles. For the computation of values between point B and point C, the tiles in the second pipeline become the upper tiles and tiles in the first pipeline are now lower tiles. This operation continues until only the lowest tiles in the image remain. The output for these tiles is computed by replicating the values for the other pipeline. The output results are pushed into a FIFO in the outputStage subsystem and popped out such that the output valid signal is similar to that of the input pixel stream.

Model Parameters
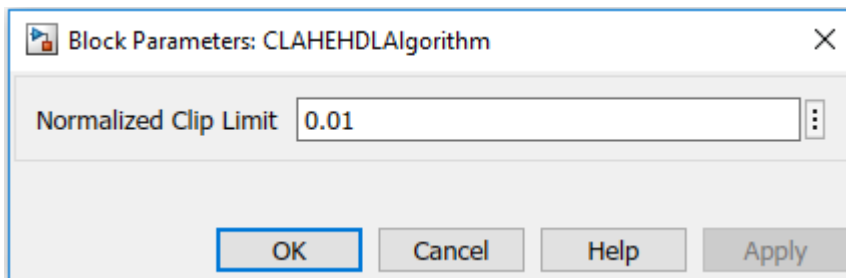
CLAHE uses a clip limit to prevent over-saturation of the image in homogeneous areas. These areas are characterized by a high peak in the histogram of an image tile due to many pixels falling in the same intensity range. For the model presented here, the clip limit is a user-defined normalized value. The default value is 0.01 (as shown in figure). The clip limit can be any value between 0 and 1 (inclusive).

Simulation and Results

This example uses an input image of size 240-by-320 pixels, whose path is specified in the inputImage block. The input image pixels are specified by a single intensity component in `uint8` data type. For 8 tiles in each direction, the computed tile size is 30-by-40 and the number of pixels in each tile is 1200. The number of histogram bins is set to 256.
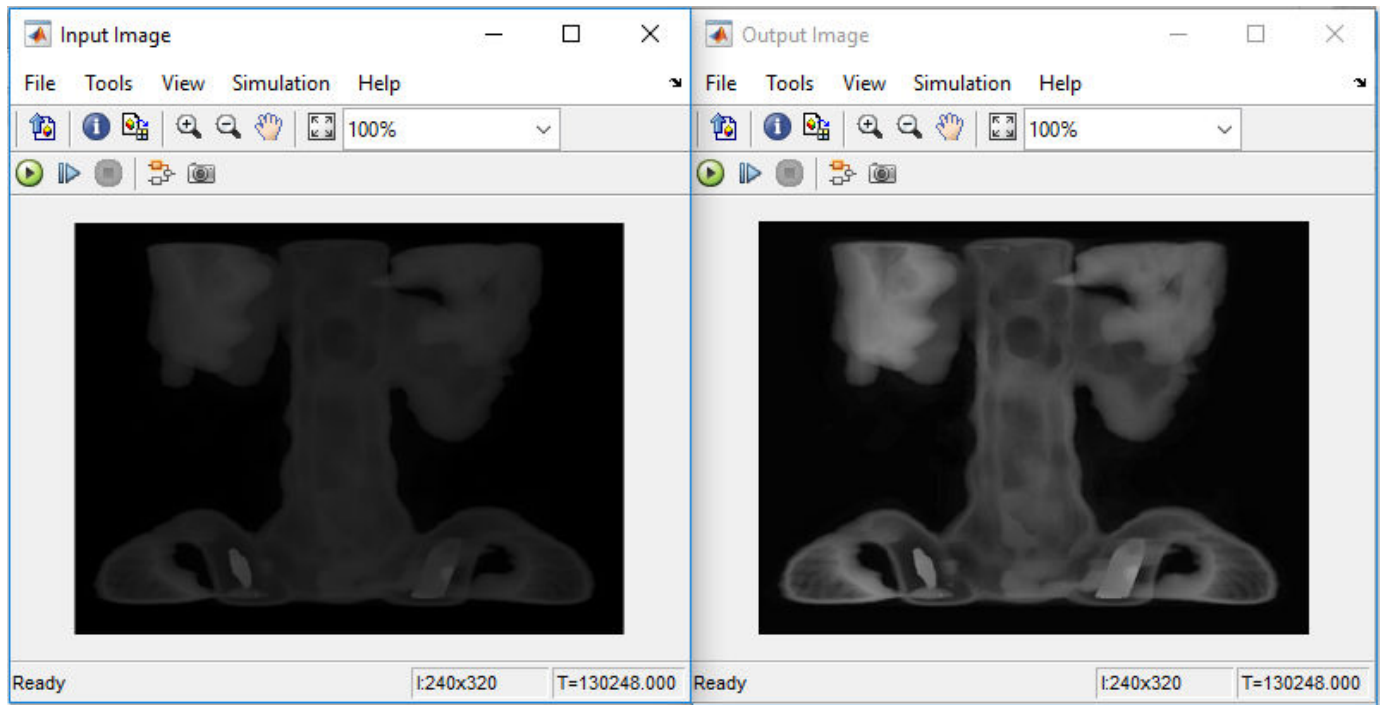
This figure shows the input image and output image from the CLAHE model. The result shows the improved contrast in the output image, without over- saturation. The result of the CLAHE HDL model matches the `adaphisteq` function in MATLAB and has error of ± 1 pixel.

HDL code can be generated for the CLAHEHDL subsystem. An HDL Coder™ license is required to generate HDL code. This design was synthesized on the Intel® Arria® 10 GX platform, for 10AX115S2F45I1SG FPGA device. The table shows the resource utilization. The HDL design achieves a clock rate of over 120 MHz.

```
% ================================================
% |Model Name              ||      CLAHEHDL     ||
% ================================================
% |Input Image Resolution  ||     320 x 240     ||
% |ALM Utilization         ||       48527       ||
% |Total Registers         ||       52887       ||
% |Total RAM Blocks        ||        62         ||
% |Total DSP Blocks        ||        38         ||
% ================================================
```

References

Karel Zuiderveld, "Contrast Limited Adaptive Histogram Equalization", Graphics Gems IV, p. 474-485, code: p. 479-484.

Image Resize

This example shows how to downsample an image by using the bilinear, bicubic, or Lanczos-2 interpolation algorithm.

In theory, you can achieve exact image reconstruction by resizing the image using a sinc kernel. However, sinc kernels have infinite spatial extent. To limit the extent, interpolation implementations use simpler kernels to approximate a sinc. The bilinear interpolation algorithm uses the weighted sum of the nearest four pixels to determine the values of the output pixels. Bicubic and Lanczos-2 interpolations are approximations of a sinc kernel. Bicubic interpolation is a more computationally efficient version of the Lanczos-2 method.

Behavioral Reference

By default, the Image Processing Toolbox™ function `imresize` uses the bicubic interpolation algorithm. You can choose bilinear or Lanczos-2 interpolation algorithms by setting the `'Method'` name-value pair argument to `'bilinear'` or `'lanczos-2'`, respectively.

```
v = VideoReader('rhinos.avi');
I = readFrame(v);
Y = imresize(I,[160,256],'Method','bilinear');
figure;
imshow(Y);
```


Interpolation Algorithms

Bilinear interpolation determines the inserted pixel value from the weighted average of the four input pixels nearest to the inserted location. h and v are horizontal scale and vertical scale, respectively, and are calculated independently.

2-141
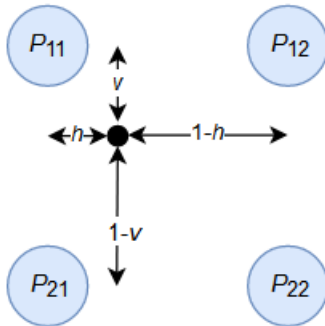
The value for each output pixel is given by $(P_{11}h + P_{12}(1 - h))v + (P_{21}h + P_{22}(1 - h))(1 - v)$.

The bicubic algorithm calculates the average of the 16 input pixels nearest to the inserted location.
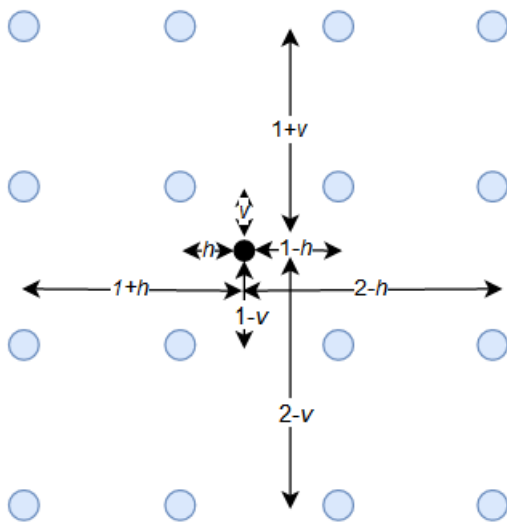
The bicubic coefficients are given by
$$\begin{cases} 1 - 2|d|^2 + |d|^3 & 0 \le |d| < 1 \\ 4 - 8|d| + 5|d|^2 - |d|^3 & 1 \le |d| < 2 \\ 0 & 2 \le |d| \end{cases}.$$

These equations show that the bilinear and bicubic algorithms calculate coefficients for each output pixel.

The Lanczos-2 algorithm precalculates the coefficients based on the resize factor. The model calls the `lanczos2_coeffi.m` script to calculate and store these coefficients. The script calculates the Lanczos-2 coefficients using 6 taps and 32 phases.

Implementation of Interpolation Algorithms for HDL

This figure shows the principle used to implement the image resize algorithm for hardware. For example, consider downsampling an image by a scale factor of 3/4. One possible implementation of

downsampling an image by 3/4 is to upsample by a factor of 3 and then downsample by a factor of 4. The figure shows the pixel indexes after these operations. Blue dots represent the original pixels, and green crosses represent the interpolated pixels after upsampling.
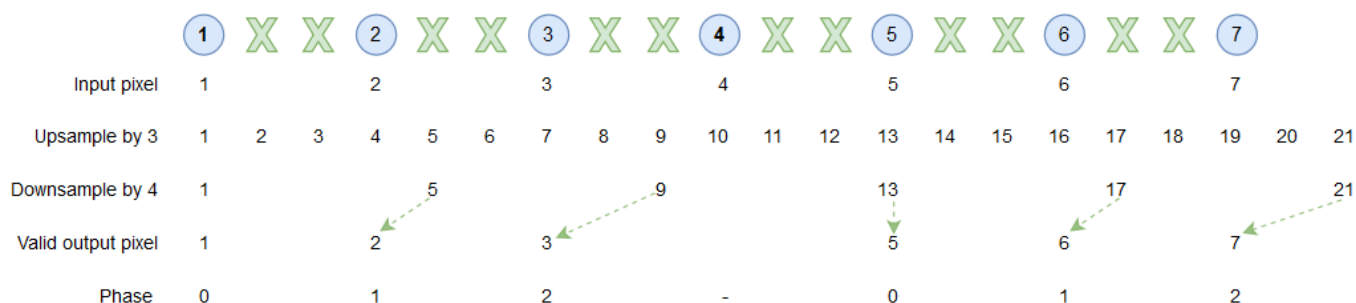
The indexes after downsampling show that not all the interpolated pixels are used in the output image. This example implements a more efficient version of the downsample step by generating interpolated pixels only when they are needed in the output image.

The phase, shown in the bottom line of the figure, is an index that selects which pixels are needed for the output image. When the phase is 0, the algorithm returns the original input pixel value. When the phase is 1, the algorithm calculates coefficients to generate the interpolated pixel in the first position. When the phase is 2, the algorithm calculates coefficients to generate the interpolated pixel in the second position.

Example Model

Similar to the `imresize` function, the imresize(downsample) subsystem in this model supports two ways to define the output image size. You can specify a scale factor ranging from 1.000 to 127.999, or you can define the output frame width and height in pixels. Double-click the imresize(downsample) subsystem to set its parameters.

