SIEMENS EDA

Algorithmic C (AC) Image Processing Library Reference Manual

Software Version v4.0.0
February 2025

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Chapter 1: Introduction to ac_ipl

The Algorithmic C Image Processing Library (*ac_ipl*) contains synthesizable C++ functions commonly used in Image Processing operations like dithering and windowing. The functions use a C++ class-based object design so that it is easy to instantiate multiple variations of objects into a more complex subsystem and utilize the AC Datatypes for true bit-accurate behavior.

The input and output arguments of these functions are parameterized so that arithmetic may be performed at the desired fixed point precision and provide a high degree of flexibility on the area/performance trade-off of hardware implementations obtained during Catapult synthesis.

The following sections provide a summary of the *ac_ipl* library:

1.1. Using the ac_ipl Library

In order to utilize any of the top-level *ac_ipl* classes, locate the required *ac_** header file from the ac_ipl directory, and include that in your code. A summary of the files and classes within the ac_ipl directory is given in Summary of Classes. A line that includes the required file for the *ac_dither* class is given below.

```
#include <ac_dither.h>
```

Each class has a public *run()* interface function that is synthesized as the top-level block by catapult and is called by the user in their design/testbench, in accordance with the requirements of class-based hierarchy.

The following coding requirements must be kept in mind while using the *ac_ipl* library. Failure to do so can result in erroneous functioning.

- For frame-based designs, the user must ensure that they write data inputs for a single image every time they call the *run()* function. The user must not pass any more or less data than that.
- If the design is being used for RGB_1PPC color inputs/outputs, the user must ensure that the *TUSER* (sof) and *TLAST* (eol) flags are set correctly. The design depends on this condition to function correctly.
- While image dimension inputs that are mapped to DirectInputs can be changed at run-time, the user must make sure that they do not change while the design is processing a frame in hardware.
- *ac_ipl* libraries often use the *ac_pixels* header file for color inputs/outputs and also often use *static_asserts* themselves. Both of these need C++11 or a later standard for compilation, failing which a *#error* directive will be triggered.

1.2. Summary of Classes

The following tables summarize the classes currently supported in the *ac_ipl* library.

Image Processing

Description	Class Name(s)	Header File
Floyd-Steinberg dithering algorithm	ac_dither	ac_dither.h
Image histogram	ac_imhist	ac_imhist.h
Canny edge detector	ac_canny	ac_canny.h
Color Space Conversion	ac_csc	ac_rgb2ycbcr.h
Color temperature correction	ac_ctc	ac_ctc.h
Local Contrast Normalization	ac_localcontrastnorm	ac_localcontrastnorm.h
Gaussian Image Pyramid	ac_gaussian_pyr	ac_gaussian_pyr.h
Wavelet Image Pyramid	ac_dwt2_pyr	ac_dwt2_pyr.h
Harris Corner Detector	ac_harris	ac_harris.h
Denoising Filter	ac_denoise_filter	ac_denoise_filter.h
Linebuffers	ac_linebuffer ac_linebuffer_1r1w ac_linebuffer_spshift ac_line_buffer_spcircular ac_linebuffer_spwrmask	ac_linebuffer.h ac_linebuffer_1r1w.h ac_linebuffer_spshift.h ac_linebuffer_spcircular.h ac_linebuffer_spwrmask.h
Packed Vector Type	ac_packed_vector	ac_packed_vector.h
Synchronization Flag Generator	ac_flag_gen_1d ac_flag_gen_2d	ac_flag_gen.h

AC Window Blocks

Description	Class Name	Header File
Continuous read and writing of 1D data using control flags to denote the start/end of the frame	ac_window_1d_flag	ac_window_1d_flag.h
Continuous read and writing of 2D data using control flags to denote the start/end of the frame	ac_window_2d_flag	ac_window_2d_flag.h
Continuous stream of 1D data using ac_channels on input and output	ac_window_1d_stream	ac_window_1d_stream.h
Continuous stream of 2D data using ac_channels on input and output	ac_window_2d_stream	ac_window_2d_stream.h
Process array of data	ac_window_1d_array	ac_window_1d_array.h

AC Window 2.0 Blocks

Description	Class Names	Header File(s)
Linebuffers for Flush Support	ac_linebuffer_flush ac_linebuffer_1r1w_flush ac_line_buffer_spcircular_flush ac_linebuffer_spwrmask_flush	ac_linebuffer_flush.h ac_linebuffer_1r1w_flush.h ac_line_buffer_spcircular_flush.h ac_linebuffer_spwrmask_flush.h
Synchronization Flag Generator for Flush Support	ac_flag_gen_flush_1d ac_flag_gen_flush_2d	ac_flag_gen_flush.h
Synchronization Flag Generator for Line Flushing	ac_flag_gen_lflush_1d ac_flag_gen_lflush_2d	ac_flag_gen_lflush.h
Data and Flag Shifters	ac_shift_N_1d ac_shift_N_2d ac_flag_shift_1d ac_flag_shift_2d	ac_shift_N.h ac_flag_shift.h
Flag Shifters for Flush Support	ac_flag_shift_flush_1d ac_flag_shift_flush_2d	ac_flag_shift_flush.h
Boundary Processing	ac_boundary_1d ac_boundary_2d	ac_boundary.h
Windowed Output Flag Generator	ac_flag_gen_out_1d ac_flag_gen_out_2d	ac_flag_gen_out.h
Windowed Output Flag Generator for Flush Support	ac_flag_gen_out_flush_1d ac_flag_gen_out_flush_2d	ac_flag_gen_out_flush.h
Windowed Output Flag Generator for Line Flushing	ac_flag_gen_out_lflush_1d ac_flag_gen_out_lflush_2d	ac_flag_gen_out_lflush.h
AC Window 2.0 Top-Level Design	ac_window_v2_1d ac_window_v2_2d	ac_window_v2.h
AC Window 2.0 Top-Level Design for Flush Support	ac_window_v2_flush_2d	ac_window_v2_flush.h
AC Window 2.0 Top-Level Design for Line Flushing	ac_window_v2_lflush_2d	ac_window_v2_lflush.h

1.3. Installing the ac_ipl Library

The library consists of the directories and files shown here:

```
.
|-- include
|   |-- ac_ipl
|   |   |-- ac_canny.h
|   |   |-- ac_rgb2ycbcr.h
|   |   |-- ac_ctc.h
|   |   |-- ac_dither.h
```

```
|   |   |-- ac_localcontrastnorm.h
|   |   |-- ac_gaussian_pyr.h
|   |   |-- ac_dwt2_pyr.h
|   |   |-- ac_imhist.h
|   |   |-- ac_harris.h
|   |   |-- ac_packed_vector.h
|   |   |-- ac_denoise_filter.h
|   |   |-- ac_flag_gen.h
|   |   |-- ac_flag_gen_flush.h
|   |   |-- ac_flag_gen_lflush.h
|   |   |-- ac_boundary_enums.h
|   |   |-- ac_flag_shift.h
|   |   |-- ac_shift_N.h
|   |   |-- ac_boundary.h
|   |   |-- ac_linebuffer.h
|   |   |-- ac_linebuffer_1r1w.h
|   |   |-- ac_linebuffer_spshift.h
|   |   |-- ac_linebuffer_spcircular.h
|   |   |-- ac_linebuffer_spwrmask.h
|   |   |-- ac_linebuffer_flush.h
|   |   |-- ac_linebuffer_1r1w_flush.h
|   |   |-- ac_linebuffer_spcircular_flush.h
|   |   |-- ac_linebuffer_spwrmask_flush.h
|   |   |-- ac_flag_gen_out.h
|   |   |-- ac_flag_gen_out_flush.h
|   |   |-- ac_flag_gen_out_lflush.h
|   |   |-- ac_window_v2.h
|   |   |-- ac_window_v2_flush.h
|   |   |-- ac_window_v2_flush_enums.h
|   |   |-- ac_window_v2_lflush.h
|   |   |-- ac_window_v2_lflush_enums.h
|   |   `-- ac_pixels.h
|   |-- ac_buffer_2d.h
|   |-- ac_window_1d_array.h
|   |-- ac_window_1d_flag.h
|   |-- ac_window_1d_stream.h
|   |-- ac_window_2d_flag.h
|   |-- ac_window_2d_stream.h
|   |-- ac_window.h
|   `-- ac_window_structs.h
|-- pdfdocs
|   `-- ac_ipl_ref.pdf
```

In order to utilize this library you must have the AC Datatypes package and the AC Math package installed and configure your software environment to provide the path to the “include” directories of these packages as part of your C++ compilation arguments.

Chapter 2: Image Processing Library

The *ac_ipl* package includes the following image processing blocks:

- Canny Edge Detector (ac_canny)
- Color Space Conversion (ac_csc)
- Color Temperature Correction (ac_ctc)
- Floyd-Steinberg Dithering Algorithm (ac_dither)
- Gaussian Image Pyramid (ac_gaussian_pyr)
- Wavelet Image Pyramid (ac_dwt2_pyr)
- Harris Corner Detector (ac_harris)
- Image Histogram (ac_imhist)
- Local Contrast Normalization (ac_localcontrastnorm)
- Denoising Filter (ac_denoise_filter)
- Line Buffer (ac_linebuffer)
- Packed Vector Type (ac_packed_vector)
- Synchronization Flag Generator (ac_flag_gen)

2.1. Canny Edge Detector (ac_canny)

The ac_canny library provides an efficient hardware implementation for the Canny edge detection algorithm. It can be pipelined with an II of 1 and works with *ac_int* input/output pixels.

2.1.1. Implementation Details and Limitations

The canny edge detection algorithm has several stages. Each stage corresponds to a sub-block of the design, which is coded as a C++ function. The sub-blocks communicate with each other and the top-level wrapper using *ac_channel* interconnects. The output pixels are 1-bit, monochrome pixels which are set to high to denote the detection of an edge, and are set to low otherwise.

Functionality of Sub-blocks

- The first sub-block, *gaussFilter*, applies a 2D Gaussian filter to smooth out and denoise the image.

- The second sub-block, *edgeFilter*, applies the edge detect kernels on the smoothed image to find the magnitude and direction of the edge gradient.

- The third sub-block, *NMS*, applies non-maximum suppression to thin out the edges detected by *edgeFilter*.

- The fourth sub-block, *hysThresh*, applies hysteresis and double thresholding, to eliminate weak edges.

Thresholding Inputs

In addition to the input/output pixel stream and dimension inputs, the top-level design is also supplied the upper and lower threshold values as two separate inputs, which are in turn supplied to the *hysThresh* sub-block. The two threshold values should be empirically determined by the user based on image content. The design cannot determine threshold values by itself, and must rely upon the user to supply them, because threshold calculation is otherwise a bottleneck that can necessitate unreasonably large buffer sizes.

Using Singleport Memories

Each of the sub-blocks uses *ac_window_2d_flag* objects to implement a sliding image window. The linebuffers used for the sliding window implementation are by default intended to be mapped to dual-port memories, if the design is to be pipelined with an II of 1. If the user wishes to pipeline the design with an II of 1 and use single-port memories, they should supply “true” as the value of the USE_SINGLEPORT template parameter. Please refer to Calling the Top-level Function for a relevant coding example.

Catapult Architectural Exploration Options

The design as a whole is pipelined with an II of 1. Every sub-block has two nested row and column loops, which are also pipelined with an II of 1. All other loops are fully unrolled by default. The unrolled loops are mostly loops internal to the *ac_window_2d_flag* code, with the remaining loops being dedicated to analyzing the windowed data or performing convolution, depending on the sub-block. Please refer to Illustration 1: Loop Options for a view of the loops in the Catapult architectural constraints editor. The loops in the sub-block *gaussFilter* are used as an example. The unrolled loops have to be left unrolled or pipelined with an II of 1, if the entire design is to be pipelined with an II of 1.

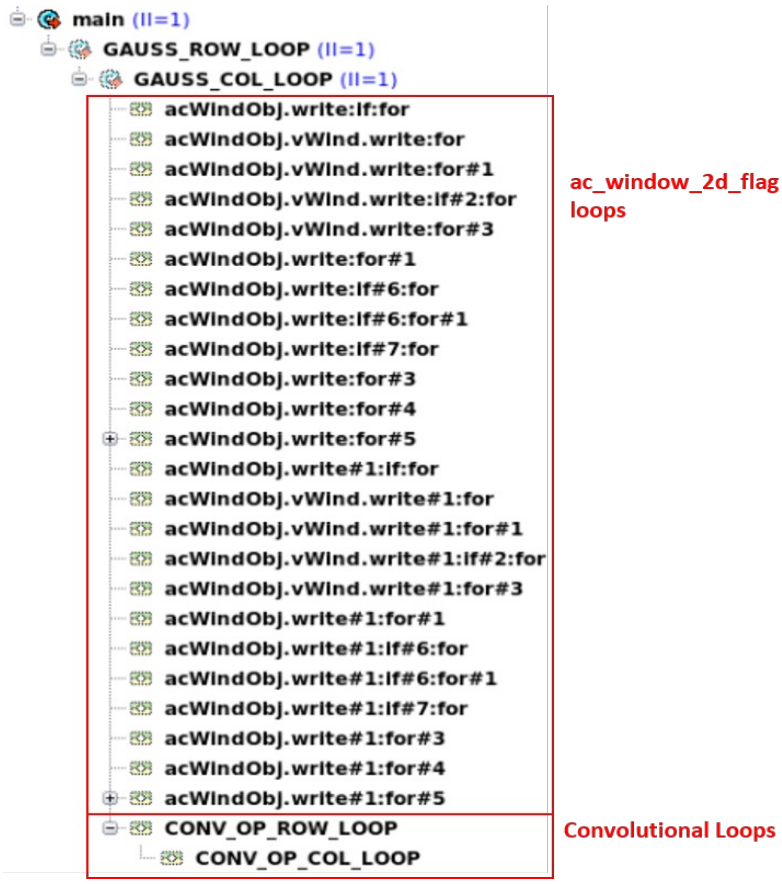

Illustration 1: Loop Options

If the ac_window_2d_flag linebuffer sizes fall above the memory threshold, they will be mapped to memories. The user can facilitate the mapping to single-port memories by using the USE_SINGLEPORT template parameter, as explained in Using Singleport Memories. The other arrays, including the internal ac_window_2d_flag arrays, will be mapped to register banks, provided that the window size stays below the memory threshold. These arrays must be mapped to register banks if the internal ac_window_2d_flag loops are to be fully unrolled, so as to facilitate multiple array accesses per clock cycle.

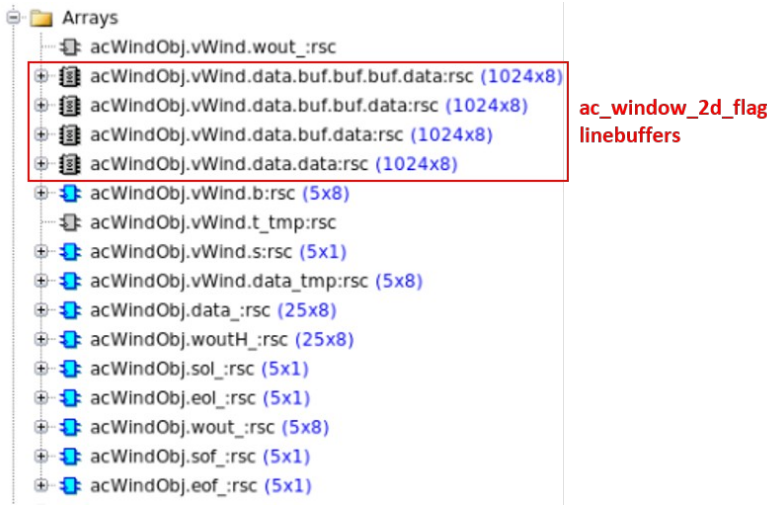

Illustration 2: Array Options

Please refer to Illustration 2: Array Options for a view of the array mapping, as seen in the Catapult architectural constraints editor. Once again, the gaussFilter sub-block is chosen as an example.

Final Output

The result of the canny edge detection library on a sample input image is shown in Illustration 3: Input/Output for Canny Edge Detector. The image on the left is input image, while the image on the right is the corresponding edge detector output.

Illustration 3: Input/Output for Canny Edge Detector

2.1.2. Model Parameters

The canny edge detector library is implemented as a class template named *ac_canny*. A public member function of the class, called *run(..)* acts as the top-level design block.

Given below is a snippet of the *ac_canny* class.

```
template <unsigned CDEPTH, unsigned W_MAX, unsigned H_MAX,
          bool USE_SINGLEPORT = false>
class ac_canny
{
  // Code
  typedef ac_int<CDEPTH, false> pixInType;
  typedef ac_int<1, false> pixOutType;
  typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;
  typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType;

#pragma hls_pipeline_init_interval 1
#pragma hls_design interface
  void CCS_BLOCK(run) (
    ac_channel<pixInType>  &streamIn,   // Pixel input stream
    ac_channel<pixOutType> &streamOut,  // Pixel output stream
    const widthInType      widthIn,     // Input width
    const heightInType     heightIn,    // Input height
    const pixInType        threshLowIn, // Lower threshold for hysteresis
    const pixInType        threshUppIn  // Upper threshold for hysteresis
  )
  // Code
};
```

The definition of the template parameters shown in the snippet is given below:

Parameters	Description
CDEPTH	Color depth of input image.
W_MAX, H_MAX	Maximum input image width and height, respectively.
USE_SINGLEPORT	Default template parameter; helps the user choose a singleport implementation.

Since USE_SINGLEPORT is set to false by default, the default canny edge detector implementation is designed with a dual-port implementation in mind.

2.1.3. Calling the Top-level Function

Consider the hardware canny edge detector implementation with the following static characteristics:

- Color-depth of 8.
- Maximum image width/height of 1024.

The following are the steps the to initialize an object of the *ac_canny* class with the above mentioned static characteristics and call the *run(..)* function, for an example design:

1. Include <ac_canny.h>

2. Define enums and data-type for input pixels/threshold values, output pixels, as well as image dimensions in the CCS_MAIN() function.

```
enum { CDEPTH = 8, W_MAX = 1024, H_MAX = 1024 };
typedef ac_int<CDEPTH, false> pixInType; // Input pixel/threshold value type
typedef ac_int<1, false> pixOutType; // Output pixel type
typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;  // Input width
type
typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType; // Input height
type
```

3. Instantiate *ac_channel*s with the input and output types defined above in the CCS_MAIN() function.

```
ac_channel<pixInType>  streamIn;
ac_channel<pixOutType> streamOut;
```

4. Initialize dimension variables and threshold values.

```
widthInType  widthIn  = 512;
heightInType heightIn = 384;
pixInType threshLowIn = 5;
pixInType threshUppIn = 25;
```

5. Write all input values to the streamIn *ac_channel*. Instantiate an object of the *ac_canny* class with the above static characteristics and call the top level member function with the ac_channels, dimension and threshold variables mentioned above.

```
ac_canny<CDEPTH, W_MAX, H_MAX> cannyObj;
cannyObj.run(streamIn, streamOut, widthIn, heightIn, threshLowIn, threshUp-
pIn);
```

2.2. Color Space Conversion (ac_csc)

The ac_csc library provides an efficient hardware implementation for the Color Space Conversion algorithm. It can be pipelined with an II of 1 and works with input RGB packed vector/output YCbCr packed vector values.

2.2.1. Features

- **Standards-compliant RGB to YCbCr conversion:** RGB to YCbCr conversion implemented accurately according to the BT.601, BT.709, BT.2020, and BT.2100 standards, supporting both full swing and studio swing equations across 8-bit, 10-bit, and 12-bit color depths
- **HLS Optimized**: Efficiently coded in C++ using high-level synthesis (HLS) best practices.

- **Robust Testbench**: Developed a comprehensive testbench for validating output against golden reference values.
- **High QoR**: Synthesized using Catapult HLS with optimized latency, area, and performance.
- **Multi-Level Verification**: Verified using Scverify, CDC, UVM, and CFormal for thorough design validation.
- **Full Coverage Achieved**: Achieved 100% code coverage, functional coverage and >90% RTL coverage.
- **RTL Lint Clean**: Passed all RTL lint checks with adherence to industry coding standards.
- **High Reliability**: End-to-end verification ensured a robust and high-performance design implementation.

2.2.2. Introduction

A color model mathematically describes how colors are represented, independent of specific physical devices like cameras or displays. Common color models include CIE, RGB, YUV, and HSL/HSV. A color space defines a specific implementation of a color model, combining it with precise parameters such as primaries, transfer functions, and white points.

In digital video, international standards specify how color spaces should be coded for different generations of broadcast and display technology:

- BT.601 defines digital coding for standard-definition (SD) video signals, covering both 4:3 and 16:9 formats.
- BT.709 defines digital coding for high-definition (HD) video, standardizing the HD color space and the 16:9 aspect ratio.
- BT.2020 defines digital coding for ultra-high-definition (UHD) video, supporting 4K and 8K resolutions with a wider color gamut.
- BT.2100 extends BT.2020 with specifications for high dynamic range (HDR) video, introducing PQ and HLG transfer functions for greater brightness and contrast.

These standards are widely used in applications such as cameras, monitors, broadcast systems, and televisions.

2.2.3. About Model

The model converts RGB color space to YCbCr color space. Currently, model supports the below implementations as shown in the table.

Standard	8-bit	10-bit	12-bit	Full Swing	Studio Swing
BT.601	Yes	Yes	NA	Yes	Yes
BT.709	Yes	Yes	NA	Yes	Yes
BT.2020	NA	Yes	Yes	Yes	Yes

BT.2100	NA	Yes	Yes	Yes	Yes

The block diagram for color space conversion model can be represented as below:

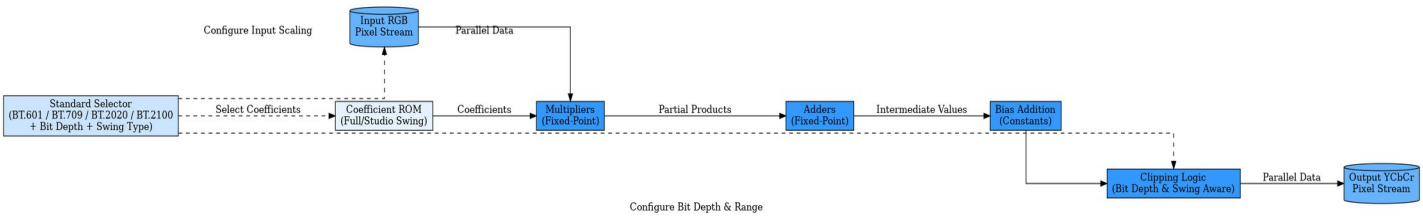

Illustration 4: Block diagram for Color Space Conversion design

1. Choose the standard first for color space conversion. Then depending on the input image, set the bit depth and then choose either full or studio swing implementation.
2. Model takes 3-input color RGB through a single channel. (ac_channel)
3. Uses Matrix Multiplication (Input RGB with coefficients) to compute YCbCr.

   The value of the coefficients is obtained by setting the precision of `std::cout` to 64 decimal places and printing the array elements with this precision. The code for generating the coefficients can be checked out in the toolkit example.
4. Bias Addition & Clipping: Adjusts Cb and Cr by adding 128 and ensures values are within valid range (Y: 16–235, Cb/Cr: 16–240) for 8 bit and adjusts Cb and Cr by adding 512 and ensures values are within valid range (Y: 64–940, Cb/Cr: 64–960) for 10 bit and similarly for 12 bit. (This is for studio swing)
5. Model writes the final converted luminance (Y) and chrominance (Cb, Cr) components into output channel YCbCr.

   For achieving low-latency high-throughput RTL implementation:
   - Top function is pipelined with II =1
   - A combinational ccore is applied on the multiplication loop which by default also fully unrolls the loop
   - Multiplication is implemented using adders

Detailed HLS implementation of the model can be found in library header file ac_rgb2ycbcr.h and ac_pixels.h in Catapult include directory.

2.2.4. Configuring the Model

The model is implemented using a class template. Template parameters can be used to configure the model for a specific HW implementation. Model also uses built-in asserts if incorrect configuration is used. The definition of the template parameters used in the library header file ac_rgb2ycbcr.h is given below:

Parameters	Description
PixIn_type	The R,G,B input values are concatenated into a single packed vector where datatype for each of R, G, B values is ac_int<8, false> or ac_int<10, false> or ac_int<12, false>. 8, 10 and 12 are the bit depths for the R, G, B values.
PixOut_type	The Y,Cb,Cr output values are concatenated into a single packed vector where datatype for each of Y, Cb, Cr values is ac_int<8, false> or ac_int<10, false> or ac_int<12, false>. 8, 10 and 12 are the bit depths for the Y, Cb, Cr values.
AcImgHeight	Maximum input image height
AcImgWidth	Maximum input image width
Q	Specifies the quantization mode with a default value of AC_TRN. It is used to quantize the output ac_fixed Y, Cb, Cr values to ac_int values.
FractBits	Precision for fractional part of the coefficients with default value of 18.
StudioSwing	Uses studio swing equation if set to true. By default it is false and uses the full swing equation
Standard	Specifies the standard to use. Default is ac_ipl::BT601<FractBits> which is the BT.601 standard.

Following assertions are built into the model:

For BT.601 standard:

Condition	Assert Type	Message
PixIn_type::base_type::width == 8 || PixIn_type::base_type::width == 10	Static	Both 8-bit and 10-bit depth encoding for BT.601 is supported.
AcImgHeight<= 720	Static	Maximum Height should be less than or equal to 720 as BT.601 standard is specifically defined for SDTV images.
AcImgWidth <= 576	Static	Maximum Width should be less than or equal to 576 as BT.601 standard is specifically defined for SDTV images.

For BT.709 standard:

Condition	Assert Type	Message
PixIn_type::base_type::width == 8 || PixIn_type::base_type::width == 10	Static	Both 8-bit and 10-bit depth encoding for BT.709 is supported.
AcImgHeight <= 1080	Static	Maximum Height should be less than or equal to 1080 as BT.709 standard is specifically defined for HDTV images.
AcImgWidth <= 1920	Static	Maximum Width should be less than or equal to 1920 as BT.709 standard is specifically defined for HDTV images.

For BT.2020 standard and BT.2100 standard:

Condition	Assert Type	Message
PixIn_type::base_type::width == 10 || PixIn_type::base_type::width == 12	Static	Both 10-bit and 12-bit depth encoding for BT.2020 and BT.2100 is supported.
AcImgHeight <= 7680	Static	Maximum Height should be less than or equal to 7680
AcImgWidth <= 4320	Static	Maximum Width should be less than or equal to 4320

The following asserts are defined for all standards.

Condition	Assert Type	Message
height <= *AcImgHeight*	Runtime	The height exceeds the maximum height.
width <= *AcImgWidth*	Runtime	The width exceeds the maximum width.

2.2.5. Development Flow Used

The accuracy of the hardware model implemented using HLS is obtained by comparing it to the software model implemented in the testbench. In the hardware model, the coefficients used are fixed point coefficients and in the software model, the coefficients that are used are floating point coefficients and are represented using the double datatype. The accuracy is determined using the error metrics mean relative error and mean square error.

The model was tested for different configurations for pre-HLS verification (which is basically for computing the accuracy as described above) and synthesis. The model was tested for a width value of 8,10 and 12 for the parameters *PixIn_type* and *PixOut_type* depending on which standard is used. Testing was also done for a maximum height and width for all the standards. Model asserts if the maximum height and width exceeds the values mentioned in the above tables for asserts. The Q parameter was tested for AC_TRN where the values are truncated towards -infinity and AC_RND where the values are rounded towards +infinity. The FractBits parameter was tested for values of 16, 18, 32, 48 which affect the accuracy of the hardware model. The model was also tested for a few input images for 8 bit where the images are available in the toolkit example and also for 10 bit and 12 bit images in .tiff format (Actually 16 bit images where 10 bit and 12 bit pixels are packed in 16 bit values). Then there are runtime asserts defined for the width and height of the input image which throw a runtime assert if they exceed the maximum width and height.

The library used in the toolkit example is the saed 32nm library. Another library used for High-Level Synthesis and RTL synthesis for the design is based on a 3 nm technology node with a worst-case process corner, operating at a voltage of 0.675V and a temperature of -40°C but this library is not provided with the toolkit example. The maximum frequency for the design using this 3nm library was determined by trying out different hardware optimizations like loop pipelining and unrolling, using combinational ccores, applying clustering, scheduling multi-cycle. Details about the maximum frequency exploration can be found in the section “PPA Exploration Results” below.

Model is developed using Catapult design and verification framework. Model is signed off for all configurations (mentioned earlier) and different Fmax, using following flows:

Flow Used	Goal	Criteria	Status
C++ Simulation	Coefficient width optimization	Relative Error (Y) = 0 RMS (Y) = 0	Pass
Catapult DA	Code review for QoR, Runtime	Zero errors in DA	Pass
C Design Checker (CDC)	Code lint for Synthesis-Simulation checks	Zero errors and warnings	Pass*
CCOV	Model Verification	Code Coverage =100%	Yes
CCOV	Model Verification	Cover all possible range of input and output values for functional coverage	Yes
Catapult Synthesis	Latency, throughput, Area & timing analysis	Pass Synthesis ideally with Latency and Throughput = 1	Pass
Industry Standard RTL Synthesis tool	Area timing sign-off	No timing violations Area (RTL synth tool) < Area (Cat)	Pass
Scverify	RTL & C++ functional equivalence signoff Latency and Throughput	No mismatches	Pass
UVM flow	RTL simulation match with predictor model	No mismatches	Pass
UVM flow	RTL Coverage	Coverage > 90%	Yes
Industry standard lint flow	RTL linting	Zero errors and warnings	Pass*

* warnings waived

2.2.6. PPA Exploration Results

The PPA exploration was conducted using the 3nm technology library for all implementations and configuration options, covering all standards (BT.601, BT.709, BT.2020, and BT.2100) and their corresponding RGB to YCbCr conversion architectures, including both full swing and studio swing equations across 8-bit, 10-bit, and 12-bit color depths. The PPA results are shown below.

The PPA results for one configuration for BT.2100 standard is given in below section:

Max frequency for 10-bit Full Swing BT.2100:

The PPA results for 333MHz, 667MHz, 1000MHz and 1333 MHz (roughly at 25%, 50%, 75%, 100% of the max frequency) can be found in the below graphs. Here, 1333 MHz is the maximum frequency observed in Catapult while the maximum frequency observed in the RTL synthesis tool is 1666 MHz.

The latency and throughput results can be summarized in the Illustration 5: Latency and Throughput vs Frequency.The area and slack results post RTL synthesis can be summarized in the Illustration 6: Area and Slack vs Frequency

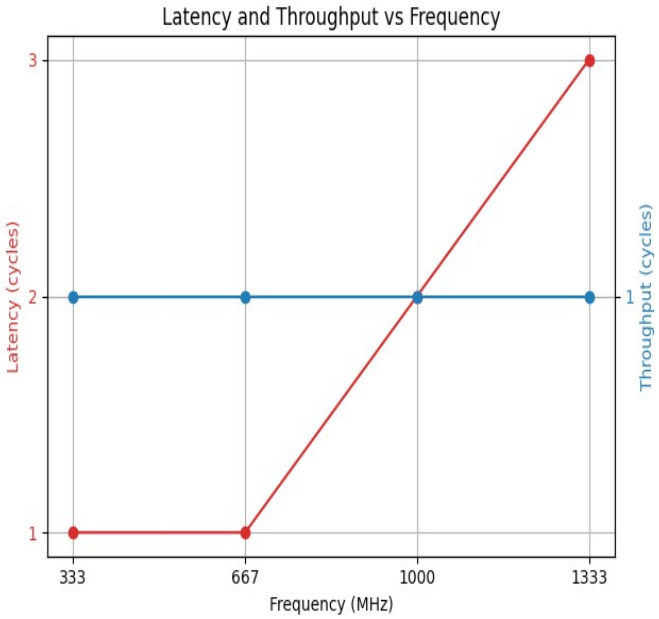

Illustration 5: Latency and Throughput vs Frequency

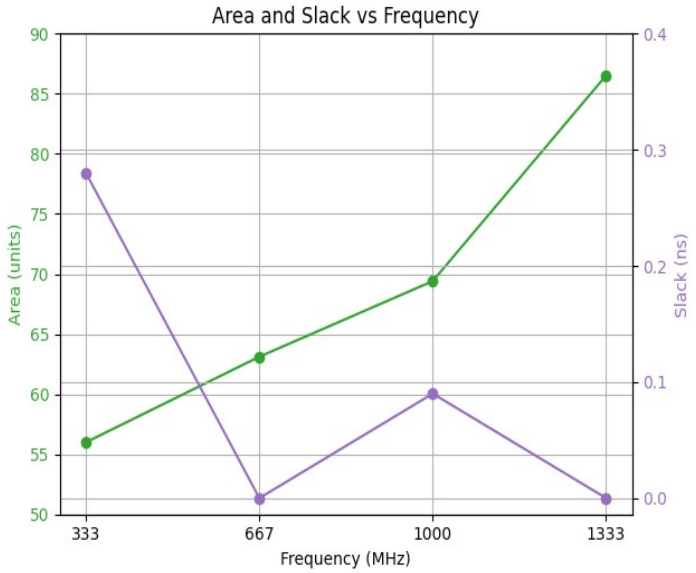

Illustration 6: Area and Slack vs Frequency

With multi-cycle scheduling, the maximum frequency can probably be increased.

2.2.7. Toolkit Example

Consider the hardware color space conversion implementation with the following characteristics:

- Color-depth of 10.
- Studio swing implementation.
- Maximum image width/height of 7680 and 4320 respectively.
- BT.2100 standard

The following are the steps the toolkit takes to initialize an object of the *ac_rgb2ycbcr* class with the above mentioned characteristics and call the *run(..)* function. This is done using a reference design file ac_rgb2ycbcr_BT2100.h, parameters_BT2100.h and a testbench file ac_rgb2ycbcr_tb_BT2100.cpp:

1. In ac_rgb2ycbcr_BT2100.h, include <ac_ipl/ac_rgb2ycbcr.h> and parameters_BT2100.h.
2. Define all the parameters in parameters_BT2100.h.

```
constexpr int CDEPTH = 10;
constexpr int AC_IMG_HEIGHT = 4320;
constexpr int AC_IMG_WIDTH = 7680;

constexpr bool STUDIO_SWING = false;
```

```
//Input and output pixel type
typedef ac_ipl::RGB_pv<ac_int<CDEPTH, false> > PixIn_type;
typedef ac_ipl::YCbCr_pv<ac_int<CDEPTH, false> > PixOut_type;
```

3. Instantiate *ac_channel*s with the input and output types defined above in ac_rgb2ycbcr_tb_BT2100.cpp.

```
ac_channel<PixIn_type>  din_ch;
ac_channel<PixOut_type> dout_ch;
```

4. Initialize dimension variables.

```
int height = 1080;
int width  = 1920;
```

5. Write all input values to the din_ch *ac_channel*. Instantiate an object of the top level *ac_rgb2ycbcr_BT2100* class and call the top level member function with ac_channels, dimension variables mentioned above.

```
ac_rgb2ycbcr_BT2100 ac_rgb2ycbcr_BT2100_obj;
ac_rgb2ycbcr_BT2100_obj.run(din_ch, dout_ch, height, width);
```

6. Instantiate the base level class ac_rgb2ycbcr in ac_rgb2ycbcr_BT2100.h and call the cvtColor function.

```
ac_csc::ac_rgb2ycbcr<PixIn_type, PixOut_type, AC_IMG_HEIGHT, AC_IMG_WIDTH, AC_RND, 18,
STUDIO_SWING> ac_rgb2ycbcr_obj;
 ac_rgb2ycbcr_obj.cvtColor(din_ch, dout_ch, height, width);
```

The toolkit example can be accessed from Help -> Examples -> IPL(ac_ipl) -> Color Space Conversion in the Catapult GUI.

Compiling the source files for BT.2020 and BT.2100 standard

In order to compile the source files for BT.2020 and BT.2100 standard, the library libtiff-devel needs to be installed. It can be installed using “sudo dnf install libtiff-devel” in a linux terminal. This is for processing tiff images.

Executing the flows

To execute the flows, README.txt files are provided as reference for most of the flows. The README.txt file lists the make commands to execute the flows.

1. For C code coverage using CCOV and RTL coverage using the UVM flow, README.txt files are provided for each of the standards under the directories BT601, BT709, BT2020, BT2100 under the C_and_RTL_coverage directory in the toolkit example. There is another README.txt file provided in the C_and_RTL_coverage directory which merges coverage to produce a combined coverage report.

2. For Functional coverage using CCOV, README.txt files are provided for each of the standards under the directories BT601, BT709, BT2020, BT2100 under the Functional_coverage directory in the toolkit example.

3. Tcl files are provided in the toolkit example for running the Catapult flow for each of the standards.

4. For CFormal flows, there are tcl files provided in each of the BT601, BT709, BT2020, BT2100 directories under the CFormal directory.

2.2.8. RTL simulation

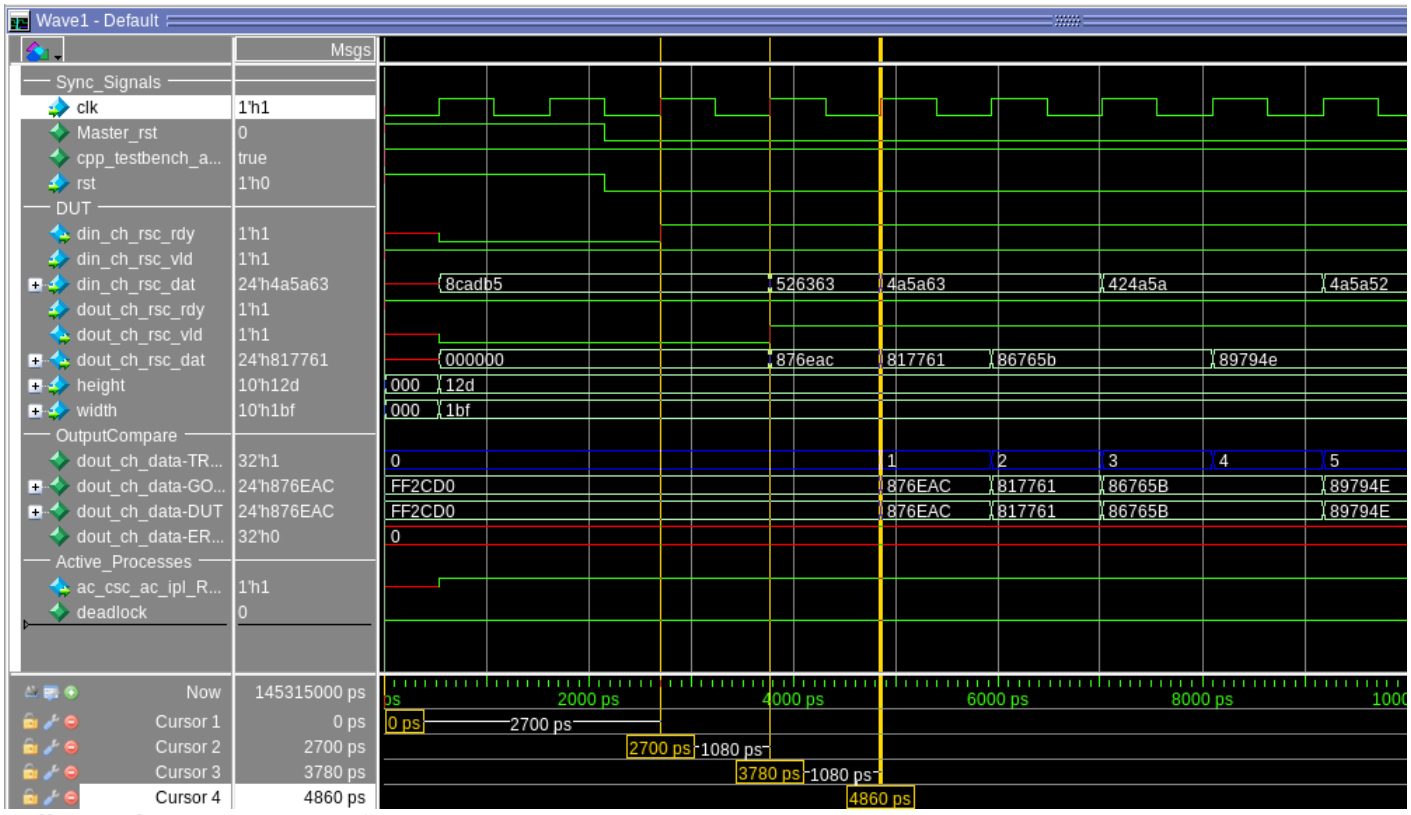

Illustration 7: RTL waveform

The waveform shown above is obtained from the RTL simulation for 10 bit full swing equation for BT.2100 standard. As observed, when both din_ch_rsc_rdy and din_ch_rsc_vld are asserted high (1), the design

produces the first valid output on dout_ch_rsc_dat after one clock cycle, indicating a latency of 1. Furthermore, the design continues to generate subsequent outputs in every clock cycle, demonstrating a throughput of 1.

Similar RTL simulation results can be observed for the other standards.

For more technical details on this HLS Design IP, contact Catapult Support.

2.3. Color Temperature Correction (ac_ctc)

The *ac_ctc* library provides a hardware implementation for a color temperature correction algorithm. It can be pipelined with an II of 1 and works on color (*RGB_1PPC)* inputs.

2.3.1. Implementation Details and Limitations

The input pixels to and output pixels from the design have the same color-depth and are streamed to the design via ac_channels. The design also accepts three direct inputs, two of which are dimension inputs (i.e. image width and height), with the third input specifying the color temperature that the user wants the image corrected to, i.e. the target color temperature. After obtaining the white point estimate for both the target color temperature and the frame, the design calculates the ratio between the two and uses that to change the value of each pixel to suit the target color temperature. The white point for both target and frame are considered in terms of RGB values.

As will be explained later, the design provides the user the option of dynamically adjusting for changes in white point from one frame to another. Hence, this design can be used for video inputs.

Target White Point Estimation

In order to determine the white point associated with the target color temperature, the design uses hard-coded LUTs which store white point values ranging from 1000 K to 13700 K in increments of 100 K. Hence, the temperature equivalents of the values store in the LUT are multiples of 100 K. If the target color temperature supplied by the user is not a multiple of 100 K, then the target white point will be that associated with the closest multiple of 100 K that is lesser than the user-supplied target. For instance, if the user-supplied target color temperature is 1560 K, the design uses the LUT values for 1500 K.

The LUT values are printed out through a separate LUT generator file (ac_ctc_lutgen.cpp). The LUT generator program uses the algorithm described by [1] to generate an RGB value for a color temperature.

Frame White Point Estimation

Depending on the template parameters supplied to the design, the design will either determine white point values through (a) user-supplied values or (b) pixel values of the previous frame processed by the design. Both methods have their advantages and disadvantages.

(a) uses values supplied at compile time, and hence using a constant divider for ratio calculation would suffice for this variation. This in turn leads to a reduction in resources used. However, (a) cannot adjust to dynamic changes in the white point of the video from one frame to another and places the burden of calculating the white point of the frame on the user. (b) determines the frame white point following the algorithm used by the color temperature correction block in [2], where the RGB values of the pixel with the

maximum RGB sum in the previous frame are considered representative of the current frame's white point. (b) can adjust for dynamic changes in a video's white point, and also relieves the user of the burden of calculating the white point themselves. However, (b) necessitates the usage of a hardware divider, as the denominator is no longer a constant value.

It should be noted that method (b) starts off by assuming that the first frame's white point has all RGB values set to 255. The divider used by (b) is that provided by the ac_div() function in the ac_math library.

Refer to Model Parameters for an explanation on how to use template parameters to switch between options (a) and (b).

Catapult Architectural Exploration Options

All the loops in the design, including the main loop, are pipelined with an II of 1, as shown in the Architectural Constraints Editor Screenshot in Illustration 8: Loop Options.

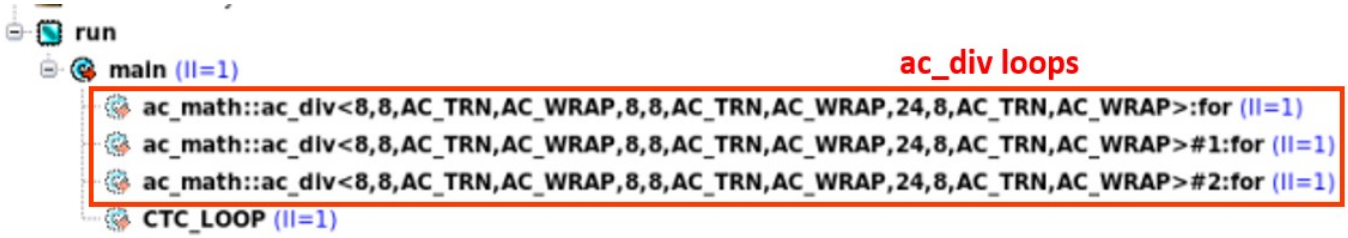

Illustration 8: Loop Options

The user can adjust the initiation interval in the constraints editor. Since CTC_LOOP contains IO accesses, it cannot be fully unrolled. While the ac_div loops can be fully unrolled, doing so is not recommended as it results in a negligible decrease in latency while also significantly increasing area of the design. In other words, the disadvantage of unrolling the ac_div loops significantly outweighs the advantage of doing so.

The LUTs for R, G and B values can be mapped to register banks or ROMs, depending on user requirements. Refer to Illustration 9: Constant Array Options for a view of the LUTs in the Constraints Editor.

Illustration 9: Constant Array Options

Limitations

- For target temperature inputs that fall in between LUT values, the design does not use interpolation for simplicity. Instead, it maps the input to the closest LUT value provided, as explained in Target White Point Estimation.
- The design cannot extrapolate for target color temperatures that lie beyond the range provided by LUT values. An AC_ASSERT is triggered at run-tume during C simulation if this condition is violated.

- Input and output types must both be *RGB_1PPC* datatypes of the same color depth.
- Template arguments for the user-provided frame white point must all be zero or non-zero. Mixing of zero and non-zero values is not supported so as to avoid division with a denominator of zero. Failure to follow this condition will trigger a static_assert in the ac_ctc class constructor.

Final Output

The result of the color correction design on a sample input frame is shown in Illustration 10. The output is corrected to a color temperature of 4000 K.

Illustration 10: CTC input/output. Input is on the left, output on the right

2.3.2. Model Parameters

The CTC library is implemented as a class template named *ac_ctc*. A public member function of the class, called run(..), acts as the top-level design block.

Given below is a snippet of the *ac_ctc* class.

```
template<unsigned CDEPTH, unsigned W_MAX, unsigned H_MAX, unsigned TEMP_MAX,
         unsigned R_WP = 0, unsigned G_WP = 0, unsigned B_WP = 0>
class ac_ctc
{
  // Code
  typedef ac_ipl::RGB_1PPC<CDEPTH> IO_TYPE;
  typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;
  typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType;
  typedef ac_int<ac::nbits<TEMP_MAX>::val, false> tempInType;

#pragma hls_pipeline_init_interval 1
#pragma hls_design interface
```

```
  void CCS_BLOCK(run) (
    ac_channel<IO_TYPE> &streamIn,  // Pixel input stream
    ac_channel<IO_TYPE> &streamOut, // Pixel output stream
    const widthInType   widthIn,    // Input width
    const heightInType  heightIn,   // Input height
    const tempInType    tempIn      // Input color temperature
  )
  // Code
};
```

The definition of the template parameters shown in the snippet is given below:

Parameters	Description
CDEPTH	Color depth of input and output.
W_MAX, H_MAX	Maximum input image width and height, respectively.
TEMP_MAX	Maximum target color temperature.
Following parameters are optional:	
R_WP, G_WP, B_WP	User-provided white point values for the input frame.

If R_WP, G_WP and B_WP are all zero, the design goes with option (b) explained in Frame White Point Estimation. However, if the user gives non-zero values for these template parameters, the design goes with option (a) instead. Since the default template parameter values are all zero, the design goes with option (b) by default.

2.3.3. Calling the Top-level Function

Let us consider a CTC algorithm implementation with the following static characteristics:

- Input and output color-depth of 8.
- Maximum image width/height of 1024.
- Maximum target color temperature of 13700 K.
- Default option for frame white point estimation is used.

The following are the steps to initialize an object of the *ac_ctc* class with the above mentioned static characteristics:

1. Include <ac_ctc.h>
2. Define template parameters and data-types in the CCS_MAIN() function.

```
enum { CDEPTH = 8, W_MAX = 1024, H_MAX = 1024, TEMP_MAX = 13700 };
typedef ac_ipl::RGB_1PPC<CDEPTH> IO_TYPE;
```

```
typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;
typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType;
typedef ac_int<ac::nbits<TEMP_MAX>::val, false> tempInType;
```

3. Instantiate *ac_channel*s with the IO type defined above in the CCS_MAIN() function.

```
ac_channel<IO_TYPE>  streamIn, streamOut;
```

4. Initialize dimension and temperature input variables.

```
 widthInType  widthIn  = 512; // Input Frame Width
 heightInType heightIn = 384; // Input Frame Height
 tempInType tempIn = 4000; // Target color temperature
```

5. Write all input values to the streamIn *ac_channel*. Instantiate an object of the *ac_ctc* class with the above static characteristics and call the top level member function with the ac_channels and dimension/temperature variables mentioned above.

```
ac_ctc<CDEPTH, W_MAX, H_MAX, TEMP_MAX> ctcObj;
ctcObj.run(streamIn, streamOut, widthIn, heightIn, tempIn);
```

2.3.4. References

[1] Helland, T. (2012, September 18). *How to Convert Temperature (K) to RGB: Algorithm and Sample Code*. Retrieved April 8, 2020, from https://tannerhelland.com/2012/09/18/convert-temperature-rgb-algorithm-code.html

[2] Xilinx. (2007, April 19). *Digital Display Panel Reference Design* [PDF file]. Retrieved April 8, 2020, from https://www.xilinx.com/support/documentation/application_notes/xapp928.pdf

2.4. Floyd-Steinberg Dithering Algorithm (ac_dither)

The ac_dither library provides a hardware implementation for the Floyd-Steinberg dithering algorithm. It can be pipelined with an II of 1 and works on greyscale (unsigned *ac_int)* as well as color (*RGB_1PPC)* inputs.

2.4.1. Implementation Details

- For greyscale inputs/outputs, the library quantizes a single greyscale input component to the output color-depth. For color inputs/outputs, the library considers each of the three color components as separate greyscale inputs, quantizing each one appropriately.

- In accordance with the Floyd-Steinberg dithering algorithm, the design diffuses the quantization error over not only the pixel that immediately comes next, but also over the next row of pixel values. The partial sums of diffused quantization errors for the next row are stored in a linebuffer.

- Dimension inputs are mapped to direct inputs.

Catapult Architectural Exploration Options

The row and column loops, as well as the top-level run() function are all pipelined with an II of 1. The internal MAC_LOOP is fully unrolled, as shown in the Architectural Constraints Editor Screenshot in Illustration 11. The user can adjust the initiation interval and/or control the unrolling of the MAC_LOOP through the constraints editor. Since COL_LOOP contains IO and memory accesses, neither it nor ROW_LOOP or the main loop can be fully unrolled.

Illustration 11: Loop Options

If the linebuffer size falls above the memory threshold, it will be mapped to memory. If the user decides to use a singleport memory while pipelining with an II of 1 and preserving the design throughput, they can toggle the *use_sp* template parameter. Refer to Model Parameters and Calling the Top-level Function for more details on the *use_sp* template parameter. The number of linebuffer memories will change depending on whether the inputs are greyscale or not. Refer to Illustration 12 for the view of the linebuffer arrays in the Architectural Constraints Editor for an arbitrary use case (Max. image width = 1024, width of partial sum data = 64 bits). If a singleport memory is used, the number of memory locations will be halved while doubling the word width.

Limitations

- If using single-port memories, the design can only work with even image widths. An *AC_ASSERT* is triggered at run-time during simulation-only runs if this condition is violated.
- While the input and output color-depths are flexible and can be changed at compile-time, the output color-depth can never exceed input color depth. A *static_assert* is triggered at compile time if this condition is violated.
- Input/output types must both be either unsigned *ac_int* or *RGB_1PPC* datatypes. Another *static_assert* is triggered if this condition is violated.

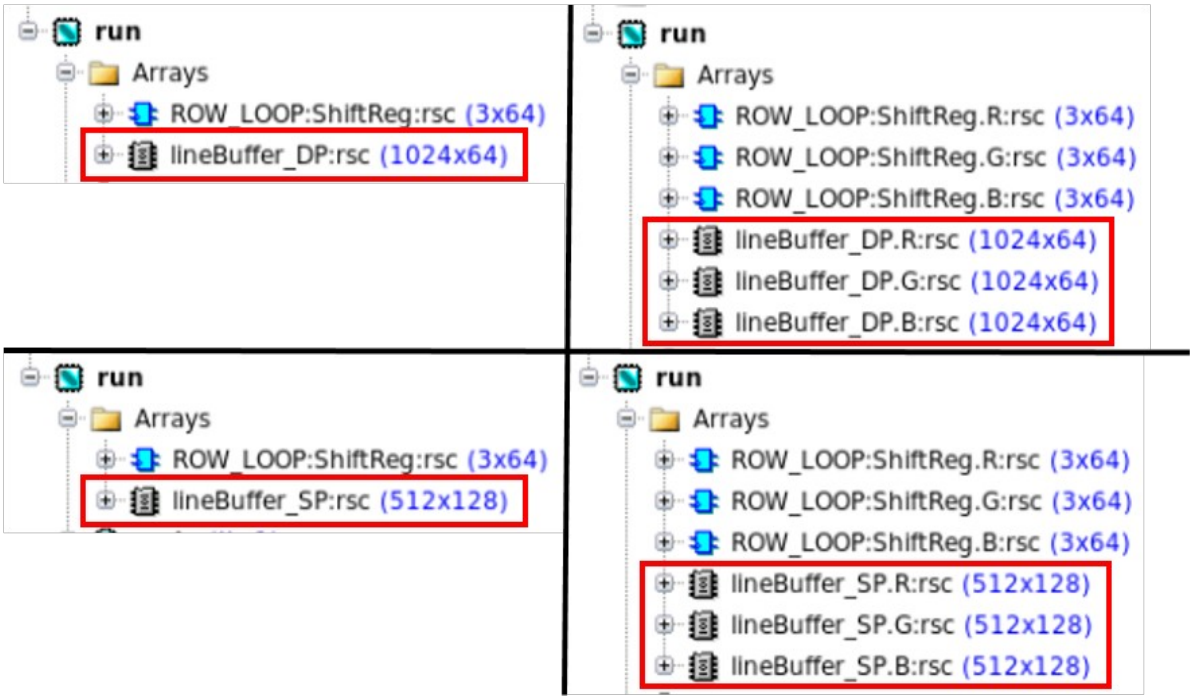

Illustration 12: Array Options. Clockwise from top right- Dualport and color, dualport and greyscale, singleport and greyscale, singleport and color.