To avoid aliasing introduced by lowering the sampling rate, the model includes a lowpass filter before the imresize(downsample) subsystem. After the imresize(downsample) subsystem, the Pixel Stream FIFO block consolidates the output pixels into contiguous lines of valid pixels surrounded by blanking intervals of invalid pixels. This FIFO is optional. Use the FIFO when you want to take advantage of the increased blanking interval to perform further computations on the pixel stream. The Measure Timing block displays the size of the output frames.

```
modelname = 'ImageResizeForFPGAHDL';
open_system(modelname);
set_param(modelname,'SampleTimeColors','on');
set_param(modelname,'Open','on');
set(allchild(0),'Visible','off');
```
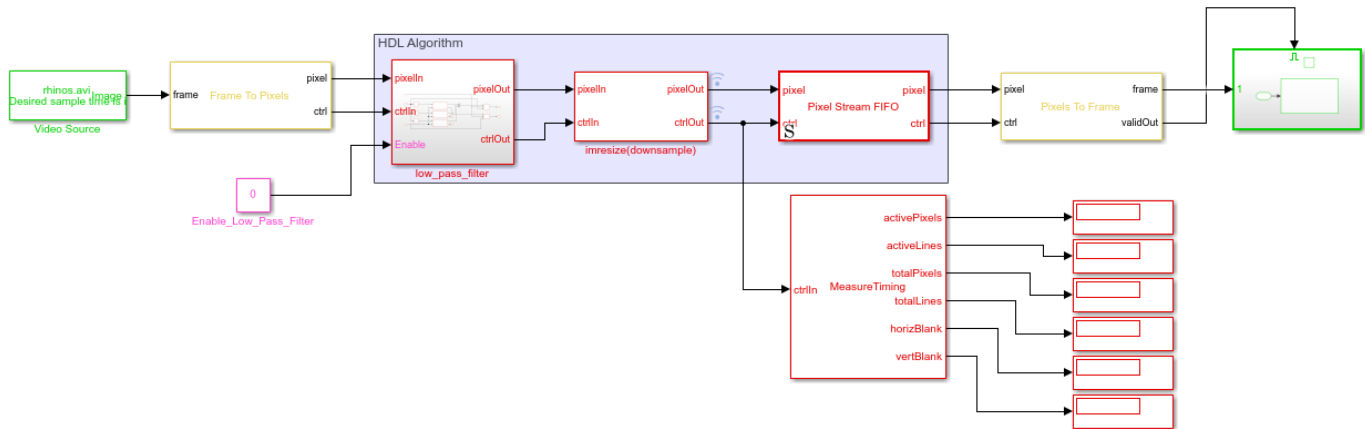
In the imresize(downsample) subsystem, the input_conversion and output_conversion subsystems convert the colorspace of the pixel stream based on the parameter on the mask. The valid_gen_horizontal and valid_gen_vertical subsystems return control signals that are used for generating coefficients and rebuilding the output control bus. If the last line of the image contains no valid pixels after downsampling, the ctrlBusRebuild subsystem rebuilds the control bus for the new size.

```
open_system([modelname '/imresize(downsample)'],'force');
```
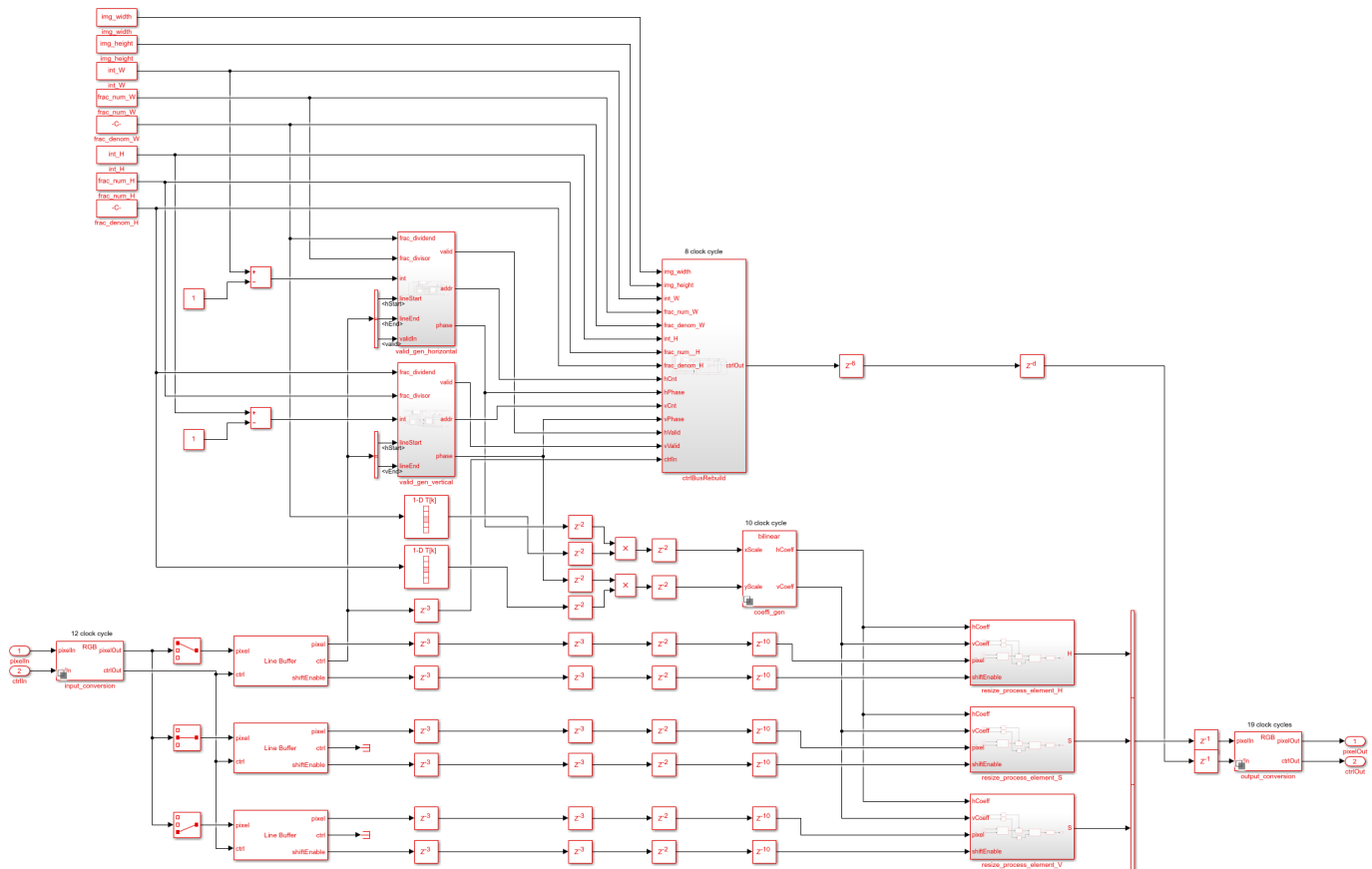
This diagram shows the expected output from the valid_gen_horizontal and valid_gen_vertical subsystems. The valid signal indicates the validity of the current address and the corresponding phase. To simplify rebuilding the control bus, the first line and row of each output frame are always valid.
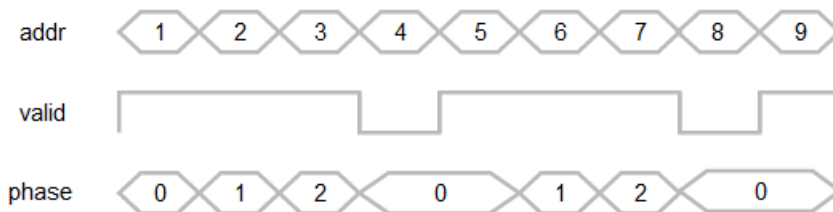
The coefficient generation subsystem, coeffi_gen, is a variant subsystem, where bilinear, bicubic, and Lanczos-2 coefficient generators are implemented separately. You can select the algorithm from the mask.

```
open_system([modelname '/imresize(downsample)/coeffi_gen'],'force');
```
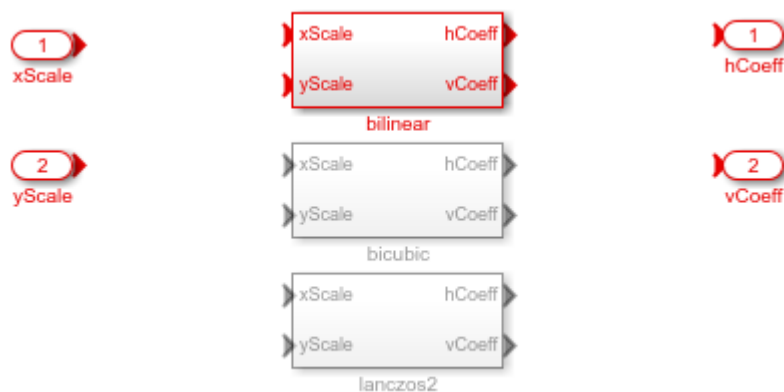

The resize_process_element subsystems multiply the coefficients with each pixel component by using a separable filter in vertical order and then in horizontal order. The trim_0_1 subsystem ensures the result is between 0 and 1.

```
open_system([modelname '/imresize(downsample)/resize_process_element_H'],'force');
```
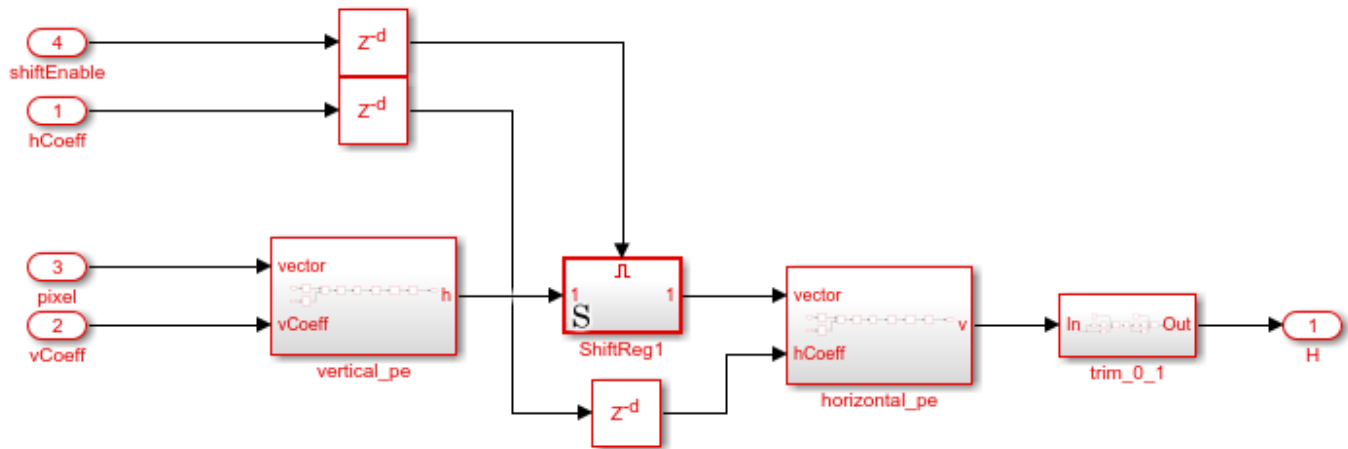
Resource Usage

These tables show the resource usage for the imresize(downsample) subsystem with 240p video input, and do not include the lowpass filter or the Pixel Stream FIFO. The design was synthesized to achieve a clock frequency of 150 MHz.

This table shows the resources for each of the three algorithms when downsampled in the HSV colorspace.

Name	Slice LUTs (218600)	Slice Registers (437200)	F7 Muxes (109300)	Slice (54650)	LUT as Logic (218600)	LUT as Memory (70400)	LUT Flip Flop Pairs (218600)	Block RAM Tile (545)	DSPs (900)
∨ N Subsystem	11436	27342	57	6849	9505	1931	6739	16.5	106
> ⊞ u_Bicubic (Bicubic_block)	3717	8930	0	2289	3068	649	2200	5.5	36
> ⊞ u_Bilinear (Bilinear_block)	2380	4933	0	1294	1930	450	1448	2.5	26
> ⊞ u_Lanczos_2 (Lanczos_2)	5339	13443	57	3373	4507	832	2955	8.5	44

This table shows the resources for each of the three algorithms when downsampled in the RGB colorspace.

Name	Slice LUTs (218600)	Slice Registers (437200)	F7 Muxes (109300)	Slice (54650)	LUT as Logic (218600)	LUT as Memory (70400)	LUT Flip Flop Pairs (218600)	Block RAM Tile (545)	DSPs (900)
∨ N Subsystem	9628	24002	24	5945	7949	1679	5666	13.5	88
> ⊞ u_Bicubic (Bicubic_block)	3194	7824	0	2007	2630	564	1832	4.5	30
> ⊞ u_Bilinear (Bilinear_block)	1729	3965	0	1059	1358	371	1138	1.5	20
> ⊞ u_Lanczos_2 (Lanczos_2)	4705	12177	24	2913	3961	744	2648	7.5	38

References

[1] Keys, R. "Cubic Convolution Interpolation for Digital Image Processing." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29, no. 6 (December 1981): 1153-60. https://doi.org/10.1109/TASSP.1981.1163711.

[2] Smith, Alvy Ray. "Color Gamut Transform Pairs." In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '78*, 12-19. Not Known: ACM Press, 1978. https://doi.org/10.1145/800248.807361.

Fog Rectification

This example shows how to remove fog from images captured under foggy conditions and implement the algorithm for FPGAs.

Fog rectification is an important preprocessing step for applications in autonomous driving and object recognition. Images captured in foggy and hazy conditions have low visibility and poor contrast. These conditions can lead to poor performance of vision algorithms performed on foggy images. Fog rectification improves the quality of the input images to such algorithms.