Final Output

The result of the dithering library on a sample input image is shown in Illustration 13.

Illustration 13: Input/Output for Dithering Library. Input is on left, output on right. Input is a 24 bpp color image, the output is 3 bpp.

2.4.1. Model Parameters

The dithering library is implemented as a class template named *ac_dither*. A public member function of the class, called run(..) acts as the top-level design block.

Given below is a snippet of the *ac_dither* class.

```
template <class IN_TYPE, class OUT_TYPE,
          unsigned W_MAX, unsigned H_MAX,
          bool use_sp = false,
          int pSumW = 32, int pSumI = 16,
          ac_q_mode pSumQ = AC_TRN, ac_o_mode pSumO = AC_SAT>
class ac_dither
{
  // Code
  typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;
  typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType;

#pragma hls_pipeline_init_interval 1
#pragma hls_design interface
  void CCS_BLOCK(run) (
    ac_channel<IN_TYPE>  &streamIn,  // Pixel input stream
    ac_channel<OUT_TYPE> &streamOut, // Pixel output stream
    const widthInType    widthIn,    // Input width
    const heightInType   heightIn    // Input height
  )
  // Code
}
```

The definition of the template parameters shown in the snippet is given below:

Parameters	Description
IN_TYPE	Input type.
OUT_TYPE	Output Type.
W_MAX, H_MAX	Maximum input image width and height, respectively.
Following parameters are optional:	
use_sp	Set to *true* to enable the usage of single-port memory with an II of 1.
pSumW, pSumI	Bitwidth and integer width allocated for partial sums of diffused quantization errors.
pSumQ, pSumO	Rounding and saturation modes for the partial sums.

The design uses dual-port memories by default. The type used to store the sum of diffused quantization errors is a signed, *ac_fixed* datatype. As can be seen in the code snippet above, the presence of default template values ensures that it is defined as *ac_fixed<32, 16, true, AC_TRN, AC_SAT>* by default.

2.4.2. Calling the Top-level Function

Let us consider a dithering algorithm implementation with the following static characteristics:

- Input color-depth of 8.
- Output color-depth of 1.
- Maximum image width/height of 1024.
- Use singleport memories.
- Intermediate type of *ac_fixed<64, 32, true, AC_TRN, AC_WRAP>*

The following are the steps to initialize an object of the *ac_dither* class with the above static characteristics:

1. Include <ac_dither.h>
2. Define data-type for input/output, as well as image dimensions in the CCS_MAIN() function..

```
enum {IN_CDEPTH = 8, OUT_CDEPTH = 1, W_MAX = 1024, H_MAX = 1024};
typedef ac_ipl::RGB_1PPC<IN_CDEPTH>  IN_TYPE;  // Input Pixel Type
typedef ac_ipl::RGB_1PPC<OUT_CDEPTH> OUT_TYPE; // Output Pixel Type
typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;  // Input width
type
typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType; // Input height
type
```

3. Instantiate *ac_channel*s with the I/O types defined above in the CCS_MAIN() function.

```
ac_channel<IN_TYPE>  streamIn;
ac_channel<OUT_TYPE> streamOut;
```

4. Initialize dimension variables.

```
 widthInType  widthIn  = 512; // Input Image Width
 heightInType heightIn = 384; // Input Image Height
```

5. Write all input values to the streamIn *ac_channel*. Instantiate an object of the *ac_dither* class with the above static characteristics and call the top level member function with the ac_channels and dimension variables mentioned above.

```
ac_dither<IN_CDEPTH,OUT_CDEPTH,W_MAX,H_MAX,64,32,AC_TRN,AC_WRAP> DitherObj;
DitherObj.run(streamIn, streamOut, widthIn, heightIn);
```

2.5. Gaussian Image Pyramid (ac_gaussian_pyr)

The ac_gaussian_pyr library provides a hardware implementation for the Gaussian Image Pyramid algorithm. By default, it is pipelined with an II of 1 and works on greyscale (unsigned *ac_int)* as well as color (*RGB_imd)* inputs.

2.5.1. Implementation Details and Limitations

Functional Overview

The Gaussian Image Pyramid is a multi-resolution representation of an input image, with the image subjected to a lowpass/smoothing operation via Gaussian filtering. The pyramid output comprises of a pre-determined number of levels. With each level, the filtered output is downsampled by 2 across both the rows and columns, leading to a halving of image height and width for each level, respectively. A 5x5 kernel is used at all levels of the pyramid, in line with that used in the Matlab Gaussian Pyramid library. Refer to [1] for more details. The kernel values are displayed in Illustration 14.

0.00390625	0.015625	0.0234375	0.015625	0.00390625
0.015625	0.0625	0.09375	0.0625	0.015625
0.0234375	0.09375	0.140625	0.09375	0.0234375
0.015625	0.0625	0.09375	0.0625	0.015625
0.00390625	0.015625	0.0234375	0.015625	0.00390625

Illustration 14: Gaussian Kernel

Block Diagram

Refer to Illustration 15 for a generic Gaussian Pyramid design with N levels, N > 2. The design follows a scheme similar to that described in [2], with each level being mapped to a separate module/block with a private memory. The outputs are downsampled by 2 across rows and columns, which results in a halving of image width and height as we move from one level to the next. The design supports 1 <= N <= 10.

The pixel input from the external producer is received via the stream_in ac_channel and fed to the Level 1 block, and is stored in four ac_window linebuffers which are as big as the maximum possible image width. The linebuffers in turn feed the registered array for the window itself, which is of size 5x5 corresponding to the 5x5 gaussian kernel. The kernel and window values are multiplied, accumulated, downsampled and passed to both the external system and the next level (the latter happens via an interconnect channel).

Because of the reduction in image dimensions as we move from one level to another, the next level only needs W_MAX/2 elements for each of its linebuffers, the level after that needs W_MAX/4 elements, and so on. In other words, the number of linebuffer elements halves as we move from one level to the next.

Level 1 feeds level 2 via the stream_inter_0 channel, level 2 feeds level 3 via the stream_inter_1 channel, and so on and so forth till we get to the Nth level, whose output is not connected to an interconnect channel. The 5x5 MAC operation happens for each level.

The external system receives the output via an array of ac_channels (stream_out), such that the nth level feeds stream_out[n-1], where 1 <= n <= N.

ac_gaussian_pyr is the top-level wrapper class which instantiates all the levels as sub-blocks, and all the interconnect ac_channels as static data members.

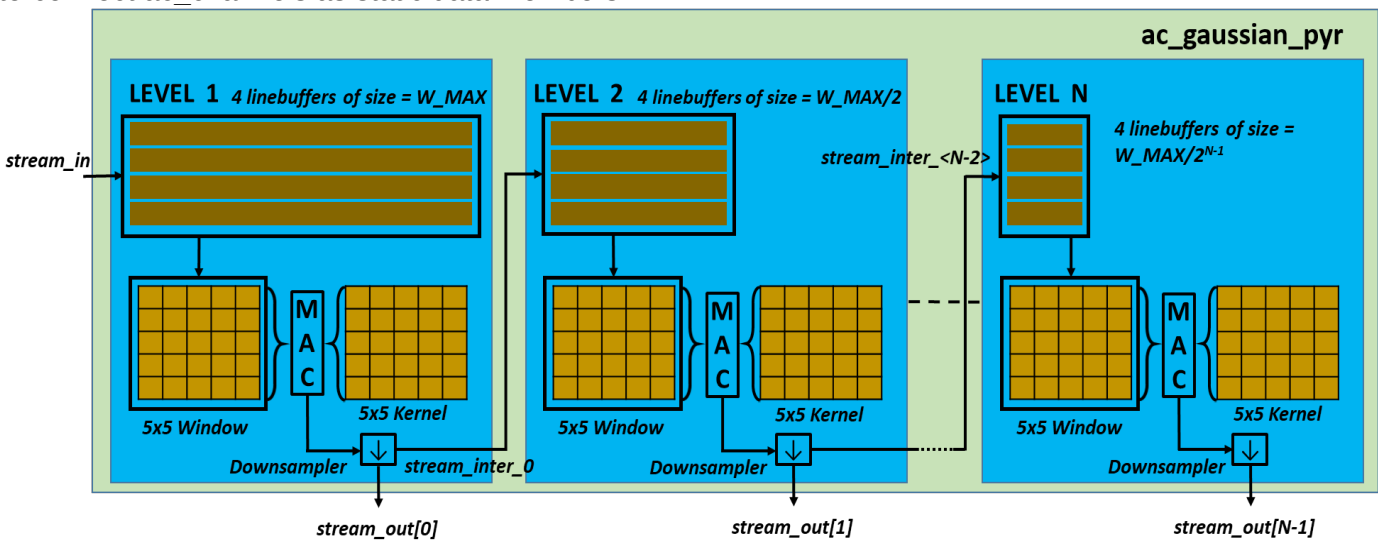

Illustration 15: Block Diagram of Gaussian Pyramid

Catapult Architectural Exploration Options

As mentioned earlier, the design is pipelined with an II of 1 by default. For color implementations, each linebuffer is divided into three separate memories, one for each color component. For grayscale implementations, we only need one memory per linebuffer.

The linebuffer mapping for Level 1 of an example dualport color implementation with three levels, as viewed in the architectural constraints editor, can be seen in Illustration 16. In this example, the linebuffer size (1024) is large enough that it falls above the memory threshold. Hence, all linebuffer arrays are mapped to memories. There are a total of 12 (4 linebuffers x 3 memories per linebuffer) memories per level for color implementations, and 4 memories per level for grayscale implementations, provided that the arrays are large enough to fall above the memory threshold. If not, they'll be mapped to registers instead, which is a possibility for designs with a large number of levels (remember that the image width gets halved at every level due to downsampling).

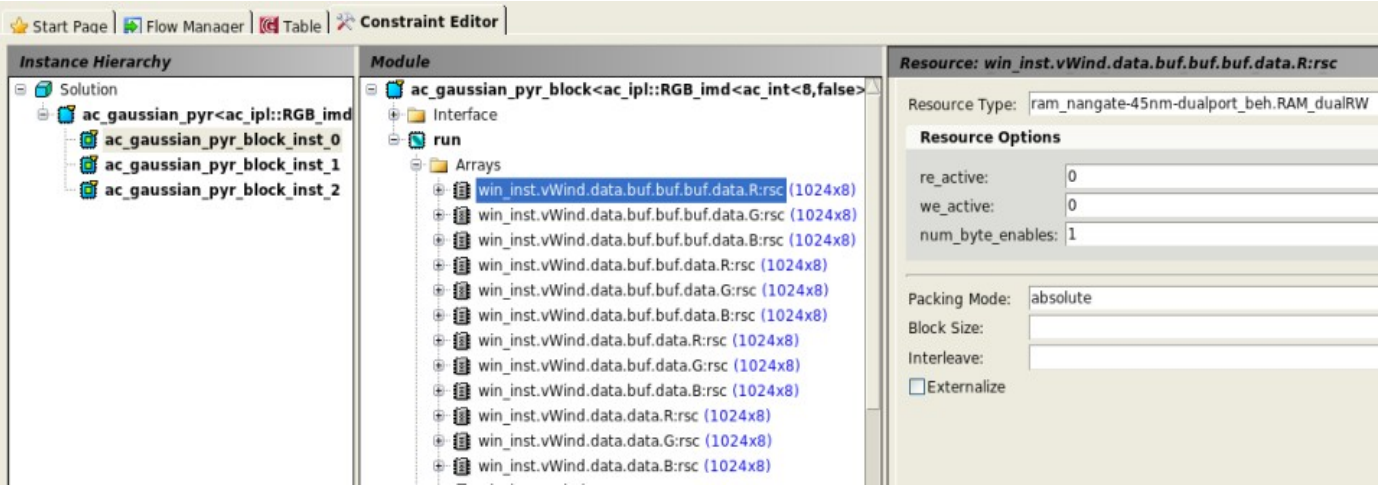

Illustration 16: Array Options

If the user wishes to maintain pipelining with an II of 1 and switch between singleport and dualport memory implementations, they can do so by changing the USE_SP template parameter. More details on doing so are given in the Model Parameters section.

The loop mapping for the Level 1 sub-block, for the example design shown above, can be seen in Illustration 17. Since the column processing loop (COL_LOOP) and, by extension, the row processing loop (ROW_LOOP) have memory and IO accesses, they cannot be fully unrolled and are instead pipelined with an II of 1, as is the design as a whole. All the other loops are fully unrolled to allow for maximum throughput and parallelism. This unrolling is possible due to the kernel and window sizes being 5x5, and the corresponding window array being small enough to be mapped to registers, hence allowing for parallel array accesses.

If the user intends on increasing the kernel/window size and maintaining full unrolling, they must make sure that the size of the kernel and window arrays falls below the memory threshold, failing which they can:

- Explicitly map the window arrays to register banks (provided that they’re below the register threshold).
- Increase the memory threshold to allow for register mapping.

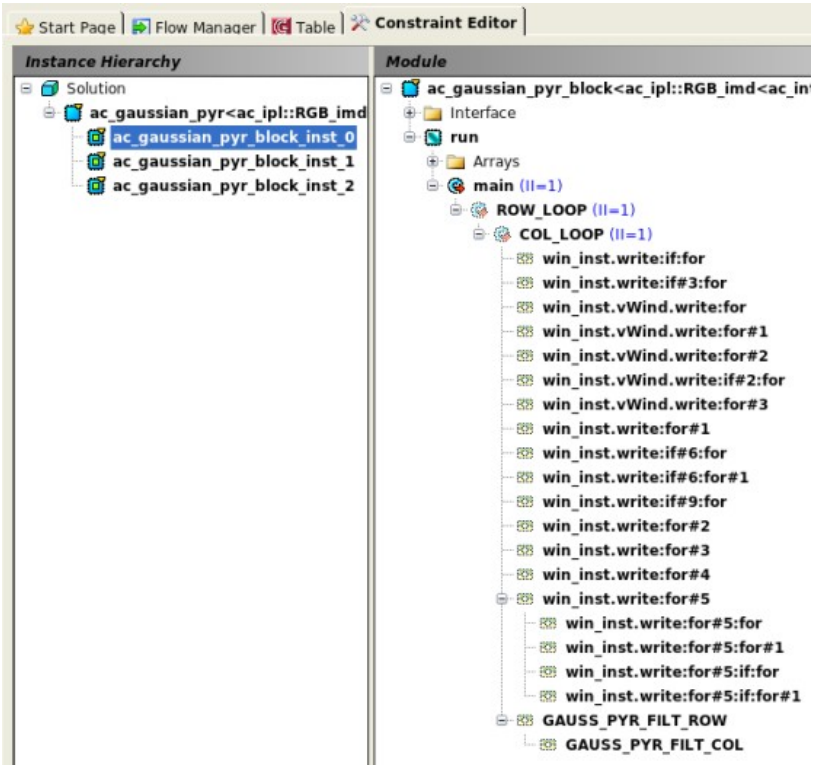

Illustration 17: Loop Options

Since the dimension inputs (w_in and h_in) are supplied to every level sub-block, they are mapped to DirectInputs. Failure to do so will result in catapult throwing an error like this at the “go architect” stage:

```
# Error: ac_gaussian_pyr.h(192): Resource '/
ac_gaussian_pyr<ac_ipl::RGB_imd<ac_int<8,false>>,ac_ipl::RGB_imd<ac_fixed<18,8,
false,AC_TRN,AC_WRAP>>,1024,1024,3,16,false>/w_in:rsc' with variable connected
to multiple sub-blocks not mapped to '[DirectInput]' (MEM-71)
```

While this is the error transcript for the width input (w_in) interface of the dualport color implementation specifically, a similar error is displayed for the other implementations if the mapping isn’t correct.

Port Descriptions

Refer to Illustration 18 for a view of the ports/pins of an example three-level Gaussian Pyramid design, as seen in Catapult Design Analyzer. The description of each of the ports is given in the table below:

Pin	Direction	Description
clk	IN	Clock signal.
rst	IN	Reset.
w_in, h_in	IN	DirectInputs for width and height, respectively.
stream_in:rsc.dat	IN	Input data stream.
stream_in:rsc.rdy	OUT	Input ready signal, goes high for a cycle to acknowledge data read.

stream_in:rsc.vld	IN	Input valid signal, goes high when data is valid on clock edge.
steam_out(n):rsc.dat	OUT	Output data stream for (n+1)th level.
steam_out(n):rsc.rdy	IN	Driven high when external consumer requests data from level (n+1).
steam_out(n):rsc.vld	OUT	Driven high by design when data is ready to be read from level (n+1).

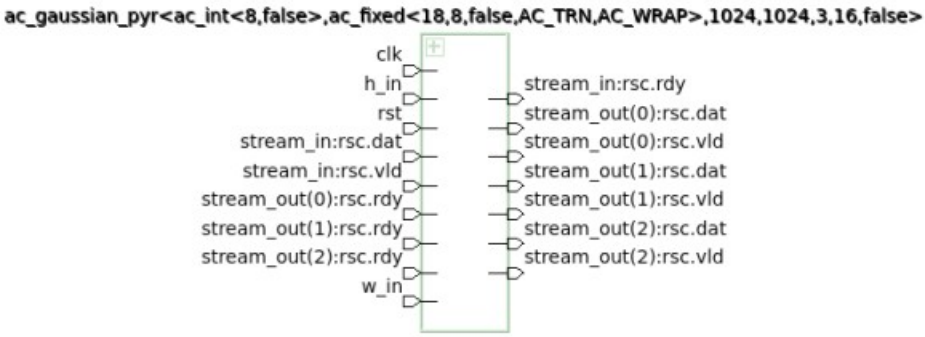

Illustration 18: Pin Diagram for ac_gaussian_pyr

Limitations

- As mentioned earlier, the dimension inputs can only be mapped to DirectInputs, as they supply multiple sub-blocks, and as such, the user must hold them stable while the design has data in the pipeline.
- Only ac_int or RGB_imd inputs that satisfy the criteria described in Model Parameters can be used.
- The image input for every level must be over 5 pixels high and 5 pixels wide, in order to ensure correct functioning with the 5x5 ac_window. This is especially important for a large number of levels, since the image input dimensions get halved for each level and might hence fall below this threshold.
- The IP does not support more than ten levels.

Final Output

Consider an example input color image, as seen in Illustration 19. We can pass this image through both a greyscale and color implementation (note that the color pixels have to be converted to greyscale pixels externally, before passing them greyscale implementation). The results of doing so can be seen in Illustration 19.

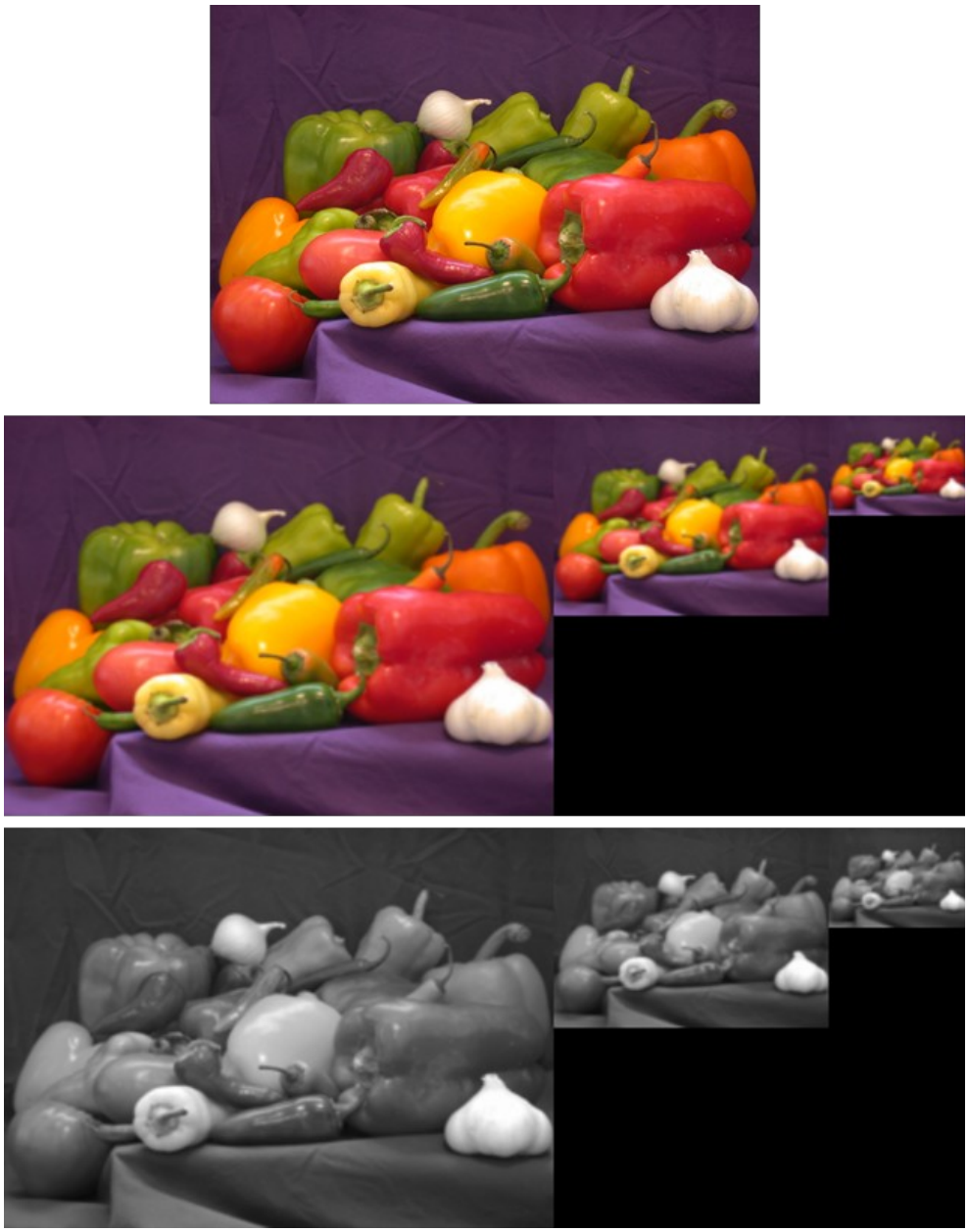

Illustration 19: Input and Output Images. From top to bottom: Input color image, output color image, output grayscale image.

2.5.2. Model Parameters

The design is coded as ac_gaussian_pyr.h, a synthesizable C++ header file. The header file contains a class template named ac_gaussian_pyr, and each of the level blocks are instantiated by this class as objects of the ac_gaussian_pyr_block class. The interconnect channels are instantiated as class data members, so as to follow Catapult coding requirements. A public member function of the ac_gaussian_pyr class, called run(..), serves as the top-level design block.

As mentioned earlier, windowing is done via an ac_window library, i.e. ac_window_2d_flag_flush_support, which is chosen so as to allow the use of straightforward, easy-to-read code that also incorporates flushing.

The following is a snippet of the ac_gaussian_pyr class:

```
#pragma hls_design top
template <class IN_TYPE, class OUT_TYPE, int W_MAX, int H_MAX, int N_LEVELS,
int MAX_FRAC_BITS = 16, bool USE_SP = true>
class ac_gaussian_pyr {
public:
  // Code
  typedef ac_int<ac::nbits<W_MAX>::val, false> W_IN_TYPE;
  typedef ac_int<ac::nbits<H_MAX>::val, false> H_IN_TYPE;
  #pragma hls_pipeline_init_interval 1
  #pragma hls_design interface
  void CCS_BLOCK(run) (
    ac_channel<IN_TYPE>  &stream_in,
    ac_channel<OUT_TYPE> stream_out[N_LEVELS],
    const W_IN_TYPE w_in, // Dimension Input: width.
    const H_IN_TYPE h_in  // Dimension Input: height.
  )
  // Code
};
```

The definition of the template parameters shown in the snippet is given below

Parameters	Description
IN_TYPE	Input type.
OUT_TYPE	Output Type.
W_MAX, H_MAX	Maximum input image width and height, respectively.
N_LEVELS	Number of levels.
Following parameters are optional:	
MAX_FRAC_BITS	Maximum number of fractional bits used for interconnect IO types. 16 by default.
USE_SP	Enables the usage of single-port memories with an II of 1 if set to true (default).

The design only allows the following IO datatype combinations:

- IN_TYPE: ac_int<I, false>, OUT_TYPE: ac_fixed<W, I, false, Q, O>
- IN_TYPE: RGB_imd<ac_int<I, false> >, OUT_TYPE: RGB_imd<ac_fixed<W, I, false, Q, O> >

Note that the number of integer bits for the integer and fixed point types must be the same.

Since the gaussian kernel values have eight fractional bits, we must add eight fractional bits to the interconnect output of every level to ensure full precision after the MAC operation. For a generic pyramid design, the number of fractional bits accumulates across levels, i.e. Level 1 interconnect output requires 8 fractional bits for full precision, Level 2 requires 16, Level 3 requires 24, etc. The MAX_FRAC_BITS template parameter places a cap on this growth. For instance, with the default value of 16 bits, the number of fractional bits won’t grow after Level 2, and will stay set at 16 for every level after that.

2.5.3. Calling the Top-Level Function

Let us consider a dualport, RGB_imd implementation with the following static characteristics:

- IN_TYPE: RGB_imd<ac_int<8, false> >
- OUT_TYPE: RGB_imd<ac_fixed<18, 8, false> >
- W_MAX/H_MAX = 1024
- N_LEVELS = 3
- MAX_FRAC_BITS = 16 (same as default)
- USE_SP = false

The following are the steps used to initialize an object of the ac_gaussian_pyr class with these characteristics and call the top-level run() function:

The following are the steps used to initialize an object of the ac_gaussian_pyr class with these characteristics and call the top-level run() function:

1. Include <ac_gaussian_pyr.h>
2. Define template parameters and data-types in the CCS_MAIN() function.

```
enum {
  OUT_FRAC_BITS = 10,
  W_MAX = 1024, H_MAX = 1024,
  N_LEVELS = 3,
  MAX_FRAC_BITS = 16, USE_SP = false
};
typedef ac_ipl::RGB_imd<ac_int<8, false> > IN_TYPE;
typedef ac_ipl::RGB_imd<ac_fixed<8 + OUT_FRAC_BITS, 8, false> > OUT_TYPE;
```

3. Instantiate ac_channels with the input and output types defined above in the CCS_MAIN function.

```
ac_channel<IN_TYPE> streamIn;
ac_channel<OUT_TYPE> streamOut[N_LEVELS]; //3 channels; 1 per pyramid level
```

4. Initialize dimension variables:

```
int widthInt  = 512; // Input image Width.
int heightInt = 384; // Input image Height.
```

5. Write all image input values to the streamIn ac_channel. Initialize an object of the ac_gaussian_pyr class with the above static characteristics and call the top level member function with the ac_channel and dimension variables mentioned above:

```
ac_gaussian_pyr<IN_TYPE, OUT_TYPE, W_MAX, H_MAX,
                  N_LEVELS, MAX_FRAC_BITS, USE_SP> gaussPyrInst;
gaussPyrInst.run(streamIn, streamOut, widthInt, heightInt);
```

2.5.4. References

[1] Matlab. Image pyramid reduction and expansion – MATLAB impyramid. Retrieved June 25, 2021, from https://www.mathworks.com/help/images/ref/impyramid.html#bvozh1q

[2] Mielke, Matthias, André Schäfer, and Rainer Brück. "Asic implementation of a gaussian pyramid for use in autonomous mobile robotics." 2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS). IEEE, 2011.

2.6. Wavelet Image Pyramid (ac_dwt2_pyr)

The ac_dwt2_pyr library provides a hardware implementation for the Two-Dimensional Wavelet Pyramid algorithm. By default, it is pipelined with an II of 1 and works on greyscale (unsigned *ac_int)* as well as color (*RGB_imd)* inputs.

2.6.1. Implementation Details and Limitations

Kernels for Different Wavelet Functions

The ac_dwt2_pyr library provides two types of wavelet implementations by default: One for the Haar Wavelet Transform, and the other for the Daubechies-2 Wavelet Transform. The user can add support for more wavelet functions than those provided by default by using ac_dwt2_pyr_lutgen.cpp to generate more LUT values.

The LUT generator file essentially takes 1-D low-pass and high-pass filter coefficients and converts them to four 2-D kernels:

- k_ap: Kernel used to calculate approximation coefficients.
- k_ho: Kernel used to calculate horizontal detail coefficients.
- k_ve: Kernel used to calculate vertical detail coefficients.
- k_di: Kernel used to calculate diagonal detail coefficients.

Converting 1-D coefficients to 2-D kernels is more compatible for 2-D wavelet decomposition applications. The decomposition is now as simple as performing a MAC operation between the kernels and windowed image values.

The wavelet function used can be changed by changing the DWT_FN_VAL template parameter for the top-level ac_dwt2_pyr design. The two possible values for this template parameter are AC_HAAR for the Haar Wavelet Tansform and AC_DB2 for the Daubechies-2 Wavelet Transform. A coding example to instantiate the top-level class with this template parameter is given in Calling the Top-Level Function.

Functional Overview

Similar to the Gaussian Image Pyramid, the Wavelet Pyramid too outputs a multi-resolution representation of an input image, while downsampling the input image by 2 across rows and columns, for each level. Unlike the Gaussian Image Pyramid, however, the Wavelet Pyramid transform calculates highpass representations of the image as well, at each level, namely the horizontal, vertical, and diagonal detail coefficients.

Block Diagram

Refer to Illustration 20 for a generic wavelet pyramid design with N levels (N > 2) and kernels of size KxK. The levels are mapped to a separate module/block with linebuffers. As the outputs are downsampled by 2 across rows and columns, each level sees a halving of the image width and height. The design supports 1<=N<=10.

The pixel input from the external producer is received via the stream_in ac_channel and fed to the Level 1 block. There, it is stored in (K-1) linebuffers which are as big as the maximum possible image width. The linebuffers in turn feed the registered array for the window itself, which is of size KxK.

At every level, a MAC operation is performed between all the 2D kernels (k_ap, k_ho, k_ve and k_di) and the image window. This results in four output values:

- out_ap: Approximation coefficients.
- out_ho: Horizontal detail coefficients.
- out_ve: Vertical detail coefficients.
- out_di: Diagonal detail coefficients.

These four values are packaged in a struct and outputted on an array of ac_channels, i.e. the stream_out array, such that the nth level feeds stream_out[n-1].

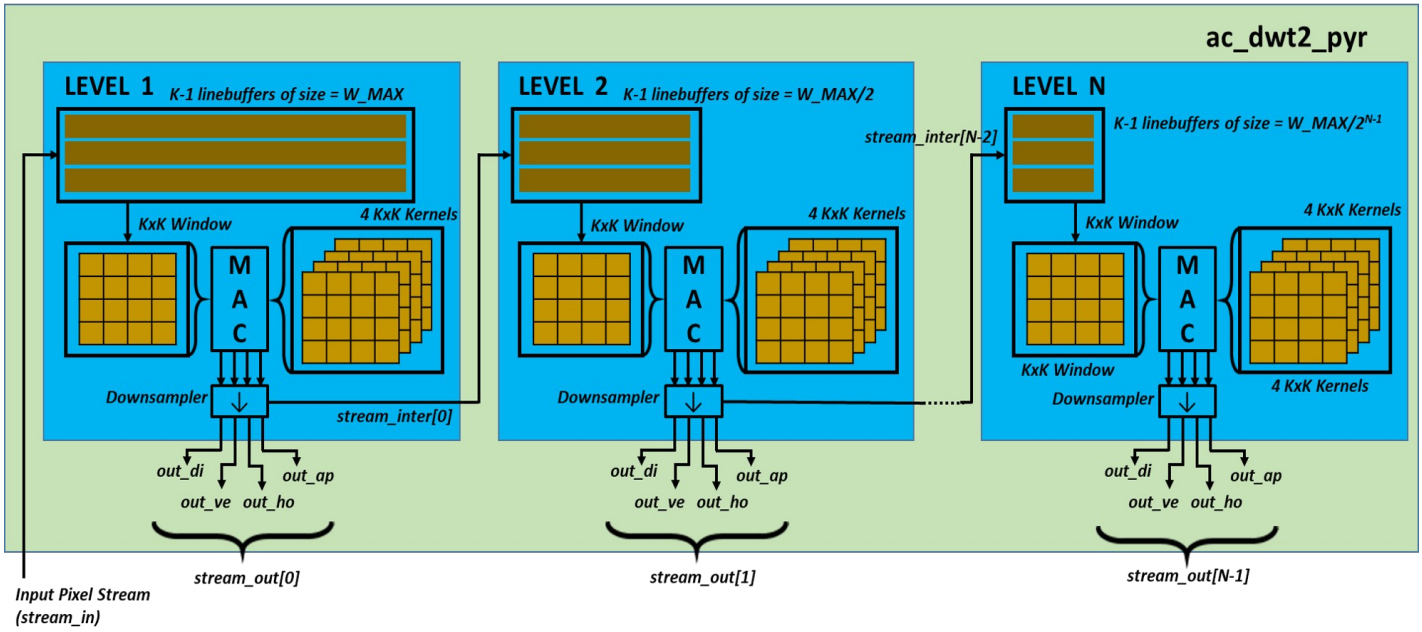

Illustration 20: Block Diagram of Wavelet Pyramid

The approximation coefficients are also sent to the next level on the interconnect channels and serve as inputs to calculate the wavelet decomposition for that level. Level 1 feeds level 2 via the stream_inter[0] interconnect channel, level 2 feeds level 3 via the stream_inter[1] channel, and so on. Due to the aforementioned halving of image dimensions as we move from one level to another, the second level only needs W_MAX/2 elements for each of its linebuffers, the level after that needs W_MAX/4, and so on.

ac_dwt2_pyr is the top-level wrapper class which instantiates all the levels as sub-blocks, and the array of interconnect ac_channels as static data members.

Catapult Architectural Exploration Options

The design is pipelined with an II of 1 by default. Color implementations need three memories per linebuffer—one for each color component—while greyscale implementations only need one memory per linebuffer.

For a kernel of size KxK, the design needs 3*(K-1) memories per level for color implementations ((K-1) linebuffers x 3 memories per linebuffer) and (K-1) memories per level for greyscale implementations. Refer to Illustration 21 for a view of the linebuffer mapping for Level 1 of a sample dualport color implementation with AC_DB2 kernels (i.e. K = 4) and W_MAX = 1024.

If the user wishes to maintain pipelining with an II of 1 and switch between singleport and dualport memory implementations, they can do so by changing the USE_SP template parameter. More details on doing so are given in Model Parameters. There can be certain cases in which linebuffers are small enough that they fall below the memory threshold and are mapped to register banks by default, e.g. if the number of levels is so large that that the max. input image width falls below the memory threshold (remember that image width gets halved at every level due to downsampling).

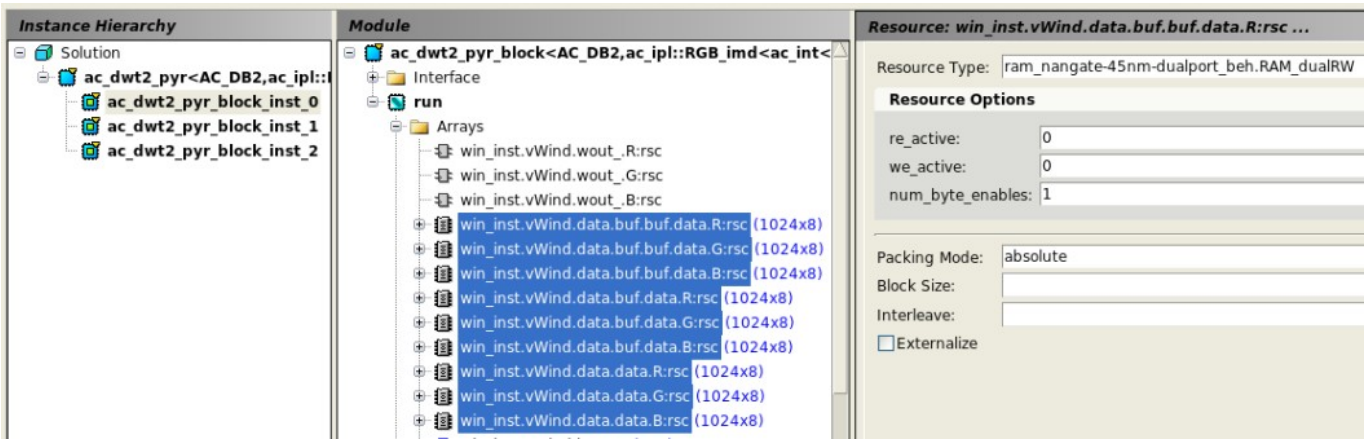

Illustration 21: Array Options

The loop mapping for the Level 1 sub-block mentioned above can be seen in Illustration 22: Loop Options. Since the column processing loop (COL_LOOP) and, by extension, the row processing loop (ROW_LOOP) have memory and IO accesses, they cannot be fully unrolled and are instead pipelined with an II of 1, as is the design as a whole. All the other loops are fully unrolled to allow for maximum throughput and parallelism. This unrolling is possible due to the Daubechies-2 window (size: 4x4) and the Haar window (size: 2x2) being small-enough 2D arrays that they can be mapped to registers, hence allowing for parallel array accesses. If the user intends on increasing the kernel/window size and maintaining full unrolling, they must make sure that the size of the kernel and window arrays falls below the memory threshold, failing which they can:

- Explicitly map the window arrays to register banks (provided that they're below the register threshold).
- Increase the memory threshold to allow for register mapping.

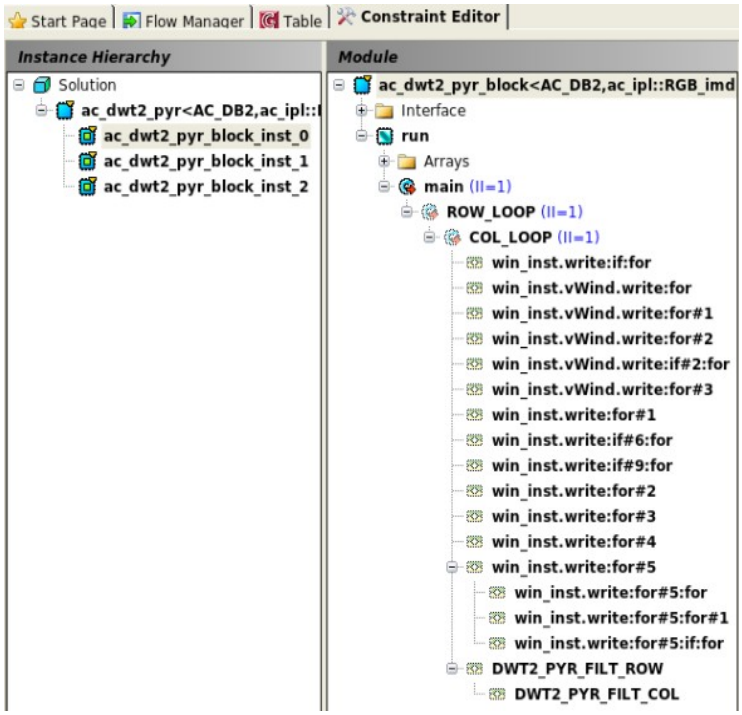

Illustration 22: Loop Options

Since the dimension inputs (w_in and h_in) are supplied to every level sub-block, they are mapped to DirectInputs. Failure to do so will result in catapult throwing an error like this at the “go architect” stage:

```
# Error: ac_dwt2_pyr.h(336): Resource '/
ac_dwt2_pyr<AC_DB2,ac_ipl::RGB_imd<ac_int<8,false>>,ac_ipl::RGB_imd<ac_fixed<24
,12,true,AC_TRN,AC_WRAP>>,1024,1024,3,24,12,false>/w_in:rsc' with variable con-
nected to multiple sub-blocks not mapped to '[DirectInput]' (MEM-71)
```

While this is the error transcript for the width input (w_in) interface of a dualport color implementation specifically, a similar error is displayed for other implementations if the mapping isn’t correct.

Port Descriptions

Refer to Illustration 23 for a view of the ports/pins of an example Wavelet Pyramid design with three levels, as seen in Design Analyzer. The description of each of the ports is given in the table below:

Pin	Direction	Description
clk	IN	Clock signal.
rst	IN	Reset.
w_in, h_in	IN	DirectInputs for width and height, respectively.
stream_in:rsc.dat	IN	Input data stream.
stream_in:rsc.rdy	OUT	Input ready signal, goes high for a cycle to acknowledge data read.
stream_in:rsc.vld	IN	Input valid signal, goes high when data is valid on clock edge.
steam_out(n):rsc.dat	OUT	Output data stream for (n+1)th level.
steam_out(n):rsc.rdy	IN	Driven high when external consumer requests data from level (n+1).
steam_out(n):rsc.vld	OUT	Driven high by design when data is ready to be read from level (n+1).

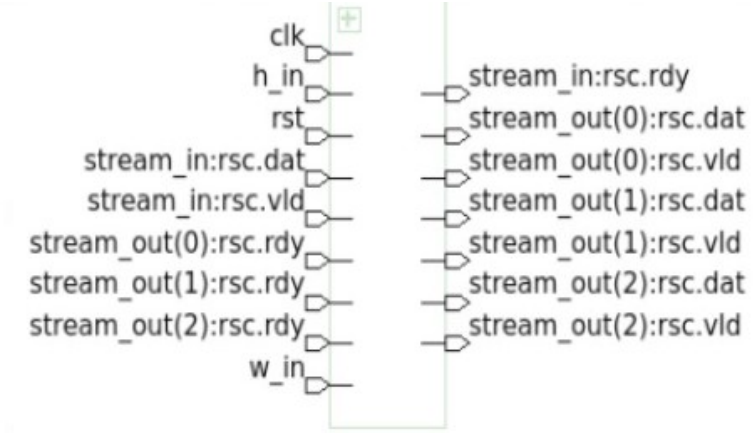

Illustration 23: Pin Diagram for ac_dwt2_pyr.

Limitations

- As mentioned earlier, the dimension inputs can only be mapped to DirectInputs, as they supply multiple sub-blocks, and as such, the user must hold them stable while the design has data in the pipeline.

- Only ac_int or RGB_imd inputs that satisfy the criteria described in Model Parameters can be used.
- The width and height for the image input for every level must be greater than the width and height of the kernels. This is especially important for a large number of levels, since the image input dimensions get halved for each level and might hence fall below this threshold. If this condition is not met, a runtime AC_ASSERT will be triggered.
- The IP does not support more than ten levels.

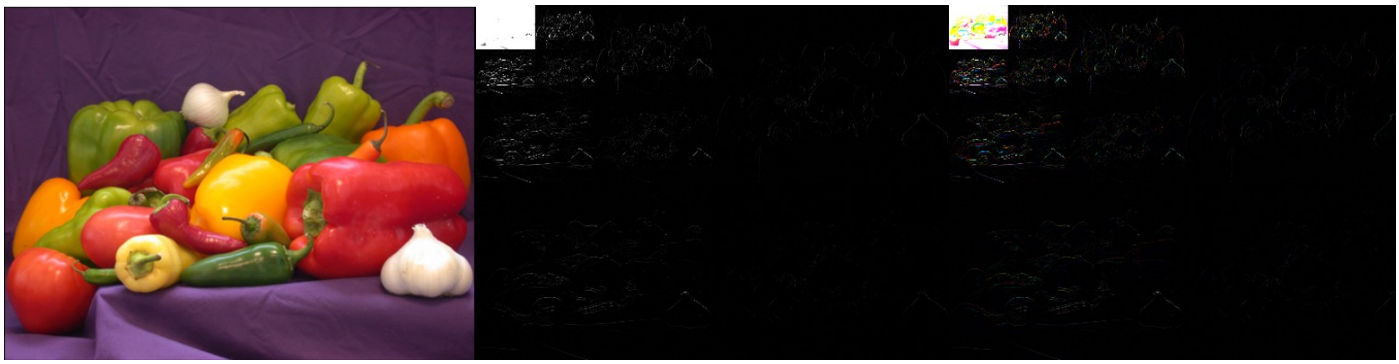

Illustration 24: Input and Output Images. From left to right: Input Color image, output greyscale image, output color image.

Final Output

Consider two example three-level designs, one which accepts greyscale inputs and the other that accepts color inputs. The output of this image image can continuously divided into quadrants for every level which are used to store the coefficients, as is represented in Illustration 25.

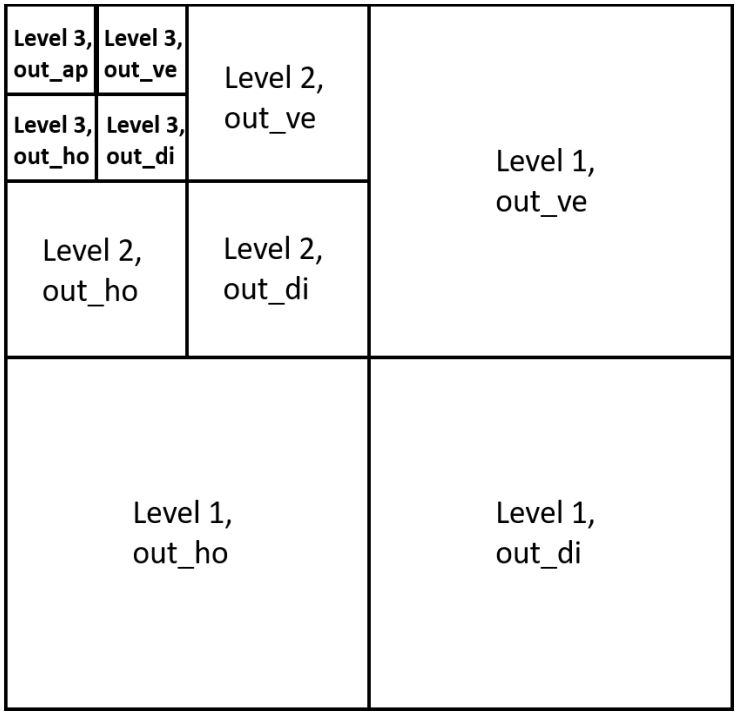

Illustration 25: Output Storage Format.

Refer to Illustration 24 for input and output images of the example designs mentioned above. For the greyscale designs, the image was first converted to a greyscale representation which isn't shown in the illustration.

2.6.2. Model Parameters

Given below is a snippet of the ac_dwt2_pyr class:

```
template <ac_dwt2_function DWT_FN_VAL, class IN_TYPE, class OUT_TYPE, int
W_MAX, int H_MAX, int N_LEVELS, int ACC_W = 32, int ACC_I = 16, bool USE_SP =
true>
class ac_dwt2_pyr {
  public:
  // Code
  typedef ac_int<ac::nbits<W_MAX>::val, false> W_IN_TYPE;
  typedef ac_int<ac::nbits<H_MAX>::val, false> H_IN_TYPE;
  #pragma hls_pipeline_init_interval 1
  #pragma hls_design interface
  void CCS_BLOCK(run) (
    ac_channel<IN_TYPE>  &stream_in, // Input stream
    ac_channel<out_struct<OUT_TYPE> > stream_out[N_LEVELS], // Output stream.
    const W_IN_TYPE w_in, // Dimension input: width.
    const H_IN_TYPE h_in // Dimension input: height.
  )
  // Code
};
```

The definition of the template parameters shown in the snippet is as follows:

Parameters	Description
DWT_FN_VAL	Enum to select between wavelet functions. Possible values: AC_HAAR, AC_DB2.
IN_TYPE	Input type.
OUT_TYPE	Output Coefficent Type.
W_MAX, H_MAX	Maximum input image width and height, respectively.
N_LEVELS	Number of levels.
Following parameters are optional:	
ACC_W	Bitwidth used for interconnect IO types. 32 by default.
ACC_I	Integer width used for interconnect IO types. 16 by default.
USE_SP	Enables the usage of single-port memories with an II of 1 if set to true (default).

The design only allows the following IO datatype combinations:

- IN_TYPE: Unsigned ac_int, OUT_TYPE: ac_fixed

- IN_TYPE: RGB_imd<Unsigned ac_int>, OUT_TYPE: RGB_imd<ac_fixed>

Note that the output ac_fixed type doesn't have to be unsigned, while the input ac_int type has to be. The precision used for the interconnect variables is the same as that used for the accumulator variables, hence the template arguments for the same have the *ACC_** prefix.

2.6.3. Calling the Top-Level Function

Let us consider a dualport, RGB_imd implementation with the following static characteristics:

- IN_TYPE: RGB_imd<ac_int<8, false> >
- OUT_TYPE: RGB_imd<ac_fixed<24, 12, true> >
- W_MAX/H_MAX = 1024
- N_LEVELS = 3/
- ACC_W = 24, ACC_I = 12
- USE_SP = false

The following are the steps used to initialize an object of the ac_dwt2_pyr class with these characteristics and call the top-level run() function:

1. Include <ac_dwt2_pyr.h>
2. Define template parameters and data-types in the CCS_MAIN() function.

```
enum {
   DWT_FN_VAL = AC_DB2,
   W_MAX = 1024, H_MAX = 1024,
   N_LEVELS = 3,
   ACC_W = 24, ACC_I = 12,
   OUT_BW = 24, OUT_IW = 12,
   USE_SP = false
};
typedef ac_ipl::RGB_imd<ac_int<8, false> > IN_TYPE;
typedef ac_ipl::RGB_imd<ac_fixed<OUT_BW, OUT_IW, true> > OUT_TYPE;
```

3. Instantiate ac_channels with the input and output types defined above in the CCS_MAIN function.

```
ac_channel<IN_TYPE> stream_in;
ac_channel<out_struct<OUT_TYPE> > stream_out[N_LEVELS];
```

   Note that out_struct is the struct mentioned in Block Diagram, i.e. the wrapper struct that packages the out_ap, out_ho, out_ve and out_di coefficients.

4. Initialize dimension variables:

```
int width_int  = 512; // Input image Width.
int height_int = 384; // Input image Height.
```

5. Write all image input values to the stream_in ac_channel. Initialize an object of the ac_dwt2_pyr class with the static characteristics mentioned above and call the top level member function with the ac_channel and dimension variables:

```
ac_dwt2_pyr<ac_dwt2_function(DWT_FN_VAL), IN_TYPE, OUT_TYPE, W_MAX, H_MAX,
N_LEVELS, ACC_W, ACC_I, USE_SP> dwt2_pyr_inst;
dwt2_pyr_inst.run(stream_in, stream_out, width_int, height_int);
```

2.7. Harris Corner Detector (ac_harris)

The ac_harris library provides an efficient hardware implementation for the harris corner detector algorithm. It can be pipelined with an II of 1 and is implemented for RGB and grayscale images.

2.7.1. Implementation Details

The harris corner detection algorithm has several stages. Each stage corresponds to a sub-block of the design, which is coded as a C++ function. The sub-blocks communicate with each other and the top-level wrapper using *ac_channel* interconnects. The block diagram for Harris Corner Detector algorithm is as below:

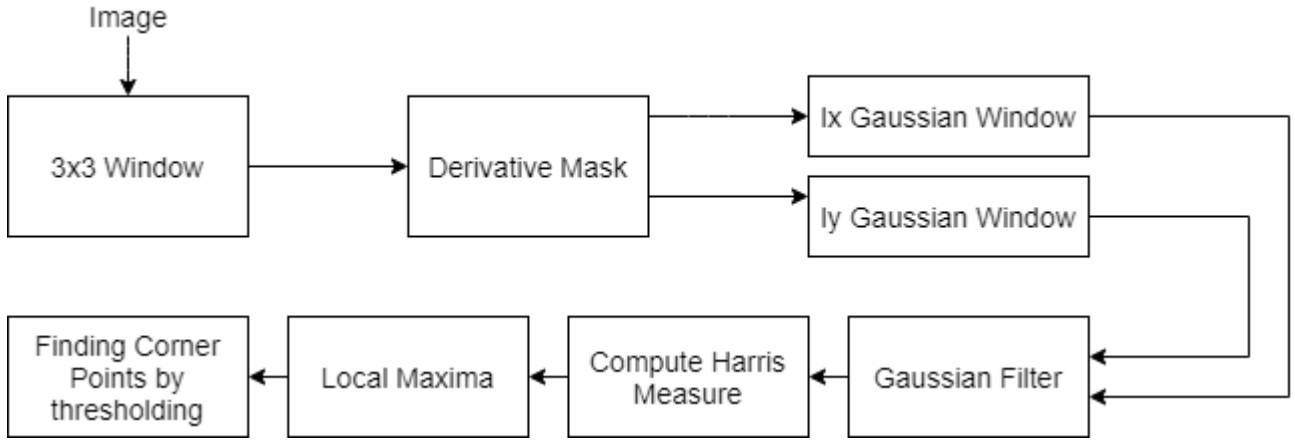

Illustration 26: Harris Corner Detector Block Diagram

Functionality of Sub-blocks

- The first sub-block, *intensity* calculates the image gradients Ix and Iy by convolution of 3x3 image window and derivative mask.