This example shows a streaming fixed-point implementation of the fog rectification algorithm that is suitable for deployment to hardware.

To improve the foggy input image, the algorithm performs fog removal and then contrast enhancement. The diagram shows the steps of both these operations.

This example takes a foggy RGB image as input. To perform fog removal, the algorithm estimates the dark channel of the image, calculates the airlight map based on the dark channel, and refines the airlight map by using filters. The restoration stage creates a defogged image by subtracting the refined airlight map from the input image.

Then, the Contrast Enhancement stage assesses the range of intensity values in the image and uses contrast stretching to expand the range of values and make features stand out more clearly.
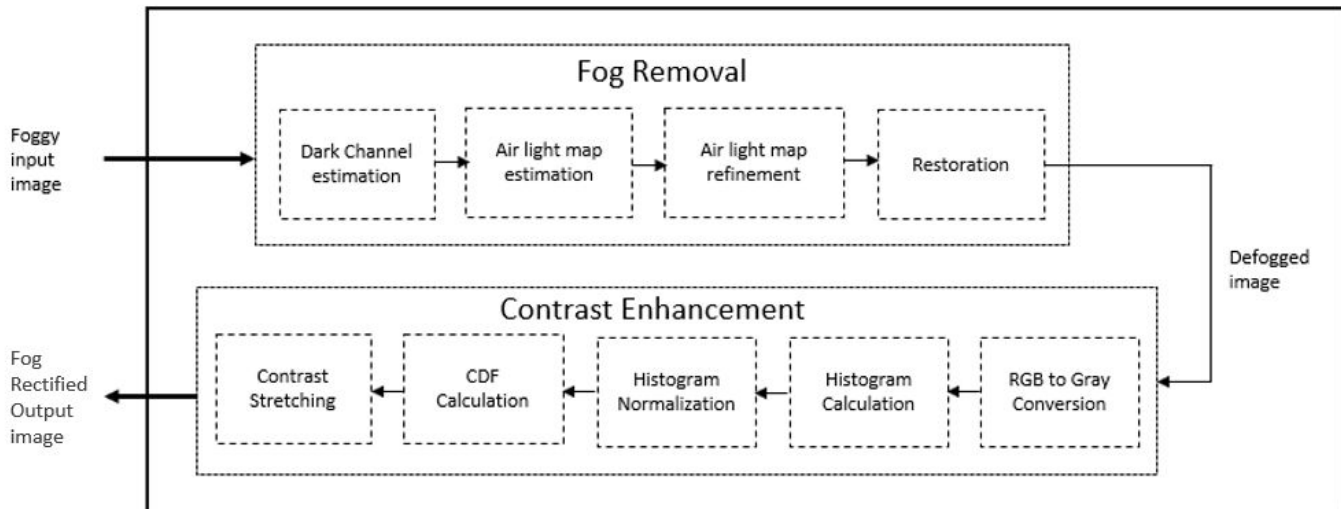
Fog Removal

There are four steps in performing fog removal.

1. **Dark Channel Estimation**: The pixels that represent the non-sky region of an image have low intensities in at least one color component. The channel formed by these low intensities is called the *dark channel*. In a normalized, fog-free image, the intensity of dark channel pixels is very low, nearly zero. In a foggy image, the intensity of dark channel pixels is high, because they are corrupted by fog. So, the fog removal algorithm uses the dark channel pixel intensities to estimate the amount of fog.

The algorithm estimates the dark channel $I_{dark}^c(x,y)$ by finding the pixel-wise minimum across all three components of the input image $I^c(x,y)$ where $c \epsilon [r,g,b]$.

2. **Airlight Map Calculation**: The whiteness effect in an image is known as *airlight*. The algorithm calculates the airlight map from the dark channel estimate by multiplying by a haze factor, z, that represents the amount of haze to be removed. The value of z is between 0 and 1. A higher value means more haze will be removed from the image.

$$I_{air}(x,y) = z \times \min_{c \epsilon [r,g,b]} I_{dark}^c(x,y)$$

3. **Airlight Map Refinement**: The algorithm smoothes the airlight image from the previous stage by using a Bilateral Filter block. This smoothing strengthens the details of the image. The refined image is referred to as $I_{refined}(x,y)$.

4. **Restoration**: To reduce over-smoothing effects, this stage corrects the filtered image using these equations. The constant, m, represents the mid-line of changing the dark regions of the airlight map from dark to bright values. The example uses an empirically derived value of $m = 0.6$.

$$I_{reduced}(x,y) = m \times min(I_{air}(x,y), I_{refined}(x,y))$$

The algorithm then subtracts the airlight map from the input foggy image and multiplies by the factor $\frac{255}{255 - I_{reduced}(x,y)}$.

$$I_{restore}(x,y) = 255 \times \frac{I^c(x,y) - I_{reduced}(x,y)}{255 - I_{reduced}(x,y)}$$

Contrast Enhancement

There are five steps in contrast enhancement.

1. **RGB to Gray Conversion**: This stage converts the defogged RGB image, $I_{restore}^c(x,y)$, from the fog removal algorithm into a grayscale image, $I_{gray}(x,y)$.

2. **Histogram Calculation**: This stage uses the Histogram block to count the number of pixels falling in each intensity level from 0 to 255.

3. **Histogram Normalization**: The algorithm normalizes the histogram values by dividing them by the input image size.

4. **CDF Calculation**: This stage computes the cumulative distribution function (CDF) of the normalized histogram bin values by adding them to the sum of the previous histogram bin values.

5. **Contrast Stretching**: Contrast stretching is an image enhancement technique that improves the contrast of an image by stretching the range of intensity values to fill the entire dynamic range. When dynamic range is increased, details in the image are more clearly visible.

5a. *i1 and i2 calculation*: This step compares the CDF values with two threshold levels. In this example, the thresholds are 0.05 and 0.95. This calculation determines which pixel intensity values align with the CDF thresholds. These values determine the intensity range for the stretching operation.

5b. *T calculation*: This step calculates the stretched pixel intensity values to meet the desired output intensity values, o_1 and o_2.

o_1 is 10% of maximum output intensity floor(10*255/100) for `uint8` input.

o_2 is 90% of maximum output intensity floor(90*255/100) for `uint8` input.

T is a 256-element vector divided into segments t_1, t_2, and t_3. The segment elements are computed from the relationship between the input intensity range and the desired output intensity range.
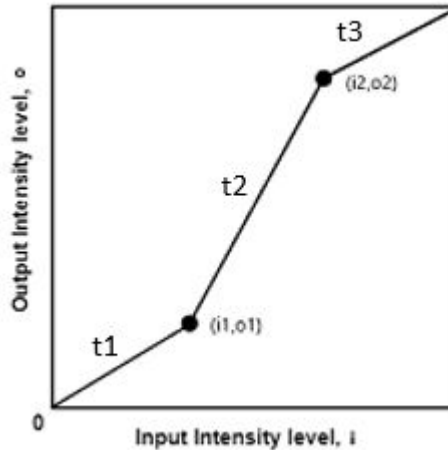
i_1 and i_2 represent two pixel intensities in the input image's range and o_1 and o_2 represent two pixel intensities in the rectified output image's range.

These equations show the how the elements in *T* are calculated.

$$t_1 = \frac{o_1}{i_1}[0 : i_1]$$

$$t_2 = (((\frac{o_2 - o_1}{i_2 - i_1})[(i_1 + 1) : i_2]) - ((\frac{o_2 - o_1}{i_2 - i_1})i_1)) + o_1$$

$$t_3 = (((\frac{255 - o_2}{255 - i_2})[(i_2 + 1) : 255]) - ((\frac{255 - o_2}{255 - i_2})i_2)) + o_2$$

$$T = [t_1 \quad t_2 \quad t_3]$$

5c. *Replace intensity values*: This step converts the pixel intensities of the defogged image to the stretched intensity values. Each pixel value in the defogged image is replaced with the corresponding intensity in *T*.

HDL Implementation

The example model implements the algorithm using a steaming pixel format and fixed-point blocks from Vision HDL Toolbox. The serial interface mimics a real time system and is efficient for hardware designs because less memory is required to store pixel data for computation. The serial interface also allows the design to operate independently of image size and format and makes it more resilient to timing errors. Fixed-point data types use fewer resources and give better performance on FPGA. The necessary variables for the example are initialized in the **InitFcn** callback.
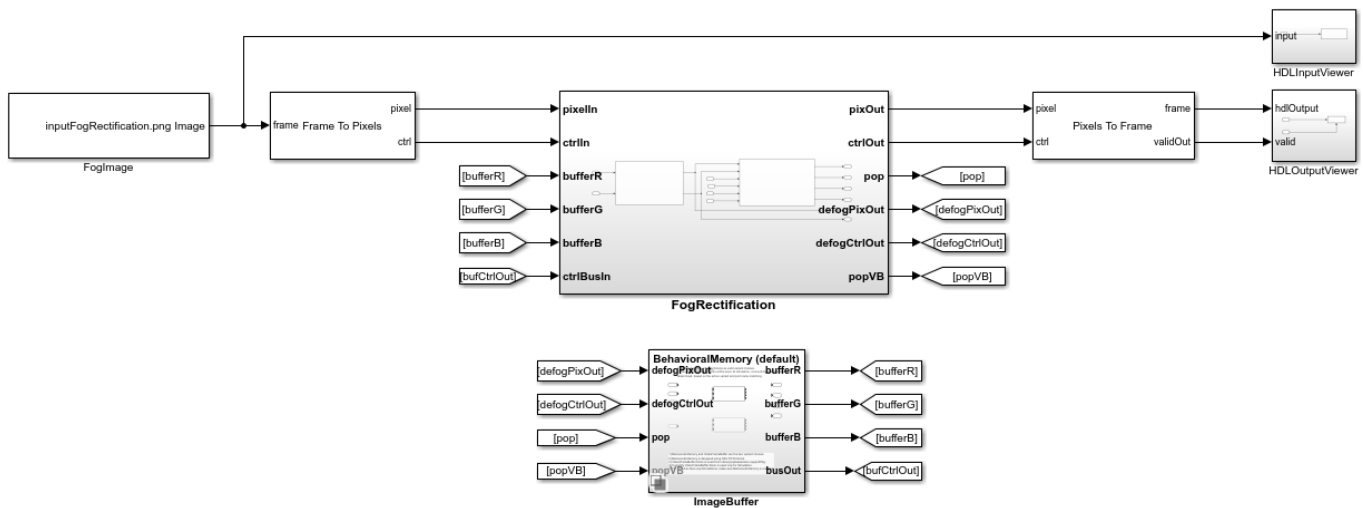
The FogImage block imports the input image to the model. The Frame To Pixels block converts the input frames to a pixel stream of `uint8` values and a `pixelcontrol` bus. The Pixels To Frame block converts the pixel stream back to image frames. The hdlInputViewer subsystem and hdlOutputViewer subsystem show the foggy input image and the defogged enhanced output image, respectively. The ImageBuffer subsystem stores the defogged image so the Contrast Enhancement stages can read it as needed.

The FogRectification subsystem includes the fog removal and contrast enhancement algorithms, implemented with fixed-point datatypes.
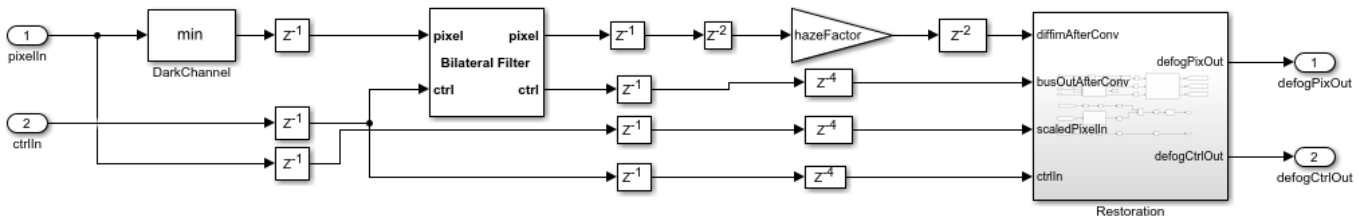
In the FogRemoval subsystem, a Minimum block named DarkChannel calculates the dark channel intensity by finding the minimum across all three components. Then a Bilateral Filter block refines the dark channel results. The filter block has the spatial standard deviation set to 2 and the intensity standard deviation set to 0.5. These parameters are used to derive the filter coefficients. The bit width of the output from filter stage is the same as that of the input.

Next, the airlight image is calculated by multiplying the refined dark channel with a haze factor, 0.9. Multiplying by this factor after the bilateral filter avoids precision loss that would occur from truncating to the maximum 16-bit input size of the bilateral filter.

The Restoration subsystem removes the airlight from the image and then scales the image to prevent over-smoothing. The Pixel Stream Aligner block aligns the input pixel stream with the airlight image before subtraction. The scale factor, m, is found from the midpoint of the difference between the original image and the image with airlight removed. The Restoration subsystem returns a defogged image that has low contrast. So, contrast enhancement must be performed on this image to increase the visibility.