- The second sub-block, *harrisresponse*, calculates the Harris Response which is given by 2*((gaussOpxx*gaussOpyy) – (gaussOpxy*gaussOpxy)) / (gaussOpxx + gaussOpyy + epsilon) where gaussOpxx is the result of convolution of the gaussian filter and Ix2 (Ix*Ix) and similarly for gaussOpyy and gaussOpxy.

- The third subblock localmaxima calculates the maximum value in a 3x3 neighborhood of the Harris Response matrix

- The fourth subblock thresholding finds out the corner points. A corner point is detected if the Harris Response is equal to local maxima and greater than the threshold. An ideal value for threshold is 1000, higher the value lesser the number of corner points and vice versa.

Using Singleport Memories

Some of the sub-blocks uses *ac_window_2d_flag* objects to implement a sliding image window. The linebuffers used for the sliding window implementation are by default intended to be mapped to dual-port memories, if the design is to be pipelined with an II of 1. If the user wishes to pipeline the design with an II of 1 and use single-port memories, they should supply “true” as the value of the USE_SINGLEPORT template parameter. Please refer to Calling the Top-level Function for a relevant coding example.

Catapult Architectural Exploration Options

The design as a whole is pipelined with an II of 1. Every sub-block has two nested row and column loops, which are also pipelined with an II of 1. All other loops are fully unrolled by default. The loops can also be chosen to be partially unrolled instead in order to reduce the area. The unrolled loops are mostly loops internal to the *ac_window_2d_flag* code, with the remaining loops being dedicated to analyzing the windowed data or performing convolution, depending on the sub-block. The interface for the harris corner detector is shown in Illustration 27: Interface for Harris Corner Detector.

Illustration 27: Interface for Harris Corner Detector

StreamIn and StreamOut are the input and output data streams. WidthIn, heightIn, component, epsilon and threshold are the direct inputs supplied by the user. P1, P2, P3, P4 and P5 are the internal ac_channels used in the design.

Please refer to Illustration 28: Loop Options for a view of the loops in the Catapult architectural constraints editor. Some of the loops in the sub-block *intensity* are used as an example.

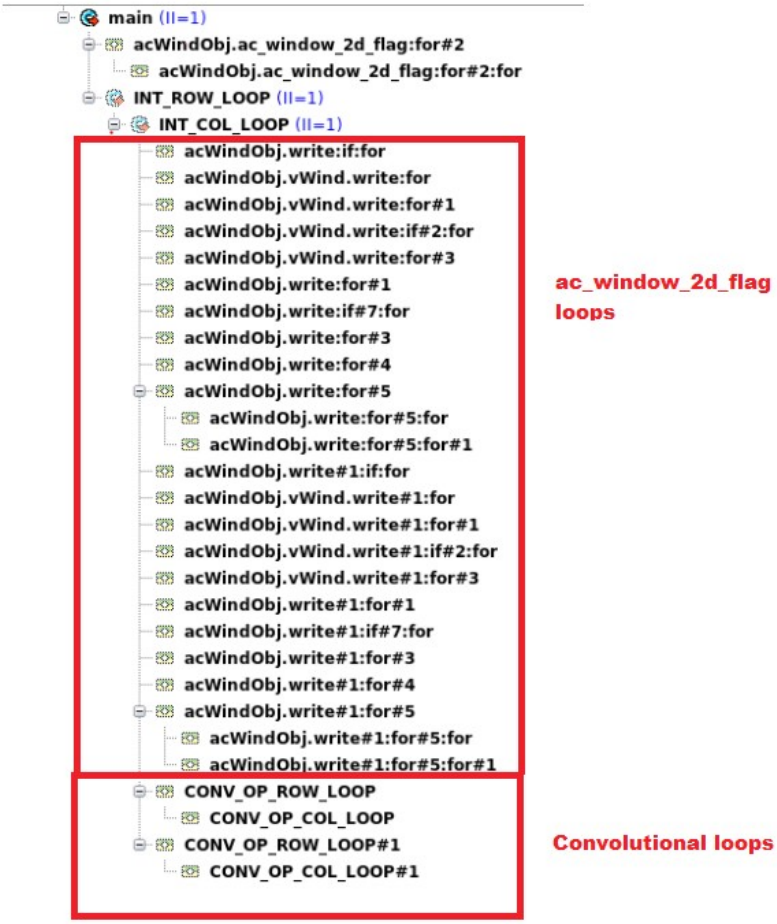

Illustration 28: Loop Options

If the *ac_window_2d_flag* linebuffer sizes fall above the memory threshold, they will be mapped to memories. The user can facilitate the mapping to single-port memories by using the *USE_SINGLEPORT* template parameter, as explained in Using Singleport Memories. The other arrays, including the internal *ac_window_2d_flag* arrays, will be mapped to register banks, provided that the window size stays below the memory threshold. These arrays must be mapped to register banks if the internal *ac_window_2d_flag* loops are to be fully unrolled.

Please refer to Illustration 29: Array Options for a view of the array mapping, as seen in the Catapult architectural constraints editor. Once again, the *intensity* sub-block is chosen as an example.

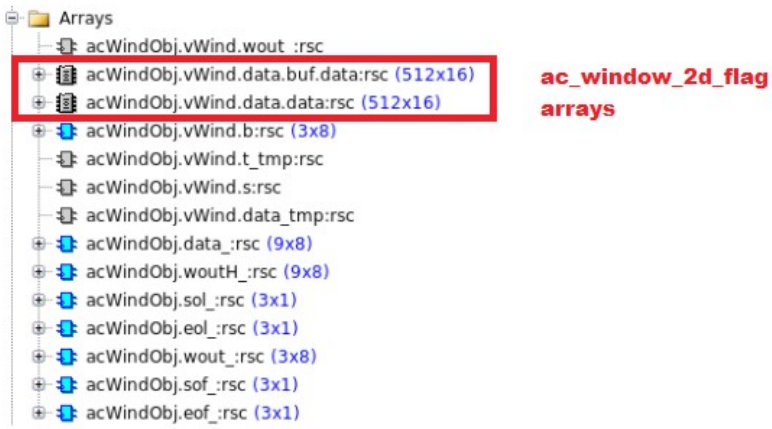

Illustration 29: Array Options

Final Output

The result of the harris corner detector library on a sample input grayscale image is shown in Illustration 30: Grayscale Input/Output Harris Corner Detector Library. The image on the left is the input image, while the image on the right is the output image whose corners are colored white.

Illustration 30: Grayscale Input/Output Harris Corner Detector Library

The result of the harris corner detector library on a sample input rgb image is shown in Illustration 31: RGB Input/Output Harris Corner Detector Library. The image on the left is the input image, while the image on the right is the output image whose corners are colored white.

Illustration 31: RGB Input/Output Harris Corner Detector Library

2.7.2. Model Parameters

The harris corner detector library is implemented as a class template named *ac_localcontrastnorm*. A public member function of the class, called *run(..)* acts as the top-level design block.

Given below is a snippet of the *ac_harris* class.

```
template <class IN_TYPE, class OUT_TYPE, unsigned CDEPTH, unsigned W_MAX, un-
signed             H_MAX, bool USE_SINGLEPORT = false>
class ac_harris
{
  // Code
}
```

The definition of the template parameters shown in the snippet is given below:

Parameters	Description
IN_TYPE	Input type of the input image
OUT_TYPE	Output type of the output image
CDEPTH	Color depth of input image.
W_MAX, H_MAX	Maximum input image width and height, respectively.
USE_SINGLEPORT	Default template parameter; helps the user choose a singleport implementation.

As *USE_SINGLEPORT* is set to *false* by default, the default harris corner detector implementation is designed with a dual-port implementation in mind.

2.7.3. Calling the Top-level Function

Let us consider a hardware harris corner detector implementation for a grayscale image with the following static characteristics:

- Color-depth of 8.
- Maximum image width/height of 1024.
- Single-port implementation support.

The following are the steps to initialize an object of the *ac_harris* class with the above mentioned static characteristics and call the *run(..)* function:

1. Include <ac_harris.h>
2. Define data-type for input pixels/threshold values, output pixels, as well as image dimensions in the CCS_MAIN() function.

```
typedef ac_int<CDEPTH, false> pixInType;  // Input pixel type
typedef ac_int<CDEPTH, false> pixOutType; // Output pixel type
typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;  // Input width
type
typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType; // Input height
type
```

3. Instantiate *ac_channel*s with the input and output types defined above in the CCS_MAIN() function.

```
ac_channel<pixInType>  streamIn;
ac_channel<pixOutType> streamOut;
```

4. Initialize dimension variables.

```
widthInType  widthIn     = 512;
heightInType heightIn    = 512;
```

5. Initialize the component, epsilon and threshold variables.

```
ac_int<2, false> component = 0;
ac_int<3, false> epsilon = 6;
ac_int<14, false> threshold = 1000;
```

6. Write all input values to the streamIn *ac_channel*. Instantiate an object of the *ac_harris* class with the above static characteristics and call the top level member function with the ac_channels, dimension, component, epsilon and threshold variables mentioned above.

```
ac_harris<pixInType, pixOutType, 8, 1024, 1024, true> harrisObj;
```

```
harrisObj.run(streamIn, streamOut, widthIn, heightIn, component, epsilon,
       threshold);
```

Limitations

- As mentioned earlier, the dimension inputs can only be mapped to DirectInputs, as they supply multiple sub-blocks, and as such, the user must hold them stable while the design has data in the pipeline.
- Only ac_int or RGB_imd inputs that satisfy the criteria can be used.

2.8. Image Histogram (ac_imhist)

The ac_imhist library provides a hardware implementation for calculating an image histogram. It can be pipelined with an II of 1 and works on greyscale (unsigned *ac_int)* as well as color (*RGB_1PPC)* inputs, while always providing unsigned *ac_int* outputs.

2.8.1. Implementation Details

- If the input passed is a greyscale image, then the outputs are histogram counts for each intensity level.
- If the input passed is a color image, then the library considers all three color components to be separate greyscale images and accumulates histogram counts across color components to give the final output.
- The histogram counts are stored in an array, called *histArr*. The number of elements in this array are the same as the number of possible intensity levels, i. e. 2W where W is the width of each pixel (for greyscale images) or color component (for color images). The design will either map the array to a register bank or a memory, depending on the number of intensity levels, the memory threshold and the register threshold.

Catapult Architectural Exploration Options

All the design loops are, by default, pipelined with an II of 1, as shown in the Architectural Constraints Editor Screenshot in Illustration 32.

The architectural exploration options for each loop are detailed as follows:

- INIT_ARR initializes all *histArr* locations to zero, so as to clear out histogram counts from previous calls to the *run()* function. If *histArr* is small enough to be mapped to a register bank, this loop can be fully unrolled. If not, the loop can be pipelined with an II >= 1.
- READ_INPUT reads inputs from the input channel and updates histogram counts according to the input value. To avoid feedback violations and reduced memory accesses, the updated histogram counts are only written to the memory when the intensity level received in the current iteration is different than the one received in the previous iteration. Since this loop has input channel accesses, it cannot be fully unrolled. It is instead possible to pipeline it with an II >= 1, if using dualport RAMs.
- WRITE_OUTPUT reads each *histArr* index and writes to the output channel. Since this loop contains output channel accesses, it cannot be fully unrolled, but it is possible to pipeline with an II >= 1.

Illustration 32: Loop Options

As mentioned earlier, the mapping of *histArr* depends on its size and pre-defined thresholds. For a greyscale image input, *histArr* only consists of a single greyscale component. For a color image input, *histArr* has separate RGB components. Refer to Illustration 33 for a view of this array in the Architectural Constraints Editor, for an arbitrary use case (Input color-depth = 8). The design can use singleport or dualport memories. However, if singleport memories are used, the design must be pipelined with an II >= 2 so as to avoid scheduling conflicts, thereby limiting the throughput. Dualport memories allow the user to pipeline the design with an II >= 1, thereby improving the throughput while also increasing the area.

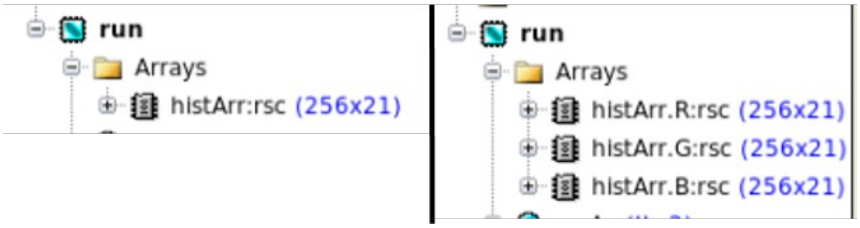

Illustration 33: Array Options. Left: Greyscale Input, Right: Color Input.

Limitations

- The input must either be an unsigned *ac_int* or *RGB_1PPC* input. A *static_assert* is triggered if this condition is violated.
- As mentioned earlier, implementations that use singleport memories cannot be pipelined with an II of 1, because that will result in scheduling conflicts.

2.8.2. Model Parameters

The image histogram library is implemented as a class template named *ac_imhist*. A public member function of the class, called run(..), acts as the top-level design block. Given below is a snippet of the *ac_imhist* class.

```
template <class IN_TYPE, int OUT_BW, unsigned W_MAX, unsigned H_MAX>
class ac_imhist
{
  // Code
  typedef ac_int<OUT_BW, false> OUT_TYPE;
  typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;
  typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType;
  // Code
#pragma hls_pipeline_init_interval 1
```

```
#pragma hls_design interface
  void CCS_BLOCK(run) (
    ac_channel<IN_TYPE>  &streamIn,  // Pixel input stream
    ac_channel<OUT_TYPE> &streamOut, // Pixel output stream
    const widthInType    widthIn,    // Input width
    const heightInType   heightIn    // Input height
  )
  // Code
};
```

The definition of the template parameters shown in the snippet is given below:

Parameters	Description
IN_TYPE	Input type.
OUT_BW	Output Bitwidth.
W_MAX, H_MAX	Maximum input image width and height, respectively.

2.8.3. Calling the Top-level Function

Let us consider an image histogram implementation with the following static characteristics:

- 24 bpp color image input.
- 22-bit output.
- Maximum image width/height of 1024.

The following are the steps to initialize an object of the *ac_imhist* class with the above mentioned static characteristics:

1. Include <ac_imhist.h>
2. Define template parameters and data-types in the CCS_MAIN() function.

```
enum {IN_CDEPTH = 8, OUT_BW = 22, W_MAX = 1024, H_MAX = 1024};
typedef ac_ipl::RGB_1PPC<IN_CDEPTH> IN_TYPE;  // Input Pixel Type
typedef ac_int<OUT_BW, false> OUT_TYPE; // Output Pixel Type
typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;  // Input width
type
typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType; // Input height
type
```

3. Instantiate *ac_channel*s with the input and output types defined above in the CCS_MAIN() function.

```
ac_channel<IN_TYPE>  streamIn;
ac_channel<OUT_TYPE> streamOut;
```

4. Initialize dimension variables.

```
widthInType  widthIn  = 512; // Input Image Width
heightInType heightIn = 384; // Input Image Height
```

5. Write all image input values to the streamIn *ac_channel*. Instantiate an object of the *ac_imhist* class with the above static characteristics and call the top level member function with the ac_channels and dimension variables mentioned above.

```
ac_imhist<IN_TYPE, OUT_BW, W_MAX, H_MAX> imhistObj;
imhistObj.run(streamIn, streamOut, widthIn, heightIn);
```

2.9. Local Contrast Normalization (ac_localcontrast-norm)

The ac_localcontrastnorm library provides an efficient hardware implementation for the local contrast normalization algorithm. It can be pipelined with an II of 1 and works with *RGB_1PPC* datatypes.

2.9.1. Implementation Details

The local contrast normalization algorithm has several stages. Each stage corresponds to a sub-block of the design, which is coded as a C++ function. The sub-blocks communicate with each other and the top-level wrapper using *ac_channel* interconnects.

Functionality of Sub-blocks

- The first sub-block, *getcenteredimage* subtracts the Gaussian filtered input image from the input image to get the centered image.

- The second sub-block, *getstddeviation*, calculates the standard deviation of the image which is the square root of the variance. Here the variance is the gaussian filtered output of the square of the centered image. Then the local contrast normalized output of the image is calculated by dividing the centered image by the standard deviation.

Using Singleport Memories

Each of the sub-blocks uses *ac_window_2d_flag* objects to implement a sliding image window. The linebuffers used for the sliding window implementation are by default intended to be mapped to dual-port memories, if the design is to be pipelined with an II of 1. If the user wishes to pipeline the design with an II of 1 and use single-port memories, they should supply “true” as the value of the USE_SINGLEPORT template parameter. Please refer to Calling the Top-level Function for a relevant coding example.

Catapult Architectural Exploration Options

The design as a whole is pipelined with an II of 1. Every sub-block has two nested row and column loops, which are also pipelined with an II of 1. All other loops are fully unrolled by default. The unrolled loops are mostly loops internal to the *ac_window_2d_flag* code, with the remaining loops being dedicated to analyzing the windowed data or performing convolution, depending on the sub-block. Please refer to Illustration 34:

Loop Options for a view of the loops in the Catapult architectural constraints editor. Some of the loops in the sub-block *getcenteredimage* are used as an example.

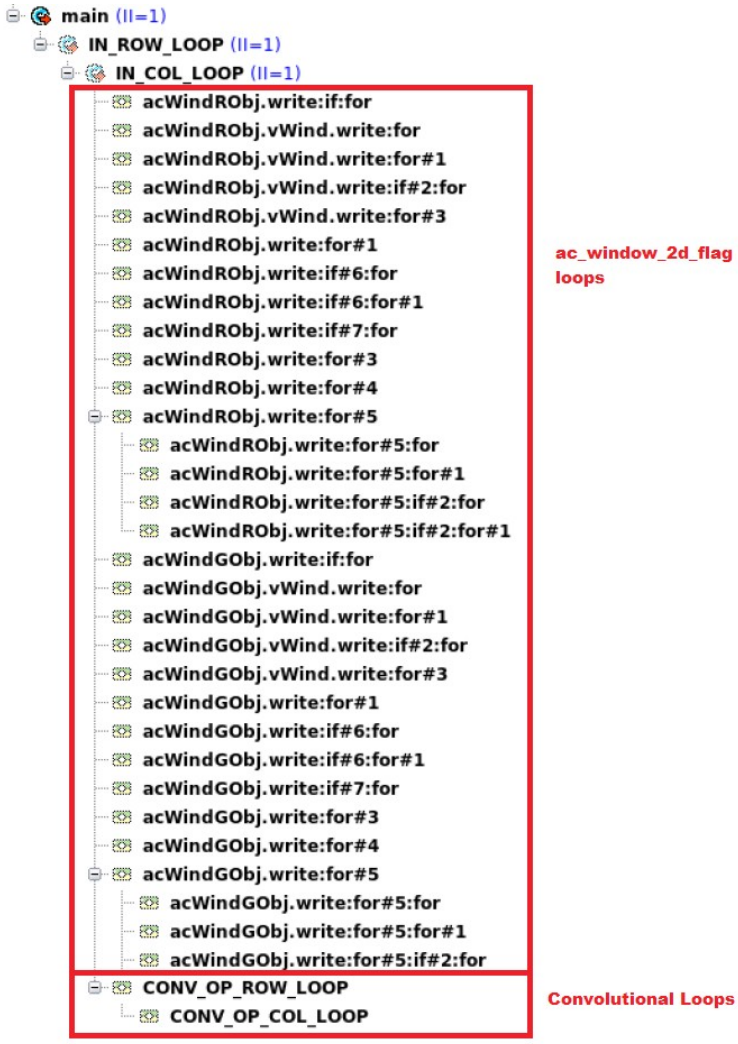

Illustration 34: Loop Options

If the *ac_window_2d_flag* linebuffer sizes fall above the memory threshold, they will be mapped to memories. The user can facilitate the mapping to single-port memories by using the *USE_SINGLEPORT* template parameter, as explained in Using Singleport Memories. The other arrays, including the internal *ac_window_2d_flag* arrays, will be mapped to register banks, provided that the window size stays below the memory threshold. These arrays must be mapped to register banks if the internal *ac_window_2d_flag* loops are to be fully unrolled, so as to facilitate multiple array accesses per clock cycle and for this purpose the memory threshold has been set to 128 since the maximum size of the internal *ac_window_2d_flag* arrays is 81.

Please refer to Illustration 35: Array Options for a view of the array mapping, as seen in the Catapult architectural constraints editor. Once again, the *getcenteredimage* sub-block is chosen as an example.

Illustration 35: Array Options

Final Output

The result of the local contrast normalization library on a sample input image is shown in Illustration 36: Input/Output for Local Contrast Normalization Library. The image on the left is input image, while the image on the right is the local contrast output.

Illustration 36: Input/Output for Local Contrast Normalization Library

2.9.2. Model Parameters

The local contrast normalization library is implemented as a class template named *ac_localcontrastnorm*. A public member function of the class, called *run(..)* acts as the top-level design block.

Given below is a snippet of the *ac_localcontrastnorm* class.

```
template <unsigned CDEPTH, unsigned W_MAX, unsigned H_MAX,
          bool USE_SINGLEPORT = false>
class ac_localcontrastnorm
{
  // Code
}
```

The definition of the template parameters shown in the snippet is given below:

Parameters	Description
CDEPTH	Color depth of input image.
W_MAX, H_MAX	Maximum input image width and height, respectively.
USE_SINGLEPORT	Default template parameter; helps the user choose a singleport implementation.

As *USE_SINGLEPORT* is set to *false* by default, the default local contrast normlization implementation is designed with a dual-port implementation in mind.

2.9.3. Calling the Top-level Function

Let us consider a hardware local contrast normalization implementation with the following static characteristics:

- Color-depth of 8.
- Maximum image width/height of 1024.
- Single-port implementation support.

The following are the steps to initialize an object of the *ac_localcontrastnorm* class with the above mentioned static characteristics and call the *run(..)* function:

1. Include <ac_localcontrastnorm.h>
2. Define data-type for input pixels/threshold values, output pixels, as well as image dimensions in the CCS_MAIN() function.

```
typedef ac_ipl::RGB_1PPC<CDEPTH> pixInType;  // Input pixel/threshold value
type
typedef ac_ipl::RGB_1PPC<CDEPTH> pixOutType; // Output pixel type
typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;  // Input width
type
typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType; // Input height
type
```

3. Instantiate *ac_channel*s with the input and output types defined above in the CCS_MAIN() function.

```
ac_channel<pixInType>  streamIn;
ac_channel<pixOutType> streamOut;
```

4. Initialize dimension variables.

```
 widthInType  widthIn     = 512;
 heightInType heightIn    = 512;
```

5. Write all input values to the streamIn *ac_channel*. Instantiate an object of the *ac_localcontrastnorm* class with the above static characteristics and call the top level member function with the ac_channels, dimension and threshold variables mentioned above.

```
ac_localcontrastnorm<8, 1024, 1024, true> LocalContrastNormObj;
LocalContrastNormObj.run(streamIn, streamOut, widthIn, heightIn);
```

2.10. Denoising Filter (ac_denoise_filter)

The ac_denoise_filter library provides an efficient hardware implementation where median filtering is done to remove the noise from the image. It can be pipelined with an II of 1 and works with ac_int inputs/outputs.

2.10.1. Introduction

Basically for the denoising filter, median filtering is carried out. An ac_window object is created using the ac_window library where the input values are stored. These values are copied to an array. This array is then sorted in ascending order. The sorting algorithm used is the insertion sorting algorithm.

2.10.2. C++ Code Overview

As mentioned earlier, the top-level design and the sub-blocks are all implemented as C++ classes. Given below is a snippet showing the template parameters of all these classes, along with the run() function and all the associated IO typedefs:

```
template <class IN_FXPT_TYPE, int NU, int M_0_SYMB_MAX>
class ac_lte_map
{
class ac_denoise_filter
{
public:
  // Define IO types.
  typedef ac_int<CDEPTH, false> pixInType;
  typedef ac_int<CDEPTH, false> pixOutType;
  typedef ac_int<ac::nbits<W_MAX>::val, false> widthInType;
  typedef ac_int<ac::nbits<H_MAX>::val, false> heightInType;

  #pragma hls_pipeline_init_interval 1
  #pragma hls_design interface
  void CCS_BLOCK(run) (
    ac_channel<pixInType>  &streamIn,   // Pixel input stream
    ac_channel<pixOutType> &streamOut,  // Pixel output stream
    const widthInType      widthIn,     // Input width
    const heightInType     heightIn    // Input height
  ) {

[…]
  // Code
```

2.10.3. Architectural Overview

As can be seen in the snippet above, all the top-level run() functions are pipelined with an II of 1, which is signaled by the pragma hls_pipeline_init_interval 1.

2.10.4. Limitations

- Better methods like wavelet denoising, neural networks, partial differential equations can be used to obtain better filtering.

2.11. Line Buffer (ac_linebuffer)

Line buffers are crucial in real-time image and video processing applications. These memory buffers temporarily store a line of pixels from an image, allowing for efficient and parallel processing of image data. Line buffers play an essential role in various applications such as real-time video analytic, image processing, and machine learning. The design of line buffers requires a delicate balance between performance, memory utilization, and power consumption.

The ac_linebuffer library includes the buffers are mapped to 3 types memory and a set of multiplexers select the data to be processed based on choice of architecture as listed in the table below respectively.

Buffer Type (BUFF_TYPE)	Memory mapped to	Architecture
AC_DUAL	ccs_ram_sync_1R1W	Shift
AC_SPSHIFT	ccs_ram_sync_singleport	Shift
AC_SPCIR	ccs_ram_sync_singleport	Circular
AC_SPWRMASK	ccs_ram_sync_singleport_wmask	Circular

2.11.1. Code Overview

Model Parameters

Parameter name	Description
INPUT_TYPE	That sets the din type, which is determines the width of each element of a buffer. Design supports only INPUT_TYPE of Packed Vector Type (ac_packed_vector) only.

OUTPUT_TYPE	Determines the output type, supported types are
	`ac_array<ac_int<32,false>,FILTER_HEIGHT>`
	`ac_array<ac_packed_vector<ac_int<32,false>, AC_BUS_WORDS>,FILTER_HEIGHT>`

AC_WIDTH	Line buffer width, Note: For dualport memory mappings, AC_WIDTH is expected to be a multiple of the words packed in the ac_packed vector input. For singleport memory mappings, AC_WIDTH is expected to be divisible by twice the number of packed words.
AC_NUM_LINES	Number of Rows in a line buffer, is also inferred as the filter height.
BUFF_TYPE	Of type ac_buff_type which is a enum described below, supported buffer types as listed above.
AC_REPEAT	Boolean type, indicating if the line data is required to be reused.