The output from the FogRemoval subsystem is stored in the Image Buffer. The ContrastEnhancement subsystem asserts a `pop` signal to read the frame back from the buffer.
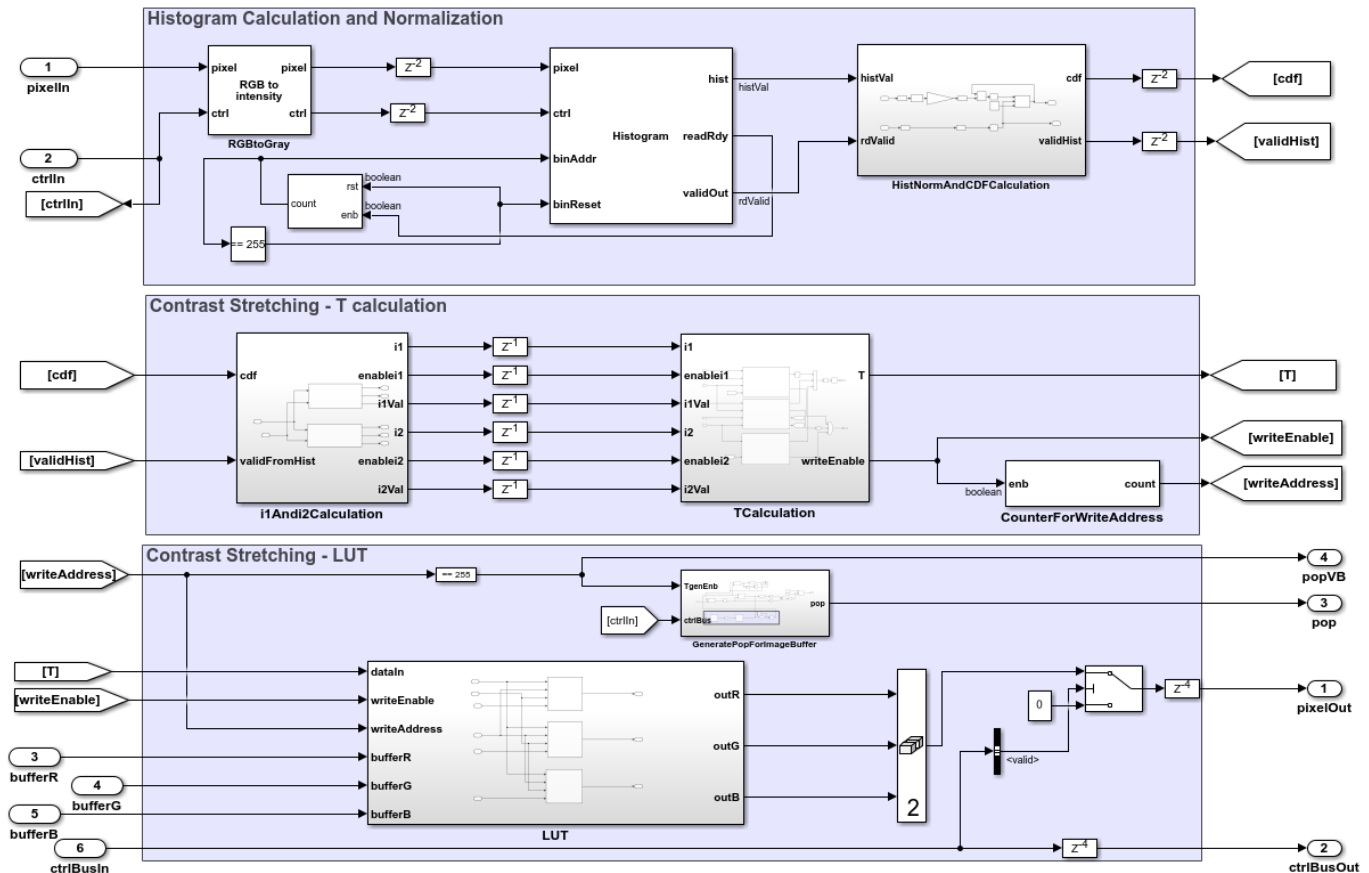
The ContrastEnhancement subsystem uses the Color Space Converter block to convert the RGB defogged image to a grayscale image. Then the Histogram block computes the histogram of pixel intensity values. When the histogram is complete, the block generates a **readRdy** signal. Then the HistNormAndCDFCalculation subsystem normalizes the histogram values and computes the CDF.

The i1Andi2Calculation subsystem computes the i_1 and i_2 values that describe the input intensity range. Then the TCalculation subsystem returns the list of target output intensity values. These 256 values are written into a lookup table. The logic in the Contrast Stretching-LUT area generates a **pop** signal to read the pixel intensities of the defogged image from the Image Buffer, and feeds these values as read addresses to the LUT. The LUT returns the corresponding stretched intensity values defined in T to replace the pixel values in the defogged image.

1) Add Subsystem or Model blocks as valid variant choices.
2) You cannot connect blocks at this level. At simulation, connectivity is automatically determined, based on the active variant and port name matching.
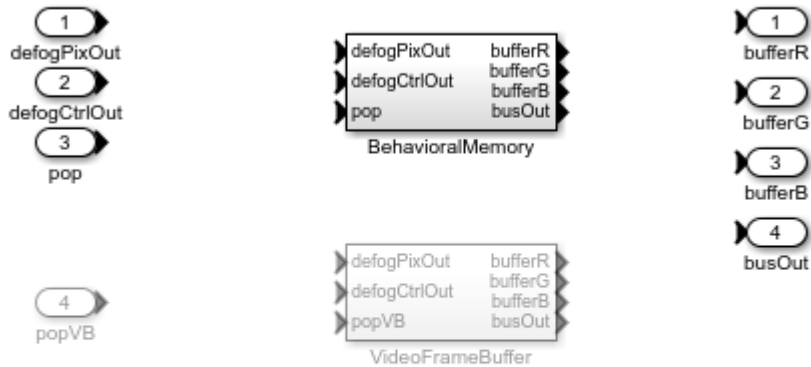
1) Behavioral Memory and VideoFrameBuffer are the two variant choices.
2) Behavioral Memory is designed using HDL FIFO blocks.
3) VideoFrameBuffer block is used from xilinxzynqbasedvision supportPkg.
4) Currently VideoFrameBuffer block is used only for Simulation.
5) For FPGA-in-the-Loop Simulations, make sure Behavioral Memory is enabled.

The Image Buffer subsystem contains two options for modeling the connection to external memory. It is a variant subsystem where you can select between the BehavioralMemory subsystem and the "Model Frame Buffer Interface" (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware) block.
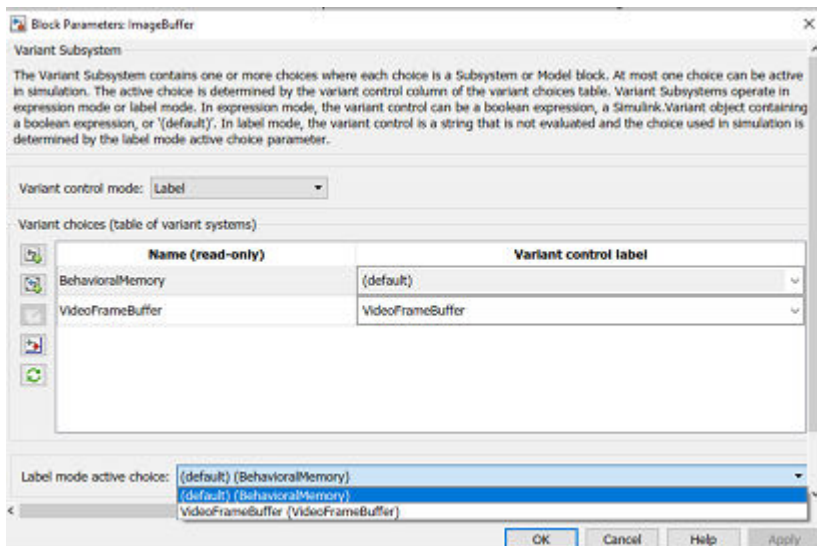
Use the BehavioralMemory subsystem if you do not have the support package mentioned below. This block contains HDL FIFO blocks. The BehavioralMemory returns the stored frame when it receives a pop request signal. The pop request to BehavioralMemory must be high for every row of the frame.

The Video Frame Buffer block requires the Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware™. With the proper reference design, the support package can map this block to an AXI-Stream VDMA buffer on the board. This frame buffer returns the stored frame when it receives the popVB request signal. The pop request to this block must be high only one cycle per frame.

The inputs to the Image Buffer subsystem are the pixel stream and control bus generated after fog removal. The pixel stream is fetched during the Contrast Enhancement operation, after the stretched intensities (*T*) are calculated.

Simulation and Results

This example uses an RGB 240-by-320 pixel input image. Both the input pixels and the enhanced output pixels use the `uint8` data type. This design does not have multipixel support.

The figure shows the input and the enhanced output images obtained from the FogRectification subsystem.
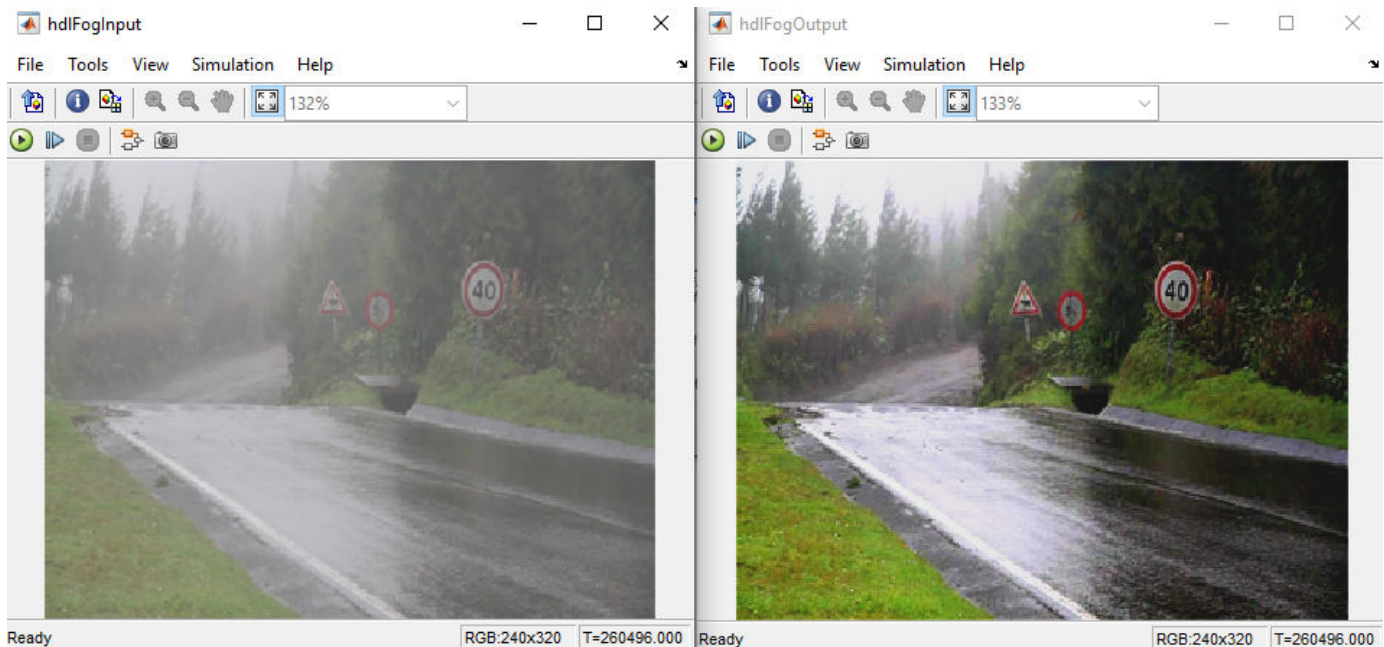
You can generate HDL code for the FogRectification subsystem. An HDL Coder™ license is required to generate HDL code. This design was synthesized for the Intel® Arria® 10 GX (115S2F45I1SG) FPGA. The table shows the resource utilization. The HDL design achieves a clock rate of over 200 MHz.

```
% ================================================================
% |Model Name              ||       FogRectificationHDL    ||
% ================================================================
% |Input Image Resolution  ||          320 x 240           ||
% |ALM Utilization         ||          11088               ||
% |Total Registers         ||          20776               ||
% |Total RAM Blocks        ||          71                  ||
% |Total DSP Blocks        ||          39                  ||
% ================================================================
```

Pixel-Streaming Design in MATLAB

This example demonstrates a workflow for designing pixel-stream video processing algorithms using Vision HDL Toolbox™ in the MATLAB® environment and generating HDL code from the design.