Supported Buffer Types

Top level ac_linebuffer class instantiates template specialized sub classes which is based on the template parameter BUFF_TYPE, each buffer is declared as a member of struct with respective asserts as illustrated in the sample code block below and the respective implementation names for each buffer types.

```
//Enums defining the Buffer types.
enum ac_buff_type {AC_DUAL,
```

```
                    AC_SPSHIFT,
                    AC_SPWRMASK,
                    AC_SPCIR
                   };

// Generic struct declaration to later be specialized for defining class type.
template <typename IN_TY, int BUFF_WIDTH, int NUM_LINES,
           ac_buff_type buff_typ,  bool ac_rep> struct buffer_type_struct {};

// Template specialized struct to declare the class instance of type
//   where mem is mapped to a Single memory resource and the buffer
//   architecture is shift based.
template <typename IN_TY, int BUFF_WIDTH, int NUM_LINES, bool ac_rep>
   struct buffer_type_struct<IN_TY, BUFF_WIDTH, NUM_LINES, AC_SPSHIFT, ac_rep>
{
typedef ac_linebuffer_spshift<IN_TY, BUFF_WIDTH, NUM_LINES, ac_rep> Buff;
static_assert(BUFF_WIDTH%2 == 0, "For single port memory mapped Buffer,
             effective Buffer Width(AC_BUFFER_WIDTH/INPUT_TYPE::packed_words)
             should be a multiple of 2");
};
```

Buffer Type (BUFF_TYPE)	Class Name
AC_DUAL	ac_linebuffer_1r1w
AC_SPSHIFT	ac_linebuffer_spshift
AC_SPCIR	ac_line_buffer_spcircular
AC_SPWRMASK	ac_linebuffer_spwrmask

Class Methods

Top level linebuffer class contains a buffer method, data clear method and output access method as described in the table below.

Method	Description

`void run(bool we,`
`INPUT_TYPE din,`
`addrType addr,`
`OUTPUT_TYPE &dout)`
	Class interface run function. This should be called every time the design receives a data.
	Boolean we (Write Enable) to enable disable writing data into the buffer.
	Data interface din of type INPUT_TYPE where INPUT_TYPE can be of packed or unpacked ac data type.
	Address interface addr of addrType (ac_int<ac::log2_ceil<AC_WIDTH/INPUT_TYPE::packed_words>::val,false>)
	Data Interface dout, of type OUTPUT_TYPE which can be of packed array type of unpacked array type.

`void clear()`	Clear method, which can be called to clear all the data from the linebuffer.

Calling Top Level Design

The test bench built to test the line buffer class illustrates how the class is instantiated and the values are fed into the design.

1. Include the required line buffer header.

```
#include <ac_ipl/ac_linebuffer.h>
```

2. Instantiate the ac_linebuffer class with required template values as above.

```
typedef ac_linebuffer<INPUT_TYPE, OUTPUT_TYPE,
                      IMAGE_WIDTH, FILTER_HEIGHT,
                      BUFFER_TYPE, AC_REPEAT_REQ> line_buff_type;
```

3. Define the Design top.

```
#pragma hls_design top
void DesignTop(bool we, INPUT_TYPE din, line_buff_type::addrType addr,OUT-
PUT_TYPE &dout){
 static line_buff_type dut;
     dut.run(we,din,addr,dout);
}
```

4. Declare the input and output variables described in the pin level details.

```
INPUT_TYPE din;                 // Input variable.
OUTPUT_TYPE dout;             // Output Variable.
line_buff_type dut;             // Buffer class Instanace
line_buff_type::addrType addr;  // Address of the line buffer element.
```

5. Clear the line buffer contents using the clear access method if required.

```
dut.clear();          // Clear the Line buffer using the clear method.
```

6. Feed the input variable with suitable values and read out the values from the output port.

```
  bool wr_en=true;
// Send the packed data into the design.
  for (int frame=0; frame<NUM_FRAMES; frame++) {
    for (int i=0; i<IMAGE_HEIGHT; i++) {
      for (int j=0; j<(IMAGE_WIDTH/AC_BUS_WORDS); j++) {
```

```
        addr = j;
        if (frame==NUM_FRAMES-1) {

          for (int k=0; k<AC_BUS_WORDS; k++) {
            din[k]=(input_base_typ)0;
          }
          wr_en=false; // Disable write enable on the last frame.
        } else {
          din=cpp_packed_array[i][j];
          //A pre built input pattern in the array is fed into the input
          wr_en=true; // Keep the write enabled for rest of the frames.
        }
        CCS_DESIGN(DesignTop)(wr_en,din,addr,dout);
        #if  defined(DEBUG) && !(__SYNTHESIS__)
        cout << "------ Output -------" << endl ;
        for (int f=0; f<FILTER_HEIGHT; f++) {
          cout << dout[f] << endl ;
        }
        cout << endl;
        #endif
      }
    }
}
```

Catapult Architectural Exploration Options

The toolkit design as a whole is pipelined with an II of 1. All sub loops in all 4 buffer types are set to be unrolled and II=1, the only caution to be observed while we build this toolkit is to make sure the BLOCK_SIZE of the memory mapped should be set to suitable buffer width value.

```
For AC_SPSHIFT and AC_SPCIR BLOCK_SIZE=AC_WIDTH(buffer width)/(2xAC_BUS_WORDS)
For AC_DUAL BLOCK_SIZE=AC_WIDTH(buffer width)/AC_BUS_WORDS
```

This can be set in the constraints editor as highlighted below or with tcl directives as illustrated

```
set IMG_WIDTH  1280
set WORDS_PACKED 2

#for dual port design
set BLOCKSIZE [expr $IMG_WIDTH / $WORDS_PACKED]
set BLKSIZ_DIR "${BLOCKSIZE}"

#for single port design
set BLOCKSIZE [expr $IMG_WIDTH / $WORDS_PACKED]
set BLOCKSIZE_2 [expr $BLOCKSIZE / $DIV_2]
set BLKSIZ_DIR "${BLOCKSIZE_2}"
```

```
directive set /DesignTop/core/dut.lb.line_buffer.data:rsc -BLOCK_SIZE
$BLKSIZ_DIR
```

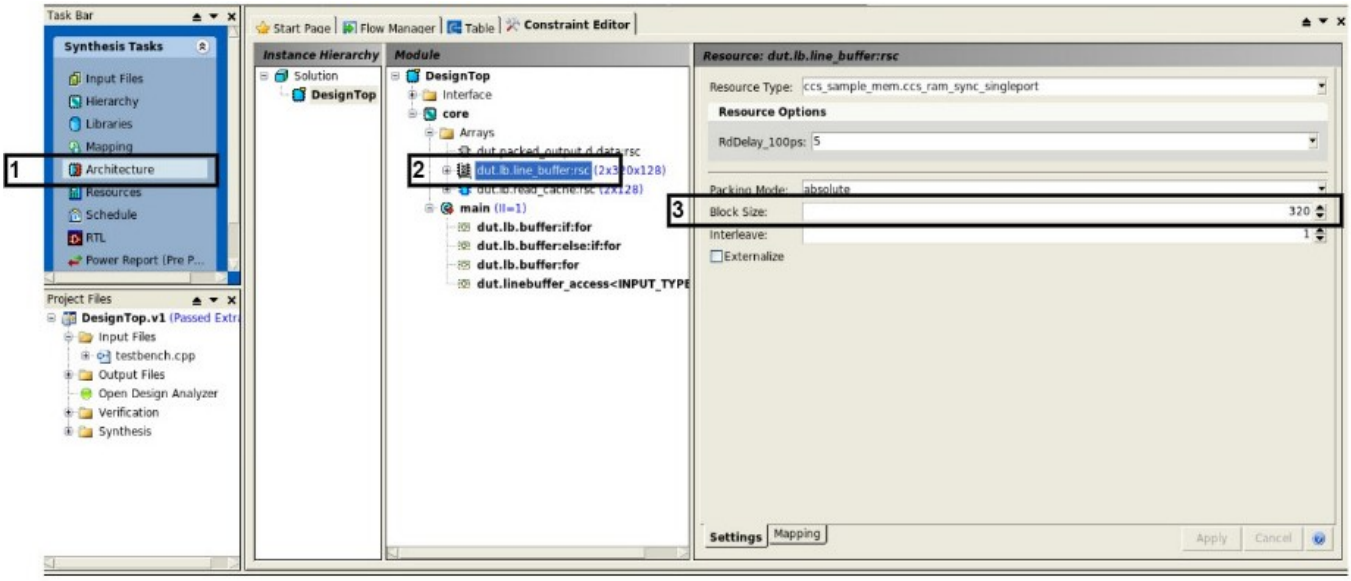

Image 3: Constraints editor illustration

2.11.2. Limitations

The following requirements must be kept in mind while using the design and calling the *run()* function. Failure to do so can result in erroneous functioning.

- For dualport memory mappings, the image width is expected to be a multiple of the words packed in the ac_packed vector input. For singleport memory mappings, the image width is expected to be divisible by twice the number of packed words.

- Further the design expects to have number of data packed in the ac_packed_vector (AC_BUS_WORDS) should be positive.

2.12. Packed Vector Type (ac_packed_vector)

The packed vector type is a container class designed to support image processing blocks that require multiple data to be streamed in parallel. The type concatenates the data into a single ac_int variable, which guarantees that the width of every element of an array of packed vector values is the final packed bitwidth rather than the width of the base data-type, which makes mapping to memories more straightforward and less prone to scheduling failures.

2.12.1. C++ Code Overview

A snippet of the class is given below.

```
template<typename T, int AC_WORDS>
class ac_packed_vector
{
public:
  static_assert(AC_WORDS > 0, "AC_WORDS must be positive.");
  typedef T base_type;
  enum {
    packed_words = AC_WORDS,
    width = T::width*AC_WORDS
  };

  // Class methods

// "data" member is public if using SCVerify.
#ifdef CCS_SCVERIFY
public:
#else
private:
#endif
  ac_int<T::width*AC_WORDS, false> data;
};
```

The “data” member variable is only set to be publicly accessible to render the type compatible with SCVerify. The user should not access the member variable directly. It must only be accessed using the class methods.

The base_type typedef as well as the packed_words and width enums are set to be public so that the user can use them in their code to extract information about the packed vector type. An example of the same is as follows:

```
typedef ac_packed_vector<ac_int<16, false>, 3> pv_type;
typedef typename pv_type::base_type pv_base_type; // Store base type in typedef
// Store packed_words and width in integer constants.
constexpr int pv_words = pv_type::packed_words;
constexpr int pv_width = pv_type::width;
```

Template Parameters

- T – Specifies the base type. This type must have a compile-time constant “width” that specifies its bitwidth and can be accessed using T::width. It must also support the slc() and set_slc() methods like ac_int and ac_fixed types currently do.

- AC_WORDS – Number of words that are packed in parallel. This influences the width of the “data” member variable. This value must be positive, failing which a static assertion will be triggered.

Constructors

Constructor	Description

ac_packed_vector()	Default constructor.
ac_packed_vector (T cpp_arr_in[AC_WORDS])	Constructs with a C++ array of type T and length AC_WORDS. Calls the pack_data() class method.
ac_packed_vector (const T cpp_arr_in[AC_WORDS])	Same as above but for a constant C++ array.
ac_packed_vector (ac_array<T, AC_WORDS> ac_arr_in)	Constructs with an AC array of type T and length AC_WORDS. Also calls pack_data().
ac_packed_vector(const T &scalar)	Sets all elements to input scalar, which is of type T.
ac_packed_vector(const c_type scalar)	Sets all elements to an input scalar of a C type. Refer to Scalar Initialization and Copying for more details.

Subscript Operator Overloading

Method	Description
helper_class operator[] (unsigned idx)	Used to read or write elements of a non-const packed vector. If reading elements, it should be used with the conversion operator overloading found in the nested helper_class. If writing elements, it is used with the assignment operator overloading of the nested class. More details on the same can be found in helper_class Methods and Accessing Elements. Will assert if idx >= AC_WORDS.
const T operator[] (unsigned idx) const	Used to read elements of a const packed vector. Will assert if idx >= AC_WORDS.

Assignment Operator Overloading

Method	Description
void operator= (T cpp_arr_in[AC_WORDS])	It is used to assign a non-const C++ array of type T and length AC_WORDS to a packed vector. It calls the pack_data() method.
void operator= (const T cpp_arr_in[AC_WORDS])	Same as above but for a const C++ array.
void operator= (ac_array<T, AC_WORDS> ac_arr_in)	It is used to assign an AC array of type T and length AC_WORDS to a packed vector. It also calls pack_data().
void operator= (const T &scalar)	Assigns an input scalar of the base type to all the elements.
void operator= (const c_type scalar)	Assigns an input scalar of a C type to all the elements. Refer to Scalar Initialization and Copying for more details.

helper_class Methods

The packed vector type also has a nested helper_class to aid in accessing elements of a non-const packed vector. Except for the conversion operator overloading, these methods are only meant to be used by other methods in ac_packed_vector and must not be accessed directly. The various methods of the helper class are described below.

Method	Description

helper_class(ac_packed_vector *c, ac_int<ac::nbits<AC_WORDS - 1>::val, false> idx) : packed_vector(*c), index(idx) {}	Constructs an object of the helper class by setting the internal packed vector reference to the first constructor argument, and setting the internal index value to the second constructor argument.
operator T() const	Conversion operator overloading. The user must use this when trying to read elements of a packed vector. More details are given in Accessing Elements.
helper_class operator= (T val)	Scalar assignment. This is used in conjunction with the subscript operator overloading in ac_packed_vector to write packed vector elements. The scalar can only be of type T.
helper_class operator= (ac_packed_vector<T,AC_WORDS>:: helper_class other_hc_obj)	Assignment operator overloading. This is used to assign elements between two packed vectors of type T and length AC_WORDS.

Miscellaneous Methods

Method	Description
void pack_data (const T cpp_arr_in[AC_WORDS])	Packs a C++ array of type T and length AC_WORDS into the underlying data. Data packing happens in an unrolled loop.
void pack_data (ac_array<T, AC_WORDS> ac_arr_in)	Packs an AC array of type T and length AC_WORDS into the underlying data. Data packing happens in an unrolled loop.
void unpack_data (T cpp_arr_out[AC_WORDS]) const	Unpacks the underlying data to a C++ array. The unpacking happens in an unrolled loop.
void unpack_data(ac_array<T, AC_WORDS> &ac_arr_out) const	Unpacks the underlying data to an AC array. The unpacking happens in an unrolled loop.
void set_data(ac_int<T::width *AC_WORDS, false> data_in)	Sets underlying data to function input.
ac_int<T::width*AC_WORDS, false> get_data() const	Returns underlying data.
void set_max()	Sets underlying data to the max possible value, i.e. all 1s.
template<int WS> inline const ac_int<WS, false> slc(signed index) const	Slices read method which operates on the underlying data. "WS" is the width of the slice, while "index" is the position of the slice LSB.
template<int W2, bool S2> inline ac_packed_vector &set_slc(signed lsb, const ac_int<W2,S2> &slc)	Slice write method which operates on the underlying data. "lsb" is the slice LSB, while "slc" is the ac_int slice assigned to the underlying data.
template <typename T2> void cpy_from_scalar(const T2 &scalar)	Copies the input scalar to all elements of the packed vector. Refer to Scalar Initialization and Copying for more details.
void reset()	Resets underlying data to 0.
static std::string type_name()	Returns the name of the type as an std::string. For more information, refer to Using the type_name() Method. If you're using this, you must make sure that T defines type_name() as a static method too.

2.12.2. Accessing Elements

There are special considerations to be taken while reading and writing packed vector elements. Let us consider write and read operations on a packed vector element using the subscript operator:

```
typedef ac_fixed<32, 16, false> T;
const int AC_WORDS = 10;
// packed vector is declared below.
ac_packed_vector<T, AC_WORDS> pv;
for (int i = 0; i < AC_WORDS; i++) {
  pv[i] = T(i); // Write i to packed vector element.
  // Read and display packed vector element.
  std::cout << T(pv[i]) << std::endl;
}
```

Consider the write operation:

```
pv[i] = T(i);
```

This can also be written as follows:

```
(pv.operator[](i)).operator=(T(i));
```

As explained in Subscript Operator Overloading, pv.operator[](i) will return a helper_class object, and the assignment operator (.operator=(T(i))) called is hence the scalar assignment operator described in helper_class Methods. Since the helper_class scalar assignment only accepts objects of the base type, we must typecast i—an int variable—to T. The following is also permitted:

```
T pv_elem_val = i;
pv[i] = pv_elem_val; // Scalar on RHS is of the packed vector base type.
```

The following is NOT permitted:

```
pv[i] = i; // i is not of type T and this assignment is invalid.
```

Now, consider the read operation used to display the packed vector elements:

```
T(pv[i])
```

This can also be written as:

```
(pv.operator[](i)).operator T()
```

Similar to the write operation explained above, (pv.operator[](i)) will also return a helper_class object. The operator T() conversion operator called is hence the conversion operator described in helper_class Methods. The final output of the read operation will hence be a scalar of type T that can be printed out using the ostream object “cout”, provided that class T has the ostream << operator overloading required to print out its data (the ac_fixed class does). The following would also achieve a similar purpose:

```
T pv_elem_val = pv[i]; // Conversion operator called while assigning to LHS.
std::cout << pv_elem_val << std::endl; // Printing an ac_fixed variable now.
```

The following, however, is NOT permitted:

```
std::cout << pv[i] << std::endl; // helper_class object cannot be printed.
```

You also need an explicit conversion stage if, say, you're trying to add or multiply packed vector elements:

```
T sum = T(pv[0]) + T(pv[AC_WORDS – 1]); // Using ac_fixed addition now.
T prod = T(pv[0]) * T(pv[AC_WORDS – 1]); // Using ac_fixed multiplication now.
```

This is, of course, assuming that type T already has support for these arithmetic operations.

The following is NOT permitted:

```
T sum = pv[0] + pv[AC_WORDS–1]; // helper_class does not have addition support.
```

2.12.3. Using the type_name() Method

Similar to the AC Datatypes (ac_int, ac_fixed, etc), ac_packed_vector also has a type_name() method, described in Miscellaneous Methods. Consider the following snippet:

```
typedef ac_packed_vector<ac_int<32, false>, 12> pv_type;
// Since the type_name() method is declared static, you can access it using
// the typedef above and the scope resolution operator.
std::cout << "pv_type : " << pv_type::type_name() << std::endl;
```

The output of the snippet is as follows:

```
pv_type : ac_packed_vector<ac_int<32,false>,12>
```

2.12.4. Pretty Printing

The contents of a packed vector can also be pretty-printed using the ostream << operator, with the overloading below:

```
template<typename T, int AC_WORDS>
std::ostream &operator<<(std::ostream &os, ac_packed_vector<T, AC_WORDS> input)
```

The pretty printing is done following the format below:

```
{input[0], input[1], input[2], input[3], ... input[AC_WORDS - 1]}
```

Consider, for instance, the snippet below:

```
ac_packed_vector<ac_int<8, false>, 5> pv;
for (int i = 0; i < 5; i++) {
```

```
  pv[i] = ac_int<8, false>(i*3); // Remember to convert to base type!
}
std::cout << "pv = " << pv << std::endl;
```

The output of the snippet is as follows:

```
pv = {0, 3, 6, 9, 12}
```

2.12.5. Scalar Initialization and Copying

If the user wishes to set all the elements of a packed vector to a certain scalar value, they can use the cpy_from_scalar method described in Miscellaneous Methods. cpy_from_scalar accepts inputs of any scalar type as long as a T2 (the type of the scalar value) -> T conversion is supported. The snippet below denotes an example with supported and unsupported scalar copying.

```
// T : ac_int<8, true>, AC_WORDS : 3
typedef ac_packed_vector<ac_int<8, false>, 5> pv_type;
pv_type pv;

// Supported, since ac_int -> ac_int conversions are allowed.
pv.cpy_from_scalar(ac_int<4, false>(4));

ac_fixed<4, 4, false> fxpt_val = 4;
// NOT supported, since ac_fixed -> ac_int conversions are NOT allowed.
pv.cpy_from_scalar(fxpt_val);
// Supported, since you're converting ac_fixed to ac_int first.
pv.cpy_from_scalar(fxpt_val.to_ac_int());
```

The scalar constructors and assignment operators described in Constructors and Assignment Operator Overloading allow scalar initialization/copying from values of the base type as well as all C types (except for long double) provided that conversion from a given C type to T is supported. All the scalar constructors and assignment operators use the cpy_from_scalar method internally, to do the actual copying. The following snippet shows how to use scalar constructors and assignments:

```
// T : ac_int<8, true>, AC_WORDS : 3
typedef ac_packed_vector<ac_int<8, false>, 5> pv_type;
pv_type pv1 = 2; // Constructed with scalar value 2.
pv_type pv2;
pv2 = 2.5; // Assigning 2.5 to all elements.
```

2.13. Synchronization Flag Generator (ac_flag_gen)

The ac_flag_gen library generates synchronization flags that indicate the start of line (SOL), end of line (EOL), start of frame (SOF) and end of frame (EOF) for an image whose dimensions are known ahead of time. This library has two classes ac_flag_gen_1d and ac_flag_gen_2d, which generate synchronization flags in one and two dimensions, respectively.

2.13.1. C++ Code Overview

A snippet of the ac_flag_gen_1d class is given below.

```
template<int AC_WIDTH, int AC_BUS_WORDS = 1>
class ac_flag_gen_1d
{
public:
  static_assert(AC_WIDTH > 0, "AC_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");
  static_assert(AC_WIDTH%AC_BUS_WORDS == 0, "AC_WIDTH must be perfectly divisi-
ble by AC_BUS_WORDS.");

  typedef ac_int<ac::nbits<AC_WIDTH>::val, false> W_TYPE;

  // Class methods

private:
  ac_int<ac::nbits<AC_WIDTH - 1>::val, false> cnt;
  W_TYPE line_width;
  bool incorrect_dim_change;
};
```

The cnt member variable is responsible for counting the number of pixels and gets reset to zero after the EOL pixel. The line_width variable is updated at the start of every line and is compared with the width dimension input to the run() function for every pixel that isn't an SOL pixel. This is done to check for unexpected changes to the dimension input. If the dimension input does change unexpectedly, the incorrect_dim_change flag is set to true. More details are given in ac_flag_gen_1d Checking. W_TYPE is the type for the width input to the run() function.

```
template<int AC_WIDTH, int AC_HEIGHT, int AC_BUS_WORDS = 1>
class ac_flag_gen_2d
{
public:
  static_assert(AC_WIDTH > 0, "AC_WIDTH must be positive.");
  static_assert(AC_HEIGHT > 0, "AC_HEIGHT must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  typedef ac_int<ac::nbits<AC_WIDTH>::val, false> W_TYPE;
  typedef ac_int<ac::nbits<AC_HEIGHT>::val, false> H_TYPE;

  // Class methods

private:
  ac_flag_gen_1d<AC_WIDTH, AC_BUS_WORDS> line;
  ac_int<ac::nbits<AC_HEIGHT - 1>::val, false> cnt;
```

```
  bool incorrect_dim_change;
  W_TYPE frame_width;
  H_TYPE frame_height;
};
```

The ac_flag_gen_2d class is built on top of the ac_flag_gen_1d class and uses an object of the latter as a member variable, which in turn computes the SOL and EOL flags. The cnt member variable is responsible for counting the number of lines received and gets reset to zero after the EOF pixel. The frame_width and frame_height variables are updated at the start of every frame and are compared with the width and height dimension inputs to the run() function, respectively, for every pixel that isn't an SOF pixel. This is done to check if one or both of the dimension inputs changed unexpectedly. If the dimension input(s) did change unexpectedly, the incorrect_dim_change flag is set to true. More details are given in ac_flag_gen_2d Checking. W_TYPE and H_TYPE are the types for the width and height input to the run() function, respectively.

ac_flag_gen_1d Template Parameters

- AC_WIDTH – Specifies the maximum line width.
- AC_BUS_WORDS – Number of pixels that are sent in parallel over the bus interface. It is set to 1 by default.

Both these template parameters must be positive. AC_WIDTH must also be perfectly divisible by AC_BUS_WORDS. If any of these conditions are violated, a static assertion is triggered.

ac_flag_gen_2d Template Parameters

- AC_WIDTH – Specifies the maximum frame width.
- AC_HEIGHT – Specifies the maximum frame height.
- AC_BUS_WORDS – Number of pixels that are sent in parallel over the bus interface. It is set to 1 by default.

Similar to the ac_flag_gen_1d Template Parameters, these parameters must also be positive and AC_WIDTH must also be perfectly divisible by AC_BUS_WORDS, with static assertions being added to guard against any violation of these conditions.

Constructors

Constructor	Description
ac_flag_gen_1d()	Default ac_flag_gen_1d constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_gen_1d Methods.
ac_flag_gen_2d()	Default ac_flag_gen_2d constructor. Does the following: (a) Sets cnt, frame_width and frame_height to 0. (b) Sets incorrect_dim_change to false.

Miscellaneous ac_flag_gen_1d Methods

Method	Description
void run (W_TYPE width, bool &sol, bool &eol)	Class interface run function. This should be called every time the design receives a pixel. The width dimension input specifies the width of the line. The sol and eol flag outputs are set to true when the SOL and EOL pixels are received, respectively.
void reset()	Resets class member variables: (a) Sets cnt and line_width to 0. (b) Sets incorrect_dim_change to false.
bool dim_change_error()	Returns the incorrect_dim_change member variable. More details on this variable are given in ac_flag_gen_1d Checking.

Miscellaneous ac_flag_gen_2d Methods

Method	Description
void run (W_TYPE width, H_TYPE height, bool &sof, bool &eof, bool &sol, bool &eol, bool only_update_line = false)	Class interface run function. This should be called every time the design receives a pixel. The vld flag input specifies whether the pixel received was valid or not. The width and height dimension inputs specify the width and height of the frame, respectively. The width input must be perfectly divisible by AC_BUS_WORDS, failing which an AC_ASSERT is triggered. The sof, eof, sol and eol flag outputs are set to true when the SOF, EOF, SOL and EOL pixels are received, respectively. The only_update_line input should be set to true when the user only wants to update the SOL and EOL flags, which is the case, for instance, in windowing examples when we're flushing outputs.
void reset()	Resets class member variables: (a) Sets cnt, frame_width and frame_height to 0. (b) Sets incorrect_dim_change to false. (c) Calls the reset method of the line member variable.
bool dim_change_error()	Returns the incorrect_dim_change member variable. More details on this variable are given in ac_flag_gen_2d Checking.

2.13.2. Incorrect Dimension Checking

As mentioned earlier, the run() functions for both ac_flag_gen_1d and ac_flag_gen_2d are meant to be called every time a design receives a pixel. They also accept dimension inputs, and there's a chance that these inputs could change incorrectly in between function calls. The following sections describe how ac_flag_gen_1d and ac_flag_gen_2d handle notifying the user of incorrect dimension changes.

ac_flag_gen_1d Checking

The first method is to throw a software assert whenever the width is changed mid-line. The following snippet highlights how and when this assert is triggered in the run() function:

```
    bool sol_temp = (cnt == 0);
```

```
    if (sol_temp) {
      line_width = width;
    } else {
      incorrect_dim_change = (width != line_width);
      #ifndef DISABLE_AC_FLAG_GEN_ASSERT
      AC_ASSERT(!incorrect_dim_change, "Dimension input changed unexpectedly.
Make sure width only changes when SOL pixel is received.");
      #endif
    }
```

For every valid pixel that isn't an SOL pixel, the run() function compares the width input against the line_width member variable that was updated at the SOL. If the two aren't the same, the incorrect_dim_change flag is set to false and an AC_ASSERT is triggered. This assert can be disabled by defining the DISABLE_AC_FLAG_GEN_ASSERT macro, which a user might want to do if they want to test the functionality of the dim_change_error() method.

The software assert will not catch dimension change errors in hardware, which is where the dim_change_error() method comes in. This method returns the incorrect_dim_change flag, which means that it returns true whenever the width changes incorrectly. The user should call the method right after they call the run() function, like so:

```
// Instantiate ac_flag_gen_1d.
ac_flag_gen_1d<1024> fg_1d_obj;
int width = 512; // Assume that line width is 512.
for (int i = 0; i < width; i++) {
  bool sol, eol;
  fg_1d_obj.run(width, sol, eol); // Call run() function.
  if (fg_1d_obj.dim_change_error()) {
    // Add hardware handling for your error here.
  }
  // Carry out your regular image processing.
}
```

ac_flag_gen_2d Checking

Similar to the ac_flag_gen_1d Checking, ac_flag_gen_2d also lets the user check for incorrect dimensions both through a software assert and through the dim_change_error() method. The snippet containing the assert is as follows:

```
    bool sol_temp, eol_temp;
    line.run(width, sol_temp, eol_temp);
    bool sof_temp = !only_update_line && (cnt == 0) && sol_temp;

    if (sof_temp) {
      frame_width = width;
      frame_height = height;
    } else {
```

```
      incorrect_dim_change = (width != frame_width) || (height !=
frame_height);
      #ifndef DISABLE_AC_FLAG_GEN_ASSERT
      AC_ASSERT(!incorrect_dim_change, "Dimension input(s) changed unexpect-
edly. Make sure the dimensions only change when SOF pixel is received.");
      #endif
    }
```

As can be seen, the run() function in ac_flag_gen_1d is called first. As a result, if the width changes mid-line, the ac_flag_gen_1d assert will be triggered. However, if (a) the height input changes on any pixel that isn't an SOF pixel or (b) the width input has changed on an SOL pixel which is not also an SOF pixel, the ac_flag_gen_2d assert will be triggered instead.

Just like in ac_flag_gen_1d, ac_flag_gen_2d also lets you incorporate hardware error checking by using the dim_change_error() method to return the incorrect_dim_change flag. An example snippet for the same is shown below:

```
// Instantiate ac_flag_gen_2d.
ac_flag_gen_2d<1024, 1024> fg_2d_obj;
int width = 512, height = 384; // Assume 512x384 image.
for (int i = 0; i < height; i++) {
  for (int j = 0; j < width; j++) {
    bool sof, eof, sol, eol;
    // Call run() function.
    fg_2d_obj.run(width, height, sof, eof, sol, eol);
    if (fg_2d_obj.dim_change_error()) {
      // Add hardware handling for your error here.
    }
    // Carry out your regular image processing.
  }
}
```

Chapter 3: ac_window Classes

The ac_window classes provided with Catapult were designed to allow the user to quickly implement one-dimensional and two-dimensional algorithms without having to explicitly describe the underlying memory architecture. The ac_window classes are optimized C++ libraries that have been designed to give highly optimized hardware while providing the user with a usage model that is similar to the way algorithms are often generically described.

Example designs that use the ac_window classes are provided in the Catapult Toolkits. To access the Toolkits, choose the **Help > Toolkits...** menu item.

NOTE: *If the Toolkits menu item does not appear on the Help menu, it means that the Toolkits option was not installed. Use the Catapult Setup Wizard to reconfigure the Catapult installation and install the Toolkits.*

The ac_window class library provides several window implementations for both 1-d and 2-d algorithms. The different implementations provide the ability to re-write algorithms using several different styles. The window classes solve the implementation bandwidth problem described above by allowing only a single array access on the design interface while "windowing" this input data into registers or memories which keeps a running history of array elements. The algorithm then operates on the stored or "windowed" data to achieve maximum performance.

All of the windowing classes expect the array data to be read sequentially and provided to the window variable in order. Thus these windowing classes would not be useful for applications that access the array data in a zigzag fashion such as motion estimation. For these types of applications a different type of windowing class is required.

The following list describes the known limitations of the ac_window classes:

- 2-d windows will not exactly match reference design when the row and column width are changed dynamically. There will be mismatches after switching the row or column size while the old values stored in the window are flushed. Once the window flushes, the data will match the reference exactly.
- Ignore_memory_precedence must be used on 2-d windows if pipeline is set to II=1. Does not apply when using true singleport RAM and the AC_SINGLEPORT mode.

The following five ac_window class implementations are described in this section:

- ac_window_1d_flag Class
- ac_window_2d_flag Class
- ac_window_2d_flag_flush_support Class
- ac_window_1d_stream Class
- ac_window_2d_stream Class

3.1. ac_window_1d_flag Class

The ac_window_1d_flag class provides a 1-d window to allow algorithms to be written so that data is both read and written continuously. This class relies of control flags to indicate the start and end of array data. These input control flags are used by the class to control when clipping or mirroring happens. The class is defined in ac_window_1d_flag.h and is defined as:

```
template<class T, int AC_WN, ac_window_mode AC_WMODE = AC_WIN>
class ac_window_1d_flag {
public:
     ac_window_1d_flag();
     T & operator[]      (int i);
     const T & operator[](int i) const ;
     void write(T src, bool sol, bool eol);
     void readFlags(bool &solOut, bool &eolOut) ;
     bool valid();
```

ac_window_1d_flag Template Parameters

- **T** - Specifies the input/output data type.
- **AC_WN** - Specifies the window size.
- **AC_WMODE** - Specifies the mode for handling the array boundaries. Valid parameters are listed below.
    - **AC_WIN** - No boundary handling.
    - **AC_MIRROR** - Mirroring used on the boundary.
    - **AC_CLIP** - Clipping used on the boundary.

ac_window_1d_flag Member Functions

- **ac_window_1d_flag()** - Constructor resets valid flag to false.
- **T & operator[](int i)** - Returns the window value indexed by "i". Currently this value must be specified as -AC_WN/2 < I < AC_WN/2, where AC_WN is the window size.
- **void write(T src, bool sol, bool eol)** - Writes data and control flags into the window. The "sol" argument indicates start of line and "eol" indicates end of line. The "sol" argument controls when the left side of the array is mirrored or clipped and "eol" controls when the right side of the array is mirrored or clipped.

- v**oid readFlags(bool &solOut, bool &eolOut)** - Reads the output control flags of the window. These flags are delayed versions of the input control flags. They are delayed by AC_WN/2 clock cycles, which is the amount of time needed to pre-fill the window.

- **bool valid()** - Returns true when the window pre-fill has completed. NOTE: Valid() returns true after AC_WN/2 iterations for mirroring and clipping modes and returns true for AC_WN iterations for AC_WIN mode.

Using the ac_window_1d_flag Class

This class typically requires a little more rewriting of the C++ and test bench because of the use of control flags. A generic algorithmic description may look like:

```
#include "test.h"
int mirror(int i, int sz){
     return ((i<0)?(i*-1):((i<sz)?i:2*sz-i-2));
}

void window_ref ( int sz, uchar linei[MAXSZ], uchar lineo[MAXSZ] )
{
     for (int i=0; i<sz; i++) {
        lineo[i] = (linei[mirror(i-2, sz)] +
                    linei[mirror(i-1, sz)] +
                    linei[mirror(i  , sz)] +
                    linei[mirror(i+1, sz)] +
                    linei[mirror(i+2, sz)]
                    );
     }
}
```

In the above example the design loops over all the input data and applies mirroring on the boundaries. Examination of the original design shows that the processing window spans 5 samples. Thus a window with AC_WN=5 is required.

The ac_window_1d_flag implementation is shown below. A struct was created to group the data and control on the interface. Note that the "for" loop has been removed from the design. The test bench must be modified to call the function until all the data has been processed. An example test bench is provided to show how this is done.

```
typedef struct{
     uchar data;
     ac_int<1,false> sof;
     ac_int<1,false> eof;
     ac_int<1,false> sol;
     ac_int<1,false> eol;
}    dataWithFlags;
```

```
#pragma design top
void window_ccs (dataWithFlags &din, dataWithFlags &dout  )
{
      //window must be static
      //window size set to 5 with mirroring
      static ac_window_1d_flag<uchar, 5,AC_MIRROR> w;
      //write inputs
      w.write(din.data,din.sol,din.eol);
      // don't output till pre-fill done
      if(w.valid())
         dout.data = ( w[-2] + w[-1] + w[+0] + w[+1] + w[+2] );
      //read window output flags
      w.readFlags(dout.sol,dout.eol);
}
```

The sol/eol Timing

The start-of-line (sol) and end-of-line (eol) input controls coincide with the first and last input array elements. The sol and eol generated by the window class are delayed based on the window MODE. For Mirroring and clipping this delay is equal to AC_WN/2 where AC_WN is the window size.

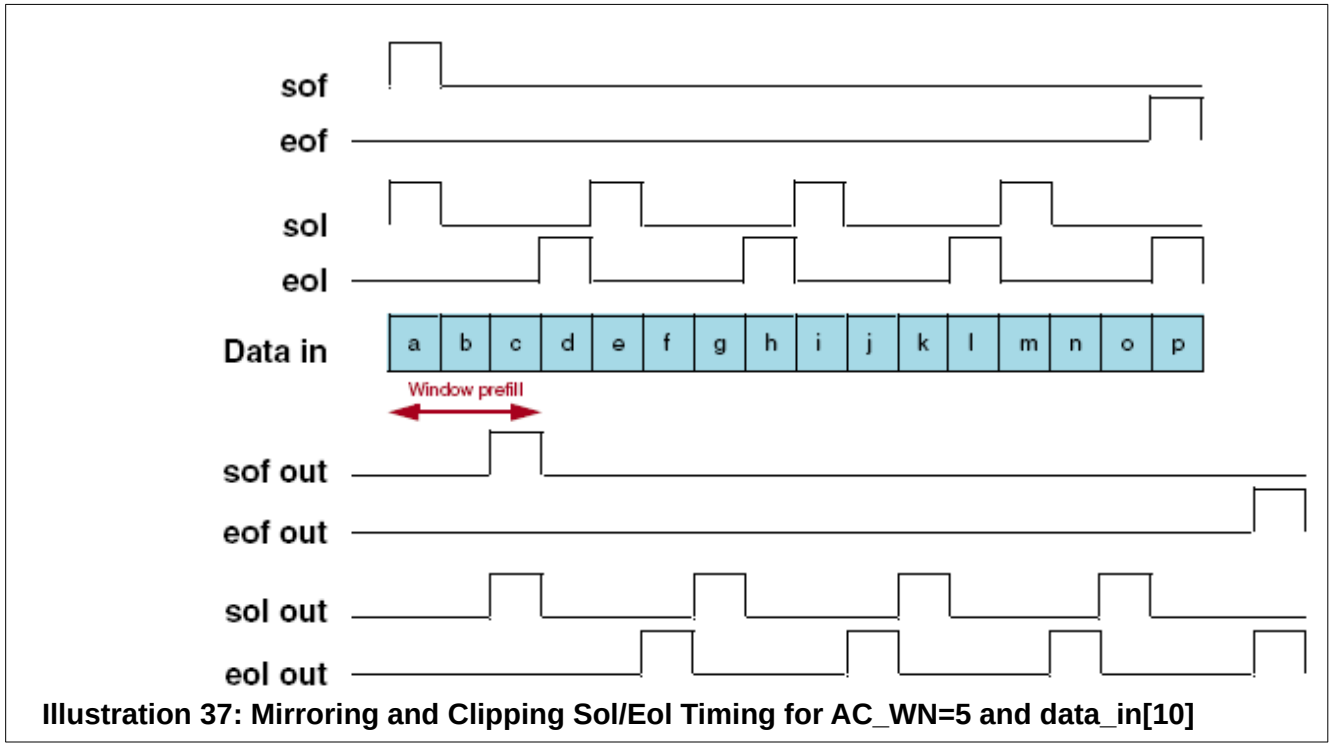

Illustration 37: Mirroring and Clipping Sol/Eol Timing for AC_WN=5 and data_in[10]

3.2. ac_window_2d_flag Class

The ac_window_2d_flag class provides a 2-d window to allow algorithms to be written so that data is both read and written continuously. This class relies on control flags to indicate the start and end of both frames

and lines of data. These input control flags are used by the class to control when clipping or mirroring happens. The 2-d window differs from the 1-d window in that it usually will contain both registers and memories. The memories are used to store whole rows of array data. This type of memory architecture is typical for video processing where lines of pixel data must be stored in RAM. The class is defined in ac_window_2d_flag.h and is defined as:

```
template<class T, int AC_WN_ROW, int AC_WN_COL, int AC_NCOL, int AC_WMODE>
    class ac_window_2d_flag{
    public:
    ac_window_2d_flag();
    ac_window_2d_flag(T bval);
    T & operator()      (int r,int c);
    const T & operator()(int r, int c) const ;
    void write(T src, bool sof, bool eof, bool sol, bool eol);
    void exec(T src, bool sof, bool eof, bool sol, bool eol, bool w);
    bool valid();
    void readFlags(bool &sof, bool &eof, bool &sol, bool &eol);
    void rewind();
```

ac_window_2d_flag Template Parameters

- **T** - Specifies the input/output data type. If this type is user-defined, then the user must include a parameterized constructor that accepts an int argument in the type definition, so as to be compatible with typecasting done in ac_buffer_2d.h, i.e. typecasting to the integer literal “0”. A default constructor must also be explicitly defined. An example for a user-defined type is given in the snippet below.

  ```
  struct user_def_type {
    ac_int<16, false> data[3][3]; // Actual data: 3x3 ac_int array.
    // Parameterized constructor: accepts int input and assigns it to all
    // elements of data array.
    user_def_type(int in) {
      #pragma hls_unroll yes
      INIT_TO_INT_ROW: for (int i = 0; i < 3; i++) {
        #pragma hls_unroll yes
        INIT_TO_INT_COL: for (int j = 0; j < 3; j++) {
          data[i][j] = in;
        }
      }
    }
    // Default constructor: Explicitly defined. This one’s empty but feel
    // free to modify your default constructor as appropriate.
    user_def_type() {}
  };
  ```

- **AC_WN_ROW** - Specifies the vertical window size.

- **AC_WN_COL** - Specifies the horizontal window size.

- **AC_WMODE** - Specifies the mode for handling array boundaries. Valid parameters are listed below.
  - **AC_WIN** - No boundary handling.
  - **AC_MIRROR** - Mirroring used on the boundary.
  - **AC_CLIP** - Clipping used on the boundary.
  - **AC_BOUNDARY** - Allows a constant boundary value to be specified in the constructor. (Instead of mirroring or clipping).
  - **AC_SINGLEPORT** - Use “true” singleport RAM. Singleport RAM is preferred over dualport RAM for ASIC designs due to lower area cost. This is only supported when using ac_int, ac_fixed, RGB_imd<ac_int> and RGB_imd<ac_fixed>. In order to add support for other types, the user must modify the template specialization of the ac_width2x struct, in the ac_window_structs.h header.
  - **AC_DUALPORT** - Use dualport RAM.
  - **AC_SYM_INDEX** - Use symmetrical index format specified as -AC_WN/2 < i < AC_WN/2, where AC_WN is the window size.
  - **AC_LIN_INDEX** - Use linear index format, such as i = 0 to 7. This format is more useful for windows with even dimensions.
  - **AC_REWIND** - Allows line buffer data to be used repeatedly by resetting the line buffer address pointers.
  - **AC_NO_REWIND** - Disable rewind option.

  To specify multiple parameters, *OR* them together in the template argument. For example:

```
static ac_window_2d_flag<
     ac_int<8,false> ,WN_SZ,WN_SZ,GRID,AC_MIRROR|AC_SINGLEPORT> w;
```

ac_window_2d_flag Member Functions

- **ac_window_2d_flag()** - Constructor resets valid flag to false and resets address counter that indexes window memories
- **T & operator()(int r,int c)** - Returns the window value indexed by "r,c". Currently this value must be specified as -AC_WN_ROW/2 < r < AC_WN_ROW/2 and -AC_WN_COL/2 < c < AC_WN_COL/2.
- **void write(T src, bool sof, bool eof, bool sol, bool eol)** - Writes data and control flags into the window. The "sof" argument indicates start of frame and "eof" indicates end of frame. The "sof" signal controls when the top row of the 2-d array is mirrored or clipped and "eof" controls when the bottom row of the 2-d row is mirrored or clipped. "sol" indicates start of line and "eol" indicates end of line.

"sol" controls when the left side of the array is mirrored or clipped and "eol" controls when the right side of the array is mirrored or clipped.

- **void readFlags(bool &sof, bool &eof, bool &sol, bool &eol)** - Reads the output control flags of the window. These flags are delayed versions of the input control flags. They are delayed by AC_WN/2 clock cycles which is the amount of time to pre-fill the window.

- **void exec(T src, bool sof, bool eof, bool sol, bool eol, bool write)** - Writes data and control flags into the window. The "sof" argument indicates start of frame and "eof" indicates end of frame. The "sof" signal controls when the top row of the 2-d array is mirrored or clipped and "eof" controls when the bottom row of the 2-d row is mirrored or clipped. "sol" indicates start of line and "eol" indicates end of line. "sol" controls when the left side of the array is mirrored or clipped and "eol" controls when the right side of the array is mirrored or clipped. “write” allows the buffers to be written when set to true. When “write” is false, the line buffer address pointers are advanced one position each time “exec” is called.

- **void rewind()** - Resets the window line buffer address to zero.

- **bool valid()** - Returns true when the window has pre-filled.

Using the ac_window_2d_flag Class

This class typically requires a little more rewriting of the C++ and test bench because of the use of control flags. A generic algorithmic description may look like:

```
#include <ac_int.h>
#include "sobel.h"
int clipo(int i, int sz){
     int tmp;
     if(i<0)
        tmp = 0;
     else if(i > sz-1)
        tmp = sz-1;
     else
        tmp = i;
     return tmp;
}

int mirror(int i, int sz){
     return ((i<0)?(i*-1):((i<sz)?i:2*sz-i-2));
}

int bounds(int i, int sz){
#ifdef MIRROR
     return mirror(i,sz);
#endif

#ifdef CLIP
     return clipo(i,sz);
```

```
#endif
}
int abso(int i){
     if(i<0)
        return -i;
     else
        return i;
}
const int cx[3][3] = {-1,0,1,-2,0,2,-1,0,1};
const int cy[3][3] = {1,2,1,0,0,0,-1,-2,-1};

#pragma design top
void sobel_orig(dType din[GRID][GRID], dType dout[GRID][GRID]){
     int Gx,Gy;
     int tmpx,tmpy;

     ROW:for(int i=0;i<GRID;i++){
        COL:for(int j=0;j<GRID;j++){
           tmpx = 0;
           tmpy = 0;
           for(int r=0;r<3;r++){
              for(int c=0;c<3;c++){
                 tmpx += din[bounds(i+r-1,GRID)]
                             [bounds(j+c-1,GRID)]*cx[r][c];
                 tmpy += din[bounds(i+r-1,GRID)]
                             [bounds(j+c-1,GRID)]*cy[r][c];
              }
           }
           Gx = tmpx;
           Gy = tmpy;
           dout[i][j] = abso(Gx) + abso(Gy);
        }
     }
}
```

In the above example the design loops over all the input data and applies mirroring on the left, right, upper, and lower boundaries. Examination of the original design shows that the processing window spans 3x3 samples. Thus a window with AC_WN=3 is required.

The ac_window_2d_flag implementation is shown below. A struct was created to group the data and control on the interface. Note that the "for" loop has been removed from the design. The test bench must be modified to call the function until all the data has been processed. An example test bench is provided to show how this is done.

```
const int cx[3][3] = {-1,0,1,-2,0,2,-1,0,1};
```

```
const int cy[3][3] = {1,2,1,0,0,0,-1,-2,-1};

int abs(int i){
     if(i<0)
        return -i;
     else
        return i;
}
#include "ac_window.h"
#pragma design top
void sobel(dataWithFlags &din, dataWithFlags &dout){
#ifdef CLIP
     static ac_window_2d_flag<int,WN_SZ,WN_SZ,GRID,AC_CLIP> w;
#endif
#ifdef MIRROR
     static ac_window_2d_flag<int,WN_SZ,WN_SZ,GRID,AC_MIRROR> w;
#endif
     bool sofOut;  // Start of frame output control
     bool eofOut;  // End of frame output control
     bool solOut;  // Start of line output control
     bool eolOut;  // End of line output control
     int Gx,Gy;
     int tmpx,tmpy;
     tmpx = tmpy = 0;
     //write all data and control inputs
     w.write(din.data,din.sof,din.eof,din.sol,din.eol);
     if(w.valid()){
        for(int r=0;r<WN_SZ;r++){
           for(int c=0;c<WN_SZ;c++){
              tmpx += w(r-WN_SZ/2,c-WN_SZ/2)*cx[r][c];
              tmpy += w(r-WN_SZ/2,c-WN_SZ/2)*cy[r][c];
           }
        }
     }

     Gx = tmpx;
     Gy = tmpy;
     w.readFlags(sofOut,eofOut,solOut,eolOut); //read output control flags
     dout.data = abs(Gx) + abs(Gy);
     dout.sof = sofOut;
     dout.eof = eofOut;
     dout.sol = solOut;
     dout.eol = eolOut;
}
```

The sof/eof/sol/eol Timing

The start-of-frame (sof) and end-of-frame (eof) controls coincide with the first and last elements of the 2-d input array. The start-of-line (sol) and end-of-line (eol) input controls coincide with the first and last elements of each array row. The sof, eof, sol and eol generated by the window class are delayed based on the window MODE. For Mirroring and clipping this delay is equal to WN/2 where WN is the window size.

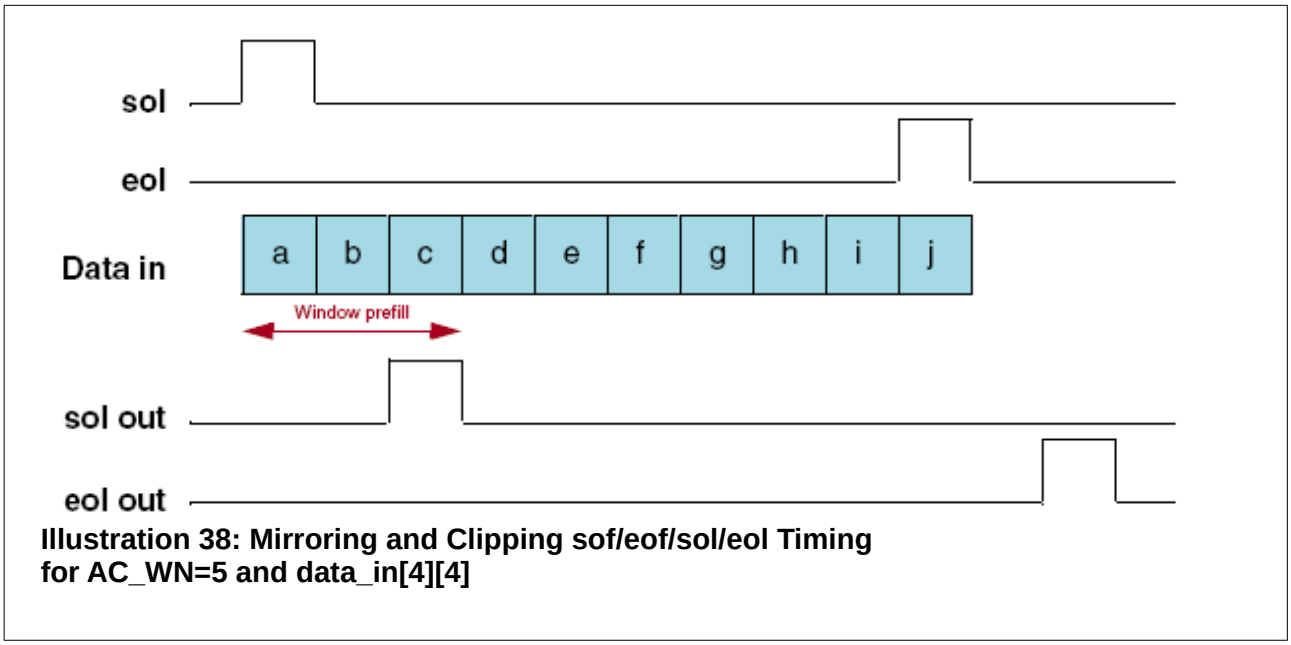

Illustration 38: Mirroring and Clipping sof/eof/sol/eol Timing for AC_WN=5 and data_in[4][4]

Using ignore_memory_precedence

The 2-d window classes typically require the use of memories to map the window's large internal arrays to memories. As new row data is read into the window, the old row data is shifted through these internal memories. Thus the internal memories are read and written at the same time. Although this is the desired behavior Catapult cannot prove that the memory reads occur before the memory writes when the design is pipelined. This will cause a pipelining failure because Catapult detects a memory dependency between the reads and writes to the memory. A constraint must be added by the user to direct Catapult to ignore the memory dependency in order to pipeline the design. This constraint is called "ignore_memory_precedence" and can usually be wild-carded based on the window class internal array name:

```
ignore_memory_precedences -from read_mem_sobel_w_vWind* -to
write_mem_sobel_w_vWind*
ignore_memory_precedences -from write_mem_sobel_w_vWind* -to
read_mem_sobel_w_vWind*
ignore_memory_precedences -from write_mem_sobel_w_vWind* -to
write_mem_sobel_w_vWind*
```

3.3. ac_window_2d_flag_flush_support Class

The ac_window_2d_flag_flush_support class is an adaptation of the ac_window_2d_flag class that supports flushing. As a result, much of the functionality of the former is preserved in the latter. However, the two classes do differ in certain key aspects, which will be explained in the ensuing sections.