This workflow tests the design using a small thumbnail image to reduce simulation time. To simulate larger images, such as 1080p video format, use MATLAB Coder™ to accelerate the simulation. See "Accelerate a Pixel-Streaming Design Using MATLAB Coder".

Test Bench

In the test bench `PixelStreamingDesignHDLTestBench.m`, the **videoIn** object reads each frame from a video source, and the **scaler** object reduces this frame from 240p to a thumbnail size for the sake of simulation speed. This thumbnail image is passed to the **frm2pix** object, which converts the full image frame to a stream of pixels and control structures. The function `PixelStreamingDesignHDLDesign.m` is then called to process one pixel (and its associated control structure) at a time. After we process the entire pixel-stream and collect the output stream, the **pix2frm** object converts the output stream to full-frame video. The **viewer** object displays the output and original images side-by-side.

The workflow above is implemented in the following lines of `PixelStreamingDesignHDLTestBench.m`.

```
...
for f = 1:numFrm
    frmFull = step(videoIn);        % Get a new frame
    frmIn = step(scaler,frmFull);   % Reduce the frame size

    [pixInVec,ctrlInVec] = step(frm2pix,frmIn);
    for p = 1:numPixPerFrm
        [pixOutVec(p),ctrlOutVec(p)] = PixelStreamingDesignHDLDesign(pixInVec(p),ctrlInVec(
    end
    frmOut = step(pix2frm,pixOutVec,ctrlOutVec);

    step(viewer,[frmIn frmOut]);
end
...
```

Both **frm2pix** and **pix2frm** are used to convert between full-frame and pixel-stream domains. The inner for-loop performs pixel-stream processing. The rest of the test bench performs full-frame processing (i.e., **videoIn**, **scaler**, and **viewer**).

Before the test bench terminates, frame rate is displayed to illustrate the simulation speed.

Pixel-Stream Design

The function defined in `PixelStreamingDesignHDLDesign.m` accepts a pixel stream and five control signals, and returns a modified pixel stream and control signals. For more information on the streaming pixel protocol used by System objects from the Vision HDL Toolbox, see "Streaming Pixel Interface" on page 1-2.

In this example, the function contains the Gamma Corrector System object.

The focus of this example is the workflow, not the algorithm design itself. Therefore, the design code is quite simple. Once you are familiar with the workflow, it is straightforward to implement advanced

video algorithms by taking advantage of the functionality provided by the System objects from Vision HDL Toolbox.

Simulate the Design

Simulate the design with the test bench prior to HDL code generation to make sure there are no runtime errors.

```
PixelStreamingDesignHDLTestBench;
```

```
10 frames have been processed in 35.68 seconds.
Average frame rate is 0.28 frames/second.
```
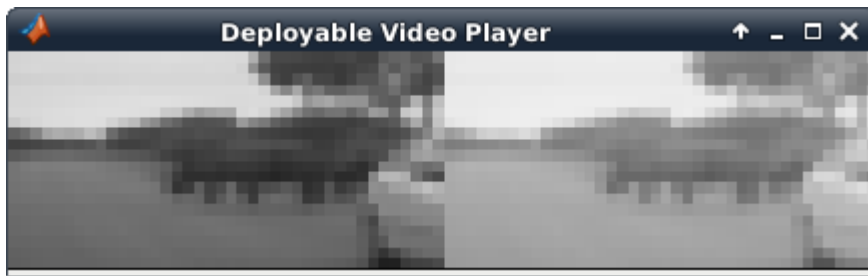

The **viewer** displays the original video on the left, and the output on the right. One can clearly see that the gamma operation results in a brighter image.

Enter the following command to create a new HDL Coder™ project,

```
coder -hdlcoder -new PixelStreamingDesignProject
```

Then, add the file `PixelStreamingDesignHDLDesign.m` to the project as the MATLAB Function and `PixelStreamingDesignHDLTestBench.m` as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" (HDL Coder) for a tutorial on creating and populating MATLAB HDL Coder projects.

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

Enhanced Edge Detection from Noisy Color Video

This example demonstrates how to develop a complex pixel-stream video processing algorithm, accelerate its simulation using MATLAB Coder™, and generate HDL code from the design. The algorithm enhances the edge detection from noisy color video. You must have a MATLAB Coder license to run this example.

It is beneficial to go over both "Pixel-Streaming Design in MATLAB" on page 2-155 and "Accelerate a Pixel-Streaming Design Using MATLAB Coder" examples before proceeding to this one.

Test Bench

In the EnhancedEdgeDetectionHDLTestBench.m file, the **videoIn** object reads each frame from a color video source, and the imnoise function adds salt and pepper noise. This noisy color image is passed to the **frm2pix** object, which converts the full image frame to a stream of pixels and control structures. The function EnhancedEdgeDetectionHDLDesign.m is then called to process one pixel (and its associated control structure) at a time. After we process the entire pixel-stream and collect the output stream, the **pix2frm** object converts the output stream to full-frame video. A full-frame reference design EnhancedEdgeDetectionHDLReference.m is also called to process the noisy color image. Its output is compared with that of the pixel-stream design. The function EnhancedEdgeDetectionHDLViewer.m is called to display video outputs.

The workflow above is implemented in the following lines of EnhancedEdgeDetectionHDLTestBench.m.

```
...
frmIn = zeros(actLine,actPixPerLine,3,'uint8');
for f = 1:numFrm
    frmFull = step(videoIn);                    % Get a new frame
    frmIn = imnoise(frmFull,'salt & pepper'); % Add noise

    % Call the pixel-stream design
    [pixInVec,ctrlInVec] = step(frm2pix,frmIn);
    for p = 1:numPixPerFrm
        [pixOutVec(p),ctrlOutVec(p)] = EnhancedEdgeDetectionHDLDesign(pixInVec(p,:),ctrlInVec(p
    end
    frmOut = step(pix2frm,pixOutVec,ctrlOutVec);

    % Call the full-frame reference design
    [frmGray,frmDenoise,frmEdge,frmRef] = visionhdlenhancededge_reference(frmIn);

    % Compare the results
    if nnz(imabsdiff(frmRef,frmOut))>20
        fprintf('frame %d: reference and design output differ in more than 20 pixels.\n',f);
        return;
    end

    % Display the results
    EnhancedEdgeDetectionHDLViewer(actPixPerLine,actLine,[frmGray frmDenoise uint8(255*[frmEdge
end
...
```

Since frmGray and frmDenoise are uint8 data type while frmEdge and frmOut are logical, **uint8(255x[frmEdge frmOut])** maps logical false and true to uint8(0) and uint8(255), respectively, so that matrices can be concatenated.

Both **frm2pix** and **pix2frm** are used to convert between full-frame and pixel-stream domains. The inner for-loop performs pixel-stream processing. The rest of the test bench performs full-frame processing.

Before the test bench terminates, frame rate is displayed to illustrate the simulation speed.

For the functions that do not support C code generation, such as `tic`, `toc`, `imnoise`, and `fprintf` in this example, use **coder.extrinsic** to declare them as extrinsic functions. Extrinsic functions are excluded from MEX generation. The simulation executes them in the regular interpreted mode. Since `imnoise` is not included in the C code generation process, the compiler cannot infer the data type and size of frmIn. To fill in this missing piece, we add the statement **frmIn = zeros(actLine,actPixPerLine,3,'uint8')** before the outer for-loop.

Pixel-Stream Design

The function defined in `EnhancedEdgeDetectionHDLDesign.m` accepts a pixel stream and a structure consisting of five control signals, and returns a modified pixel stream and control structure. For more information on the streaming pixel protocol used by System objects from the Vision HDL Toolbox, see the "Streaming Pixel Interface" on page 1-2.

In this example, the **rgb2gray** object converts a color image to grayscale, **medfil** removes the salt and pepper noise. **sobel** highlights the edge. Finally, the **mclose** object performs morphological closing to enhance the edge output. The code is shown below.

```
[pixGray,ctrlGray] = step(rgb2gray,pixIn,ctrlIn);          % Convert RGB to grayscale
[pixDenoise,ctrlDenoise] = step(medfil,pixGray,ctrlGray);  % Remove noise
[pixEdge,ctrlEdge] = step(sobel,pixDenoise,ctrlDenoise);   % Detect edges
[pixClose,ctrlClose] = step(mclose,pixEdge,ctrlEdge);      % Apply closing
```

Full-Frame Reference Design

When designing a complex pixel-stream video processing algorithm, it is a good practice to develop a parallel reference design using functions from the Image Processing Toolbox™. These functions process full image frames. Such a reference design helps verify the implementation of the pixel-stream design by comparing the output image from the full-frame reference design to the output of the pixel-stream design.

The function `EnhancedEdgeDetectionHDLReference.m` contains a similar set of four functions as in the `EnhancedEdgeDetectionHDLDesign.m`. The key difference is that the functions from Image Processing Toolbox process full-frame data.

Due to the implementation difference between `edge` function and `visionhdl.EdgeDetector` System object, reference and design output are considered matching if frmOut and frmRef differ in no greater than 20 pixels.

Create MEX File and Simulate the Design

Generate and execute the MEX file.

```
codegen('EnhancedEdgeDetectionHDLTestBench');
EnhancedEdgeDetectionHDLTestBench_mex;
```

```
frame 1: reference and design output differ in more than 20 pixels.
```
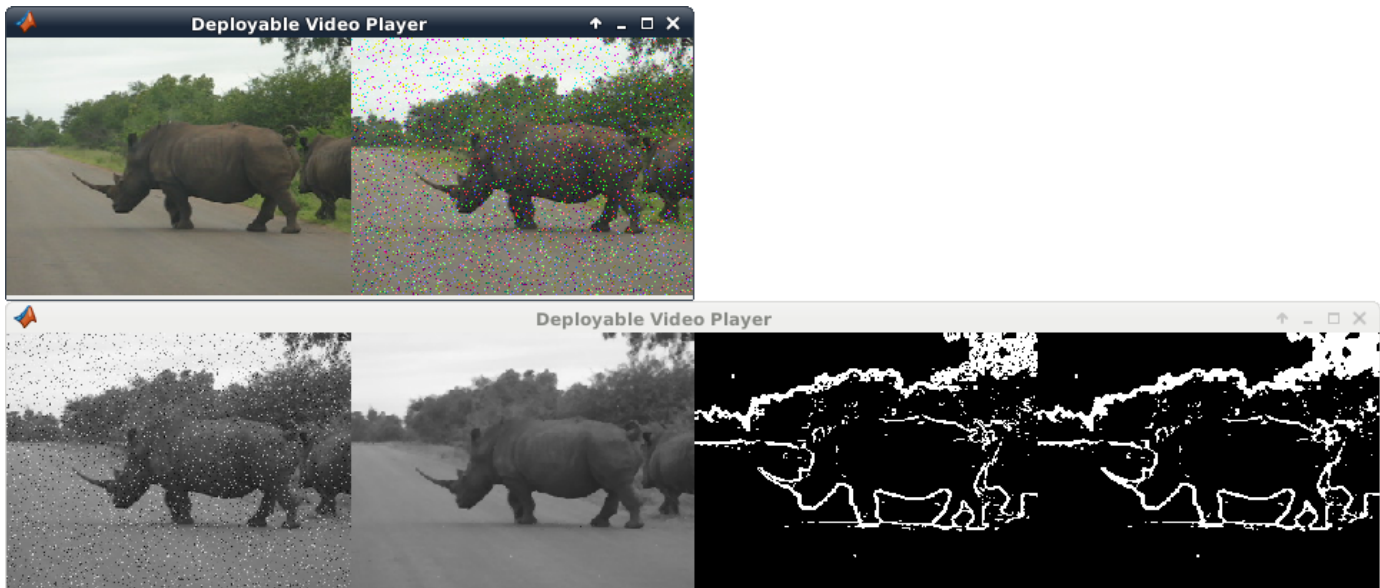
The upper video player displays the original color video on the left, and its noisy version after adding salt and pepper noise on the right. The lower video player, from left to right, represents: the grayscale image after color space conversion, the de-noised version after median filter, the edge output after edge detection, and the enhanced edge output after morphological closing operation.

Note that in the lower video chain, only the enhanced edge output (right-most video) is generated from pixel-stream design. The other three are the intermediate videos from the full-frame reference design. To display all of the four videos from the pixel-stream design, you would have written the design file to output four sets of pixels and control signals, and instantiated three more **visionhdl.PixelsToFrame** objects to convert the three intermediate pixel streams back to frames. For the sake of simulation speed and the clarity of the code, this example does not implement the intermediate pixel-stream displays.

HDL Code Generation

To create a new project, enter the following command in the temporary folder

```
coder -hdlcoder -new EnhancedEdgeDetectionProject
```

Then, add the file 'EnhancedEdgeDetectionHDLDesign.m' to the project as the MATLAB Function and 'EnhancedEdgeDetectionHDLTestBench.m' as the MATLAB Test Bench.

Refer to "Getting Started with MATLAB to HDL Workflow" (HDL Coder) for a tutorial on creating and populating MATLAB HDL Coder projects.

Launch the Workflow Advisor. In the Workflow Advisor, right-click the 'Code Generation' step. Choose the option 'Run to selected task' to run all the steps from the beginning through HDL code generation.

Examine the generated HDL code by clicking the links in the log window.

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?" on page 3-3
"HDL Code Generation Support in Vision HDL Toolbox" on page 3-3
"Streaming Pixel Interface in HDL" on page 3-3

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.

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" on page 2-155

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.

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, copy it 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.

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".
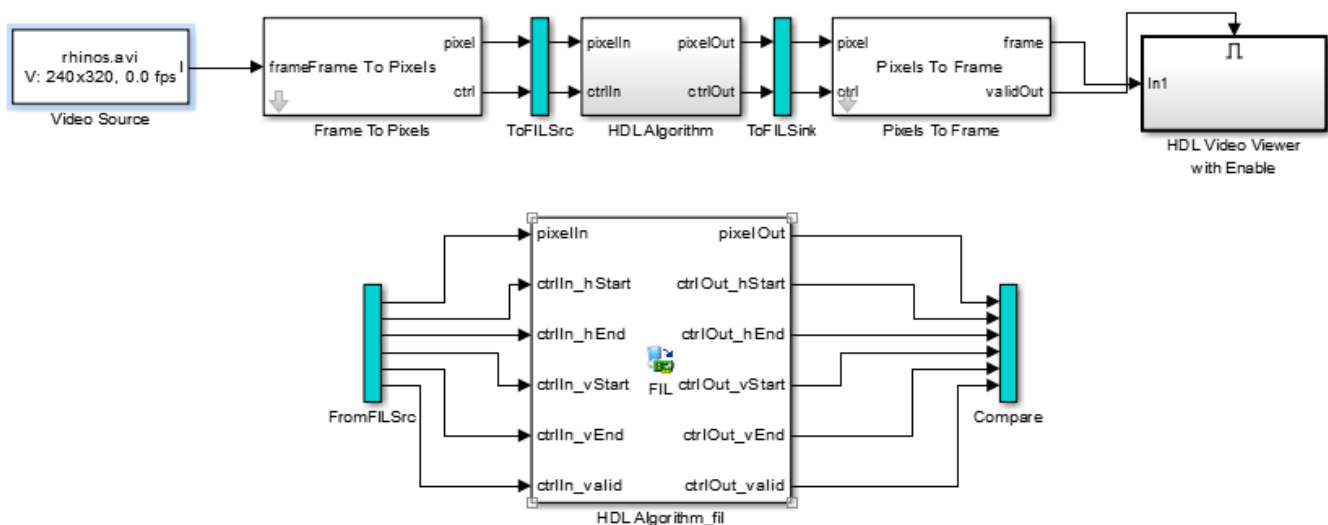
The top part of the model replicates your Simulink design. The generated FIL block at the bottom communicates with the FPGA. ToFILSrc subsystem copies the pixel-stream input of the HDL Algorithm block to the FromFILSrc subsystem. The ToFILSink subsystem copies the pixel-stream output of the HDL Algorithm 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.
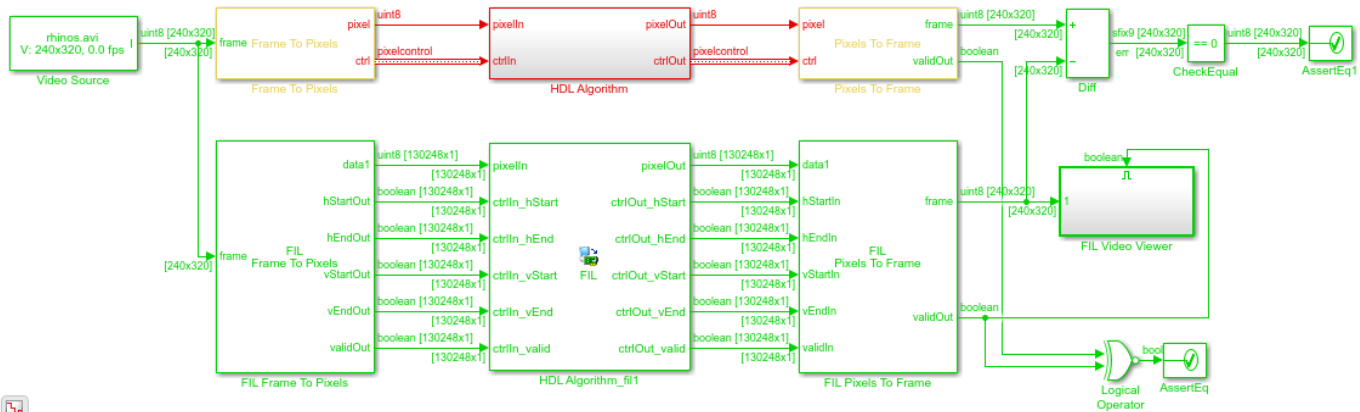

To convert from the generated model to the modified model:

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.
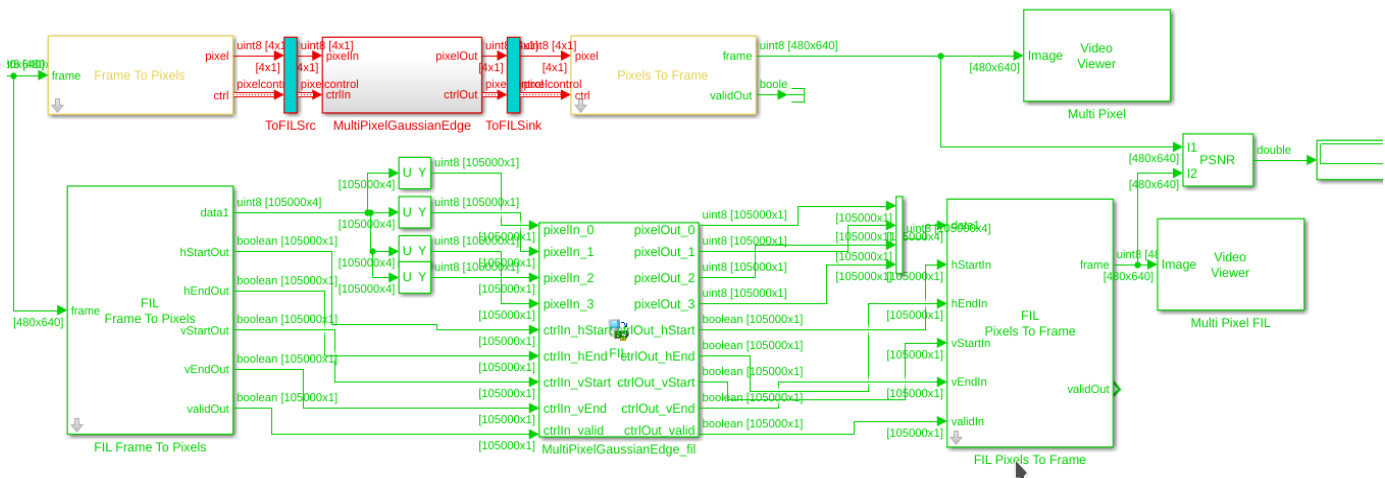
Also, in **Configuration Parameters > HDL Code Generation > Global Settings > Coding style**, select the **Scalarize vector ports** checkbox.
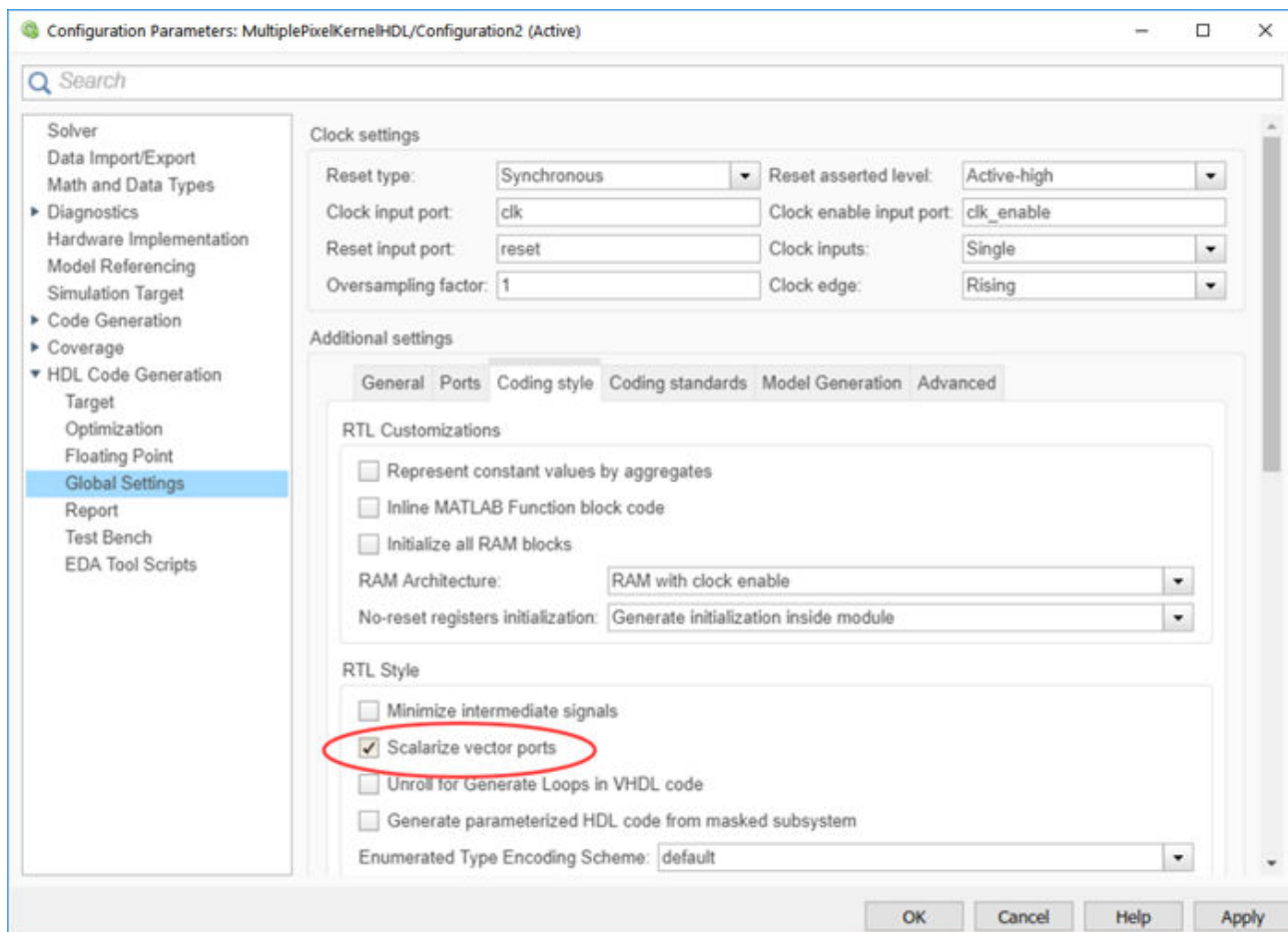
FPGA-in-the-Loop Simulation with Vision HDL Toolbox System Objects

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" on page 2-155. 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 ), ctrlInVe
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_vStart,ctrlIn

  ...

% 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" on page 2-155. 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( pixInVec( p )
%          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]',[ctrlInVec.vEnd]
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),ctrlInVec(
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.
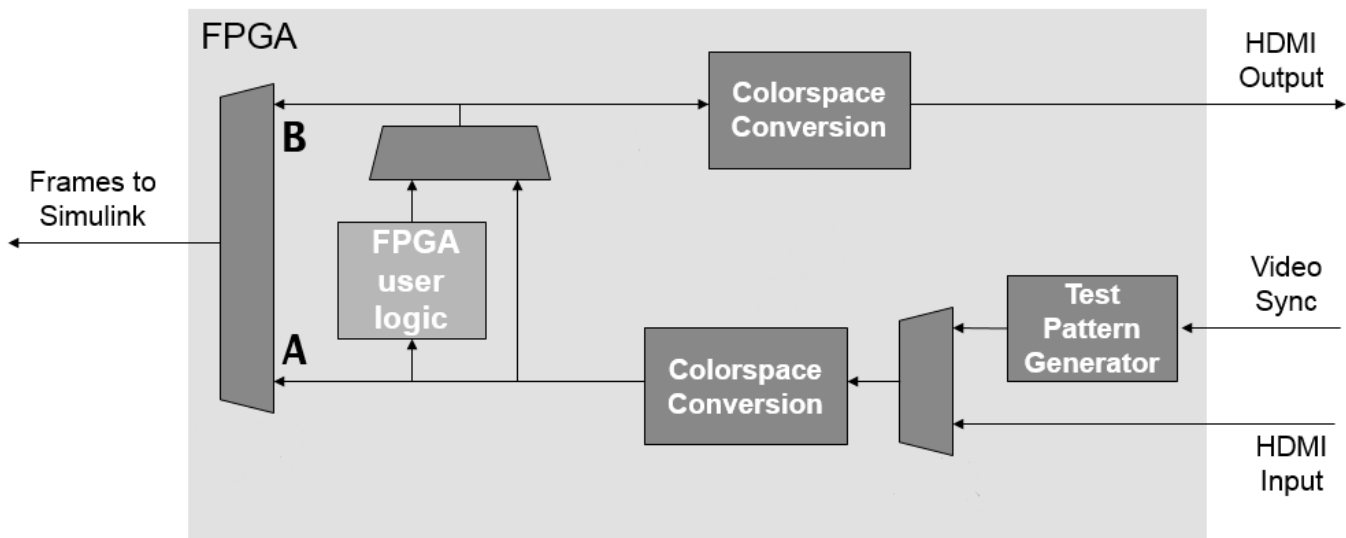
Note The reference design on the Zynq device requires the same video resolution and color format for the entire data path. The resolution you select must match that of your camera input. The design you target to the user logic section of the FPGA must not modify the frame size or color space of the video 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"

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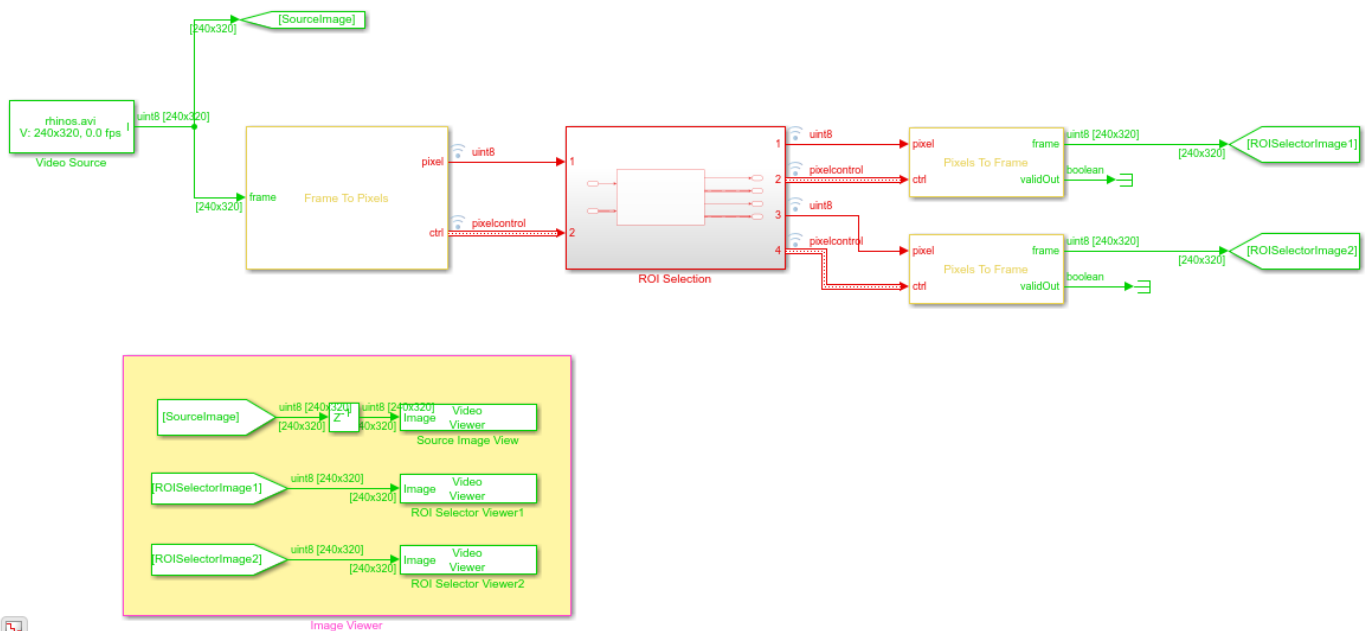
The example model includes a Video Source block that contains a 240p video sample. Each pixel is a scalar `uint8` value that represents intensity. The green and red lines represent full-frame processing and pixel-stream processing, respectively.

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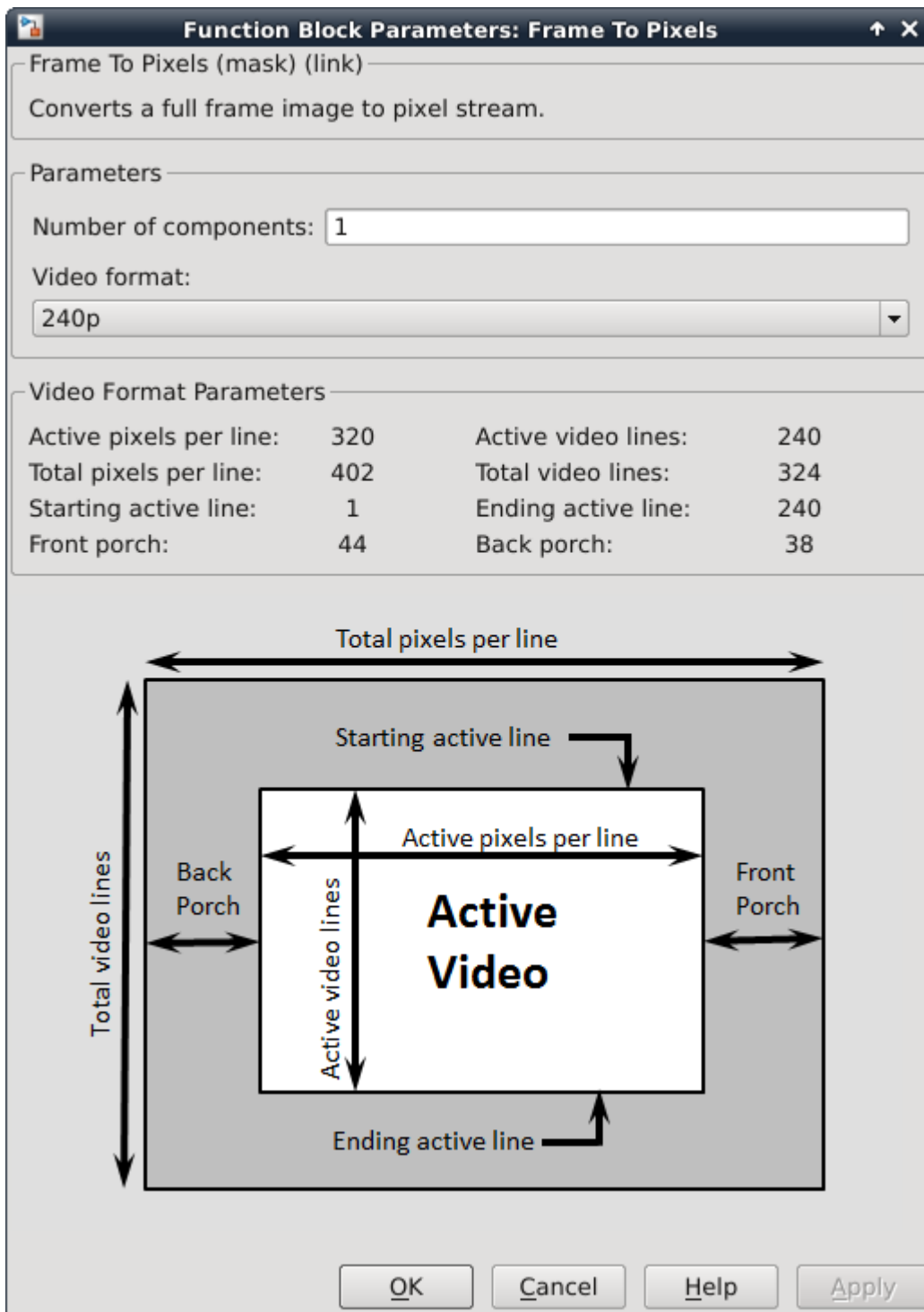
The **Number of components** parameter is set to 1 for grayscale image input, and the **Video format** parameter is 240p to match the video source.

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.
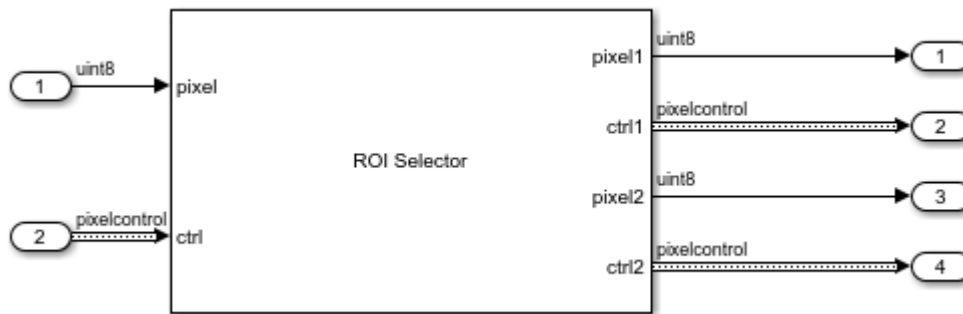
Use the ROI Selector block to select regions of interest. You can use the **Regions** parameter to experiment with different region sizes and examine their effect on the output frames. In this model, the **Regions** parameter is set to `[100 100 50 50;220 170 100 70]` which represents two regions, each 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.
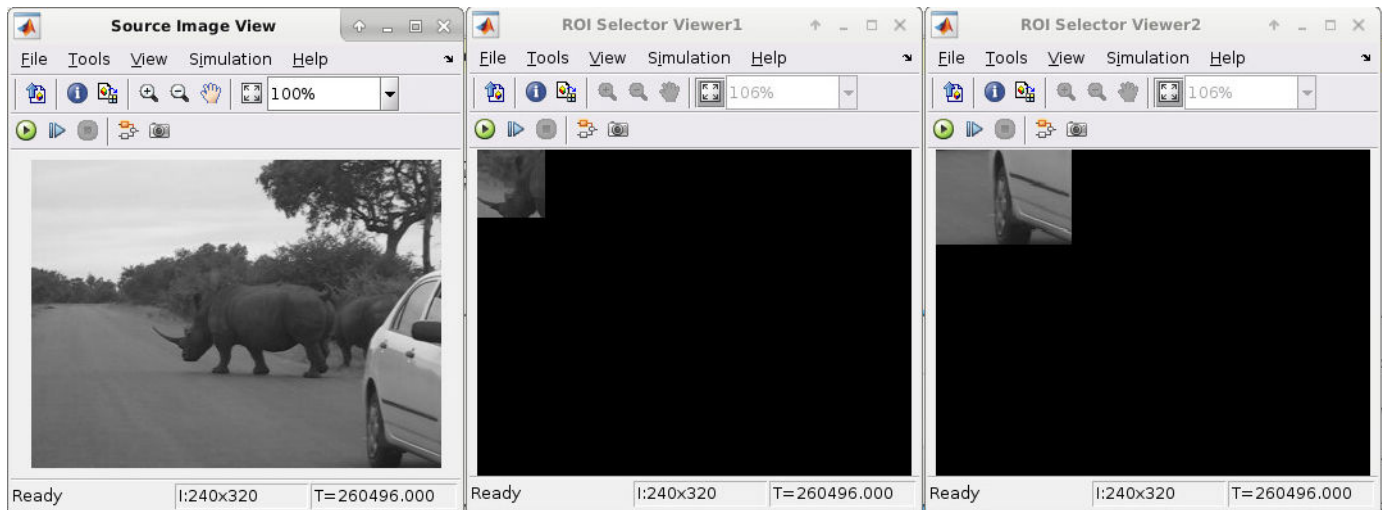
The Unit Delay block on the top level of the model is to time-align the matrices for a fair comparison.

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')
```
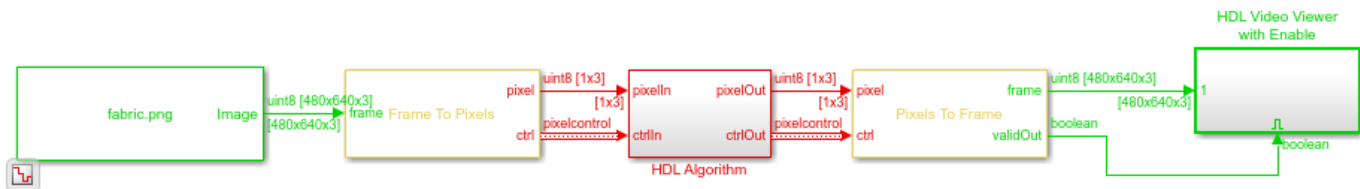
See Also

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.
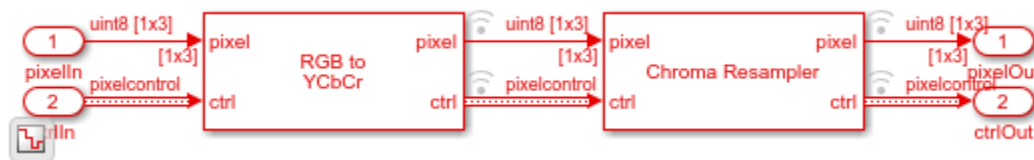
The waveform of the input and output pixel stream of the Chroma Resampler block shows the downsampling of the CbCr component values. The latency of the Chroma Resampler block depends on the size of the antialiasing filter. This example uses the default filter, which has 29 taps.
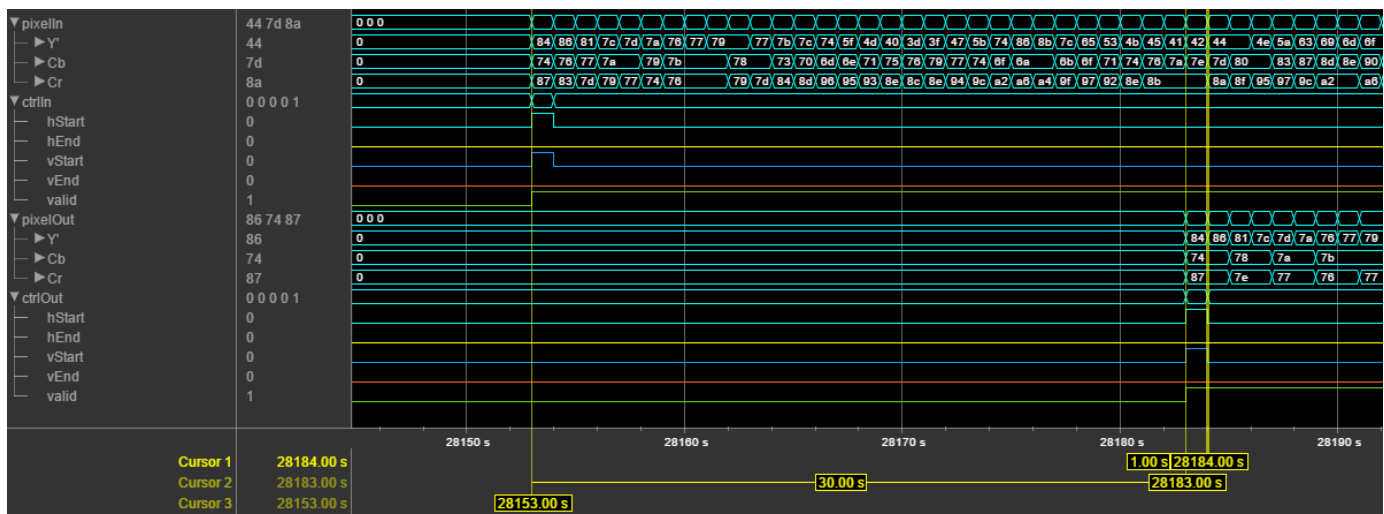
To check and generate the HDL code referenced in this example, you must have an HDL Coder™ license.

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.
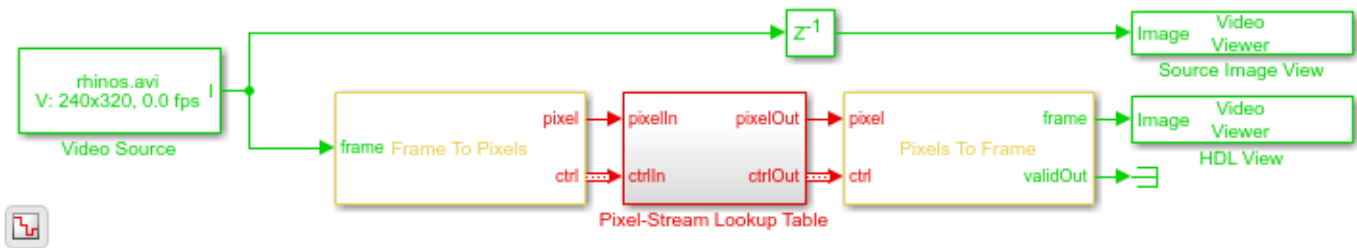
See Also

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.
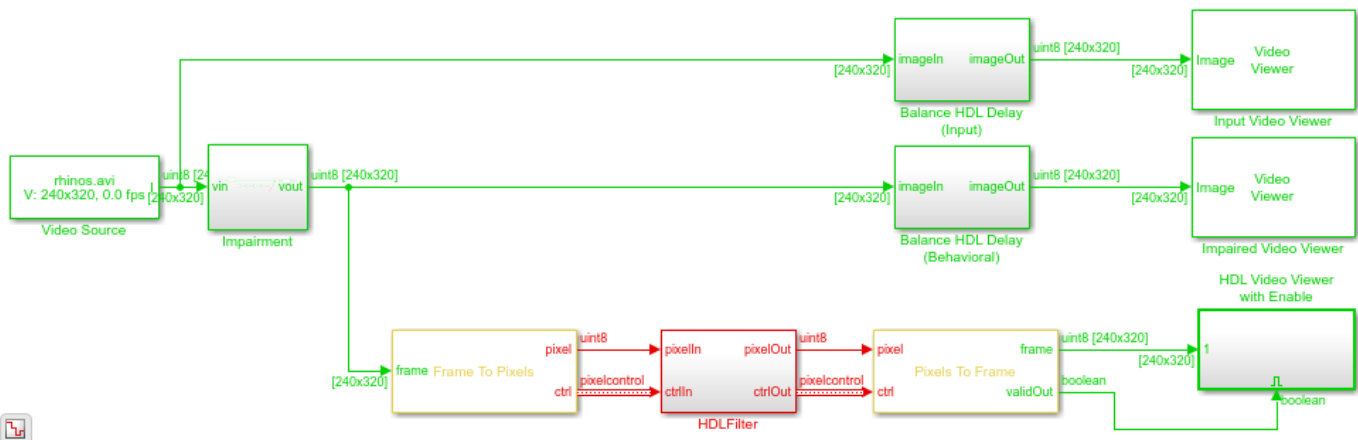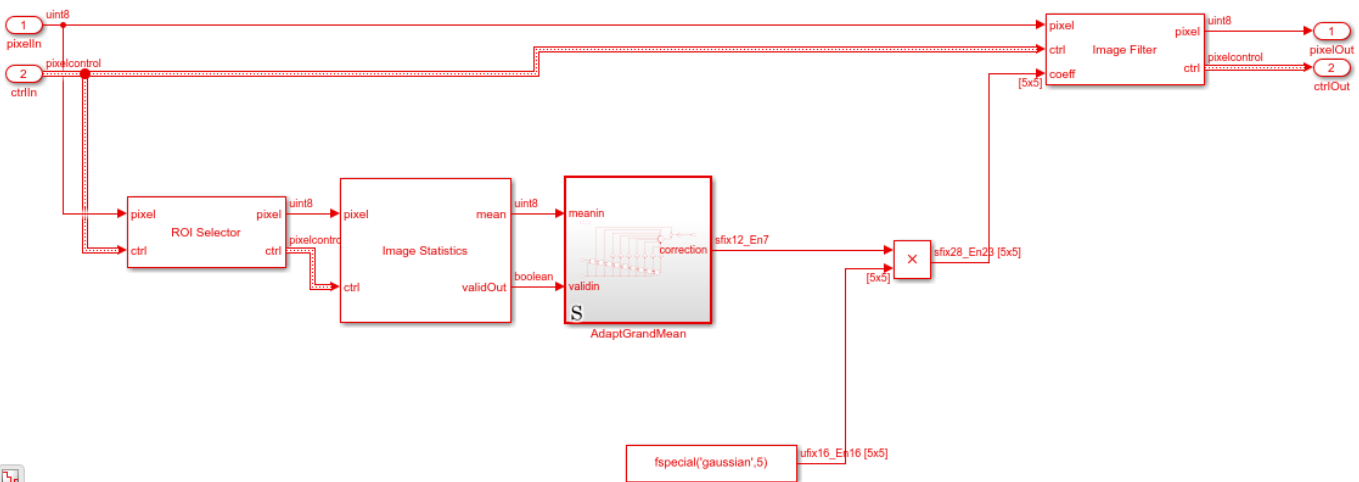

The Impairment

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.
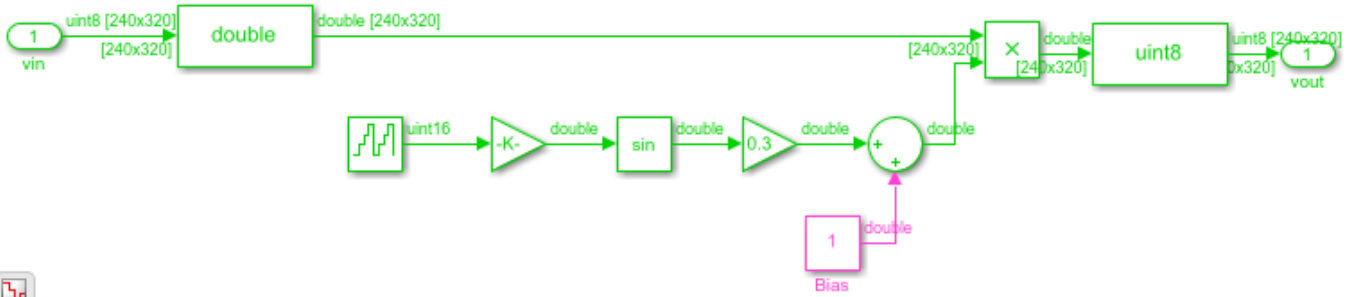
The Filter Algorithm

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.
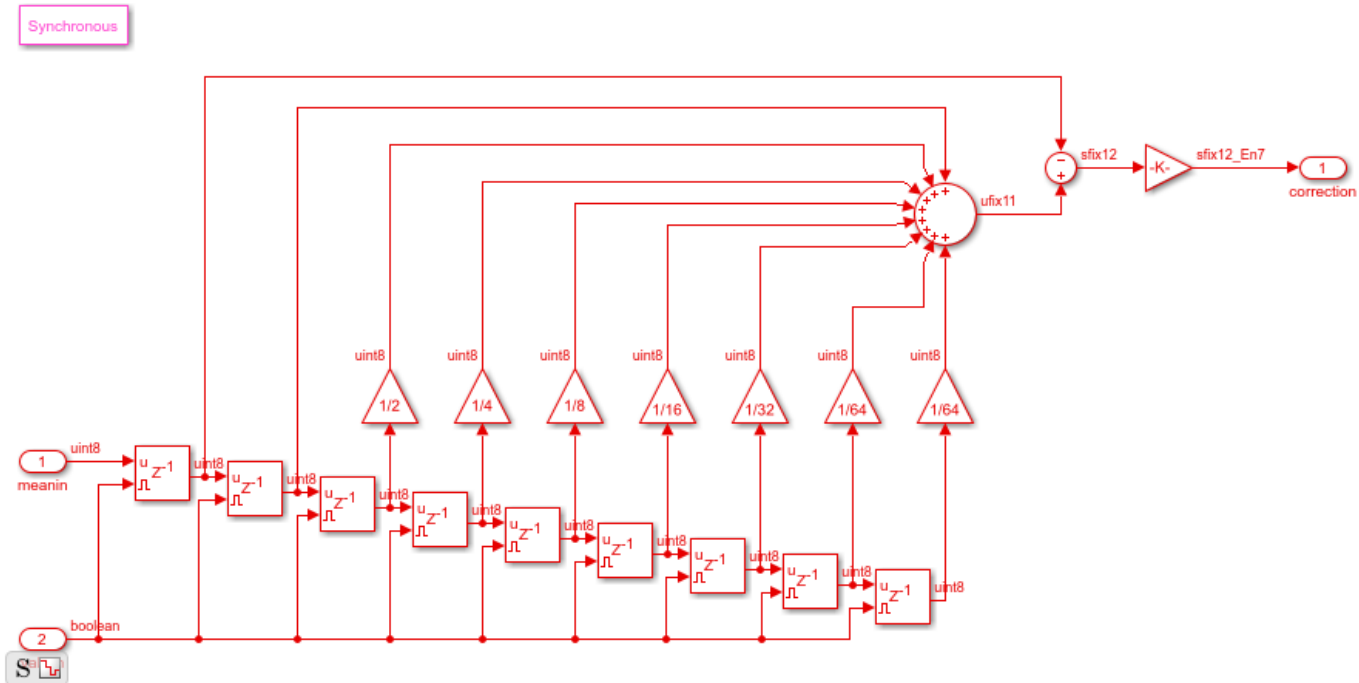
Apply the Correction

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