This class is defined in ac_window_2d_flag_flush_support.h as follows:

```
template<class T, int AC_WN_ROW, int AC_WN_COL, int AC_NCOL, int AC_WMODE>
class ac_window_2d_flag_flush_support
{
public:
  ac_window_2d_flag_flush_support();
  ac_window_2d_flag_flush_support(T bval);
  void reset();
  T &operator()      (int r,int c);
  const T &operator()(int r, int c) const ;
  void write(T src, bool sof, bool eof, bool sol, bool eol);
  void exec(T src, bool sof, bool eof, bool sol, bool eol, bool w);
  bool valid();
  void readFlags(bool &sof, bool &eof, bool &sol, bool &eol);
  void rewind();
  bool isExtraWriteAllowed();
  bool spExtraItDynamic();
```

Loop Control Modifications

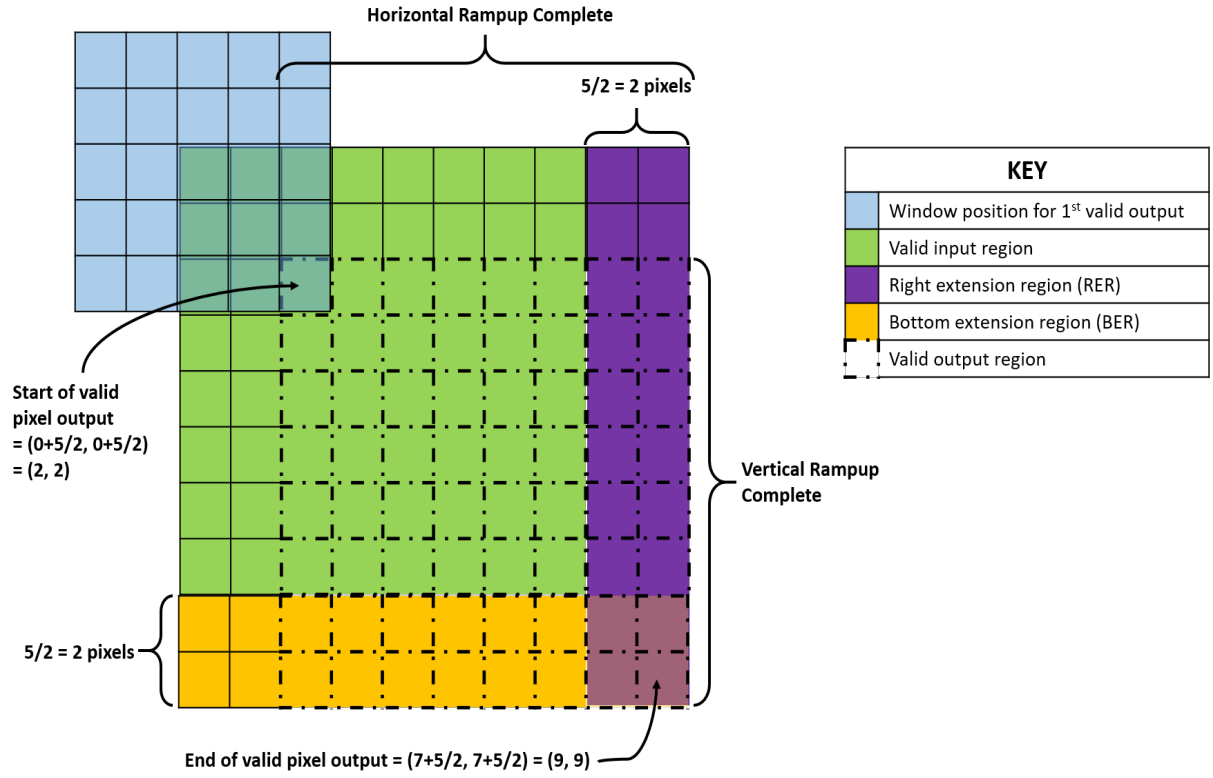

Illustration 39: Loop Control for Flushing

While using the ac_window_2d_flag_flush_support class, the user must extend the horizontal and vertical loops in order to account for the lag produced by both the horizontal and vertical ramp-up. This is done by adding extra iterations to both these loops. During the extra iterations, the design does not consume valid input data. The flushing happens during these extra iterations. The padded regions of the image that correspond to the extra iterations added to both the horizontal and vertical loops are called the right and bottom extension regions (RER and BER), respectively, as illustrated in Illustration 39: Loop Control for Flushing. The (i, j) coordinate system is used to denote loop positions. Note that the horizontal and vertical loops are also called row and column loops, respectively.

In the illustrated example, we are dealing with an 8x8 image that's being sample using a 5x5 window, with the AC_BOUNDARY windowing mode. We can think of the rampup as happening in two directions, horizontally and vertically. The vertical rampup has to do with pre-filling the linebuffer, that is, feeding the first two rows of the input image into the linebuffer. By the time we finish processing the input pixel at (1, 7), we have completed the vertical ramp-up for the entire image. The horizontal rampup, on the other hand, has to do with pre-filling column pixels into the window registers. In this case, the first two column pixels are pre-filled into the window registers. The horizontal ramp-up starts with the start of every line in the input image and is completed with the start of every line in the output image. The extension regions let the window compensate for the rampup-induced lag and stall the consumption of valid inputs while advancing in the corresponding directions and, once we've ramped up sufficiently, flushing the design of valid outputs.

From this, we can clearly see that, unlike the ac_window_2d_flag design, the version with flushing support does not support the continuous consumption of inputs/production of outputs, by necessity.

A coding example which shows the modified looping control can be found in Using the ac_window_2d_flag_flush_support Class.

Odd Image Widths and Singleport Memories

For singleport memories in designs pipelined with an II of 1, we need to ensure that we alternate between memory reads and writes every cycle so as to avoid scheduling conflicts. Due to this need of alternating between reads and writes, we only really write to the memory once every two cycles. Whenever we start processing a new line, the first memory access is a read, the second a write, the third a read, and so on. The singleport memory itself is double-wide, so as to compensate for the reduced number of accesses.

As a result of this alternating memory access pattern, we run into a problem with odd image column sizes. If we restrict the memory accesses to only the iterations where valid input pixels are received, the eol pixels aren't written to the linebuffers.

An example that demonstrates why this happens is illustrated in Illustration 40: Singleport Memory Access Patterns for Odd Image Widths. For this example, we assume a three-column wide input image matrix and consider iterations that begin with the start-of-line(sol) pixel for the second row of a 2x3 section of the image. We will assume that the linebuffer has already been filled with pixels from the previous row. The *X* pixels in the fourth column are placeholder input pixels, which are padding added to allow the eol pixels to fit inside the last location of the double-wide memory. tmp_in and tmp_out are temporary caches that store temporary data for reads from and writes to the singleport memory, respectively.

Iterations I to III alternate between memory access types as follows: read, write, read. We hence get two reads but only one write. To account for this discrepancy, we have to add an extra iteration at the end of the column loop. This extra iteration (Iteration IV in the illustration) writes the eol pixel value (i.e. *5*) to the linebuffer.

If a line happens to follow the second row in the 2x3 section displayed, we will now be able to access the aforementioned eol pixel value from the linebuffer, while processing the line.

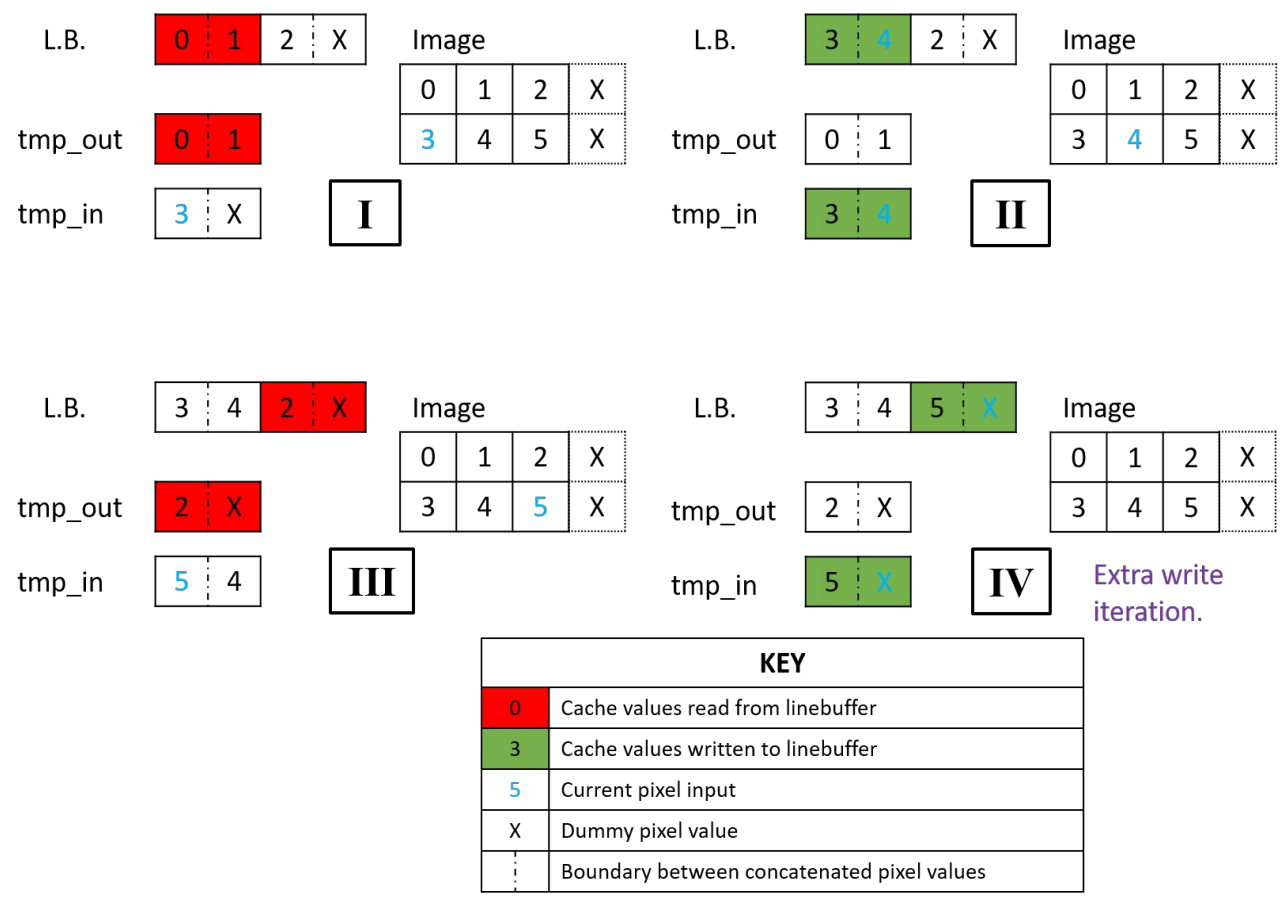

Illustration 40: Singleport Memory Access Patterns for Odd Image Widths

How we handle the extra iteration depends on what the window width is, as shown in Illustration 41: Handling the Extra Iteration.

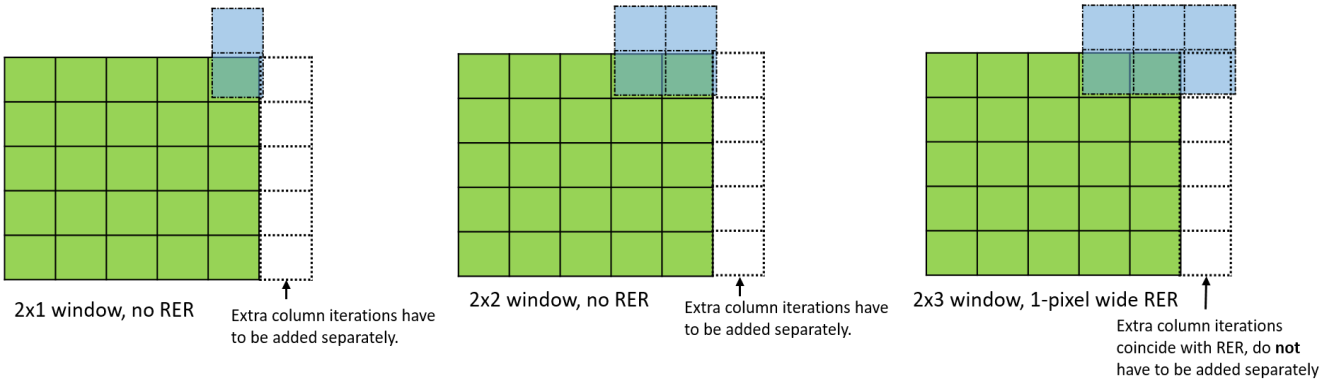

Illustration 41: Handling the Extra Iteration

The illustration depicts a 5x5 image being sampled by windows with varying widths, at the eol position. The windowing mode is AC_BOUNDARY. As can be seen in the image, for window width >= 3, the extra column

iterations would coincide with the RER iterations and we do not need to add them separately to the column loop. However, 2x2 and 2x1 windows have no padding on the right-hand side of the image, and as such they do not have an RER. We hence need to separately add an extra iteration to the column loop.

Whether or not this extra iteration needs to be separately added is something that can be ascertained only dynamically, once we know the position of the eol pixel and determine how wide the image is. The *spExtraItDynamic()* member function returns a boolean *true* if the extra iteration has to be added that way, letting the user change their column loop control as necessary.

An extra element of complexity is added when using the *exec()* function for cases when we need an extra write iteration. We have to set the value passed for the write flag as *true* for this extra iteration. The member function *isExtraWriteAllowed()* returns a boolean *true* if this is the case, letting the user adjust the parameters passed to *exec()* function accordingly.

ac_window_2d_flag_flush_support Template Parameters

The template parameters used are the same as those used for the *ac_window_2d_flag* design. For more details, refer to ac_window_2d_flag Template Parameters.

ac_window_2d_flag_flush_support Member Functions

For the most part, the functionality of the member functions is the same as those used for the ac_window_2d_flag design. Refer to ac_window_2d_flag Member Functions for more details.

However, some of the member functions have important changes made to them. They are listed as follows:

- **ac_window_2d_flag_flush_support()** - The non-parameterized version of the class constructor initializes the boundary value to zero so as to let the user have a default boundary value they can use without calling the parameterized constructor.

- **void write(T src, bool sof, bool eof, bool sol, bool eol)** - The write function is now designed to handle discontinuous inputs, and as such it makes sure that the placeholder inputs written to it in the RER are not written to the linebuffers. The function now updates two more rampup flags, one in the horizontal direction and the other in the vertical direction. The window is only ramped up and only produces valid outputs if both these rampup flags are true. Additional flags are also provided to set the horizontal rampup flag to *false* while processing the extra column loop iteration for the case explained in Odd Image Widths and Singleport Memories.

- **void exec(T src, bool sof, bool eof, bool sol, bool eol, bool w)** - The same changes made to the *write* function also apply to the *exec* function.

As explained in Odd Image Widths and Singleport Memories, there are two functions added to handle extra column loop iterations when needed:

- **bool spExtraItDynamic()** - Tells us whether an extra column loop iteration is needed or not.

- **bool isExtraWriteAllowed()** - Tells us whether the write flag to *exec* function needs to be adjusted for the extra column loop iteration where a write is allowed.

Using the ac_window_2d_flag_flush_support Class

As explained earlier, the loop control for the ac_window_2d_flag_flush_support class differs from the ac_window_2d_flag class in order to enable flushing, accommodate the extension regions and handle discontinuous input consumption/output generation. To demonstrate these differences in coding style, we consider a simple synthesizeable 3x3 sobel edge detector which uses the window class with flushing support. For simplicity's sake, we will only demonstrate the usage for dualport linebuffers.

```
const int WN_SZ = 3; // Row/column size for kernel and window.
// Sobel Kernel Data
const int cx[WN_SZ][WN_SZ] = {-1,0,1,-2,0,2,-1,0,1};
const int cy[WN_SZ][WN_SZ] = {1,2,1,0,0,0,-1,-2,-1};
const int IMG_ROW = 8; // Image row size
const int IMG_COL = 8; // Image column size
const int WIN_MODE = AC_CLIP; // Windowing mode
//const int WIN_MODE = AC_MIRROR; // Uncomment if you want to use AC_MIRROR
windowing

int abs_val(int in) { // Calculate absolute value
  return in < 0 ? -in : in;
}

#include <ac_window_2d_flag_flush_support.h>
#include <ac_channel.h>

#pragma design top
void sobel(ac_channel<int> &din_ch, ac_channel<int> &dout_ch) {
  // Declare windowing object
  static ac_window_2d_flag<int, WN_SZ, WN_SZ, IMG_COL, WIN_MODE> w;
  // Add an extra WN_SZ/2 iterations to row and column loops to complete flush-
ing
  ROW_LOOP: for (int i = 0; i < IMG_ROW + WN_SZ/2; i++) {
    COL_LOOP: for(int j = 0; I < IMG_COL + WN_SZ/2; j++) {
      // Initialize input framing control signals
      bool din_sof = (i == 0) && (j == 0);
      bool din_eof = (i == IMG_ROW – 1) && (j == IMG_COL – 1);
      bool din_sol = (i == 0);
      bool din_eol = (i == IMG_COL – 1);
      // Feed valid input data if in valid input region. If not, feed place-
holder data (0).
      int din = i < IMG_ROW && j < IMG_COL ? din_ch.read() : 0;
      w.write(din, din_sof, din_eof, din_sol, din_eol); // Write data+framing
signals
      int Gx = 0, Gy = 0; // Initialize gradient data
      if (w.valid()) { // Check if we are in valid output region.
#pragma hls_unroll yes
```

```
        for (int r = 0; r < WN_SZ/2; r++) {
#pragma hls_unroll yes
          for (int c = 0; c < WN_SZ/2; c++) {
            Gx += w(r-WN_SZ/2,c-WN_SZ/2)*cx[r][c];
            Gy += w(r-WN_SZ/2,c-WN_SZ/2)*cy[r][c];
          }
        }
        int dout = abs_val(Gx) + abs_val(Gy); // Calculate output
        dout_ch.write(dout); // Write output to output channel.
      }
    }
  }
}
```

The reference for this design is the same as the one shown in Using the ac_window_2d_flag Class.

3.4. ac_window_1d_stream Class

The ac_window_1d_stream class uses the ac_window_1d_flag class to create a more general purpose window implementation that supports a continuous stream of data. The use of the control flags is hidden from the user and internal counters are used to generate the required sol/eol. The only restriction of this class is that it requires the use of ac_channel for writing the window output. This is done since the window "pre-fill" will cause the last few samples of output data to be written as the next input array is read. The class is defined in ac_window.h and is defined as:

```
template<class T, int AC_WN, int AC_NCOL, ac_window_mode AC_WMODE = AC_WIN>
class ac_window_1d_stream{
public:
     ac_window_1d_stream();
     int operator++ ();
     int operator++ (int) { // non-canonical form (s.b. const window)
        return this->operator++(); }
     T & operator[]      (int i);
     const T & operator[](int i) const ;
     void write(T *s, int n);
     bool valid();
```

ac_window_1d_stream Template Parameters

- **T** - Specifies the input/output data type.
- **AC_WN** - Specifies the window size.
- **AC_NCOL** - Specifies the maximum array size
- **AC_WMODE** - Specifies the mode for handling the array boundaries.

- **AC_WIN** - No boundary handling.
- **AC_MIRROR** - Mirroring used on the boundary.
- **AC_CLIP** - Clipping used on the boundary.

ac_window_1d_stream Member Functions

- **ac_window_1d_stream()** - Constructor resets valid flag to false.
- **int operator++ ()** - Loads a new input into the window and advances the stored data through the window.
- **T & operator[](int i)** - Returns the window value indexed by "i". Currently this value must be specified as -AC_WN/2 < I < AC_WN/2 where AC_WN is the window size.
- **void write(T *s, int n)** - Points to the 1-d array of data being processed and sets the current array size.
- **bool valid()** - Returns true when the window has pre-filled.

Using the ac_window_1d_stream Class

This class typically requires a little rewriting of the C++ and test bench because of the use of channels. A generic algorithmic description may look like:

```
int mirror(int i, int sz){
return ((i<0)?(i*-1):((i<sz)?i:2*sz-i-2));
}
void window_ref (int linei[MAXSZ], int lineo[MAXSZ], int sz )
{
     for (int i=0; i<sz; i++) {
        lineo[i] = (linei[mirror(i-2, sz)] +
                    linei[mirror(i-1, sz)] +
                    linei[mirror(i  , sz)] +
                    linei[mirror(i+1, sz)] +
                    linei[mirror(i+2, sz)]
                    );
     }
}
```

The ac_window_1d_stream implementation looks very similar except that we now use a channel on the function output.

```
#pragma design top
void window_ccs(int din[128], ac_channel<int> &dout , int sz )
{
     static ac_window_1d_stream<int, 5,128, AC_MIRROR> w;
```

```
     w.write(din,sz);
     for(int i=0;i<MAXSZ;i++){
          w++;
        if(w.valid())
          dout.write(w[-2] + w[-1] + w[+0] + w[+1] + w[+2] );
        if(i==sz-1)
          break;
     }
}
```

The test bench can then be used to compare the original and modified outputs using channels:

```
#include "ac_window_1d_stream_example.h"
#ifdef CCS_SCVERIFY
#include "mc_testbench.h"
void testbench::main()
#else
     int main()
#endif
{
     int linei[128];
     int lineo[128];
     ac_channel<int> ccso;
     ac_channel<int> refo;
     int err = 0;
     int lsz[3] = {8, 16, 18};
     for (int testno=0; testno<3; testno++) {
        for (int i=0; i<128; i++) {
          linei[i]=i+1+testno*128;
        }
#ifdef CCS_SCVERIFY
        exec_window_ccs(linei,  ccso, lsz[testno]);
#else
        window_ccs(linei,  ccso, lsz[testno]);
#endif
        window_ref(linei, lineo, lsz[testno]);

        for (int i=0; i<lsz[testno]; i++) {
          refo.write(lineo[i]);
        }

        while(ccso.size() && refo.size()) {
          int ccst = ccso.read();
          int reft = refo.read();
          if (ccst!=reft) {
```

```
            err++;
         }
#ifdef DEBUG
         printf("%2x  %2x  ", ccst, reft);
         if (ccst!=reft) {
            printf("#\n");
         } else {
            printf("=\n");
         }
#endif
      }
   }

   printf("test passed with %d error(s)...\n", err);
   if (!err) printf("OK!\n");
}
```

3.5. ac_window_2d_stream Class

The ac_window_2d_stream class uses the ac_window_2d_flag class to create a more general-purpose window implementation that supports a continuous stream of data. The use of the control flags is hidden from the user and internal counters are used to generate the required sof/eof/sol/eol. The only restriction of this class is that it requires the use of ac_channel for writing the window output. This is done since the window "pre-fill" will cause the last few samples/lines of output data to be written as the next input array is read. The class is defined in ac_window.h and is defined as:

```
template<class T, int AC_WN_ROW, int AC_WN_COL, int AC_NROW, int AC_NCOL,
ac_window_mode AC_WMODE = AC_WIN>
class ac_window_2d_stream{
public:
   ac_window_2d_stream();
   int operator++ ();
   int operator++ (int) { // non-canonical form (s.b. const window)
      return this->operator++(); }
   T & operator()      (int r,int c);
   const T & operator()(int r, int c) const ;
   void write(T *s, int row_sz, int col_sz);
   bool valid();
```

ac_window_2d_stream Template Parameters

- **T** - Specifies the input/output data type.
- **AC_WN_ROW** - Specifies the vertical window size.
- **AC_WN_COL** - Specifies the horizontal window size.

- **AC_NROW** - Maximum number of array rows.
- **AC_NCOL** - Maximum number of array columns.
- **AC_WMODE** - Specifies the mode for handling the array boundaries.
  - **AC_WIN** - No boundary handling.
  - **AC_MIRROR** - Mirroring used on the boundary.
  - **AC_CLIP** - Clipping used on the boundary.
  - **AC_BOUNDARY** - Allows a constant boundary value to be specified in the constructor. (Instead of mirroring or clipping).
  - **AC_SINGLEPORT** - Use “true” singleport RAM. Singleport RAM is preferred over dualport RAM for ASIC designs due to lower area cost. This is only supported when using ac_int, ac_fixed, RGB_imd<ac_int> and RGB_imd<ac_fixed>. In order to add support for other types, the user must modify the template specialization of the ac_width2x struct, in the ac_window_structs.h header.
  - **AC_DUALPORT** - Use dualport RAM.
  - **AC_SYM_INDEX** - Use symmetrical index format specified as -AC_WN/2 < i < AC_WN/2, where AC_WN is the window size.
  - **AC_LIN_INDEX** - Use linear index format, such as i = 0 to 7. This format is more useful for windows with even dimensions.
  - **AC_REWIND** - Allows line buffer data to be used repeatedly by resetting the line buffer address pointers.
  - **AC_NO_REWIND** - Disable rewind option.

  To specify multiple parameters, *OR* them together in the template argument. For example:

```
static ac_window_2d_flag<
       ac_int<8,false> ,WN_SZ,WN_SZ,GRID,AC_MIRROR|AC_SINGLEPORT> w;
```

ac_window_2d_stream Member Functions

- **ac_window_2d_stream()** - Constructor resets valid flag to false and resets address counter that indexes window memories.
- **int operator++ ()** - Loads a new input into the window and advances the stored data through the window.
- **T & operator()(int r,int c)** - Returns the window value indexed by "r,c". Currently this value must be specified as -AC_WN_ROW/2 < r < AC_WN_ROW/2 and -AC_WN_COL/2 < c < AC_WN_COL/2.

- **void write(T *s, int row_sz, int col_sz)** - Points to the 2-d array of data being processed and sets the current array row and col size.

- **bool valid()** - Returns true when the window has pre-filled.

Using the ac_window_2d_stream Class

This class typically requires a little rewriting of the C++ and test bench because of the use of channels. A generic algorithmic description may look like:

```
#include <ac_int.h>
#include "sobel.h"
int clipo(int i, int sz){
     int tmp;
     if(i<0)
        tmp = 0;
     else if(i > sz-1)
        tmp = sz-1;
     else
        tmp = i;
     return tmp;
}

int mirror(int i, int sz){
     return ((i<0)?(i*-1):((i<sz)?i:2*sz-i-2));
}

int bounds(int i, int sz){
#ifdef MIRROR
     return mirror(i,sz);
#endif

#ifdef CLIP
     return clipo(i,sz);
#endif
}
int abso(int i){
     if(i<0)
        return -i;
     else
        return i;
}
const int cx[3][3] = {-1,0,1,-2,0,2,-1,0,1};
const int cy[3][3] = {1,2,1,0,0,0,-1,-2,-1};

#pragma design top
```

```
void sobel_orig(dType din[GRID][GRID], dType dout[GRID][GRID], int row_sz, int
col_sz){
     int Gx,Gy;
     int tmpx,tmpy;

     ROW:for(int i=0;i<row_sz;i++){
        COL:for(int j=0;j<col_sz;j++){
           tmpx = 0;
           tmpy = 0;
           for(int r=0;r<3;r++){
              for(int c=0;c<3;c++){
                 tmpx += din[bounds(i+r-1,GRID)]
                             [bounds(j+c-1,GRID)]*cx[r][c];
                 tmpy += din[bounds(i+r-1,GRID)]
                             [bounds(j+c-1,GRID)]*cy[r][c];
              }
           }
           Gx = tmpx;
           Gy = tmpy;
           dout[i][j] = abso(Gx) + abso(Gy);
        }
     }
}
```

This can be re-written using the ac_window_2d_stream class as:

```
#include <ac_int.h>
#include "sobel.h"

const int cx[3][3] = {-1,0,1,-2,0,2,-1,0,1};
const int cy[3][3] = {1,2,1,0,0,0,-1,-2,-1};

int abs(int i){
     if(i<0)
        return -i;
     else
        return i;
}
#include "ac_window.h"
#pragma design top
void sobel(int din[GRID][GRID], ac_channel<int> &dout, int row_sz, int col_sz){
#ifdef CLIP
     static ac_window_2d_stream<int,WN_SZ,WN_SZ,GRID,GRID,AC_CLIP> w;
#endif
#ifdef MIRROR
```

```
     static ac_window_2d_stream<int,WN_SZ,WN_SZ,GRID,GRID,AC_MIRROR> w;
#endif

     int Gx,Gy;
     int tmpx,tmpy;

     w.write(&din[0][0],row_sz,col_sz); //write all data and control inputs

     for(int i=0;i<GRID;i++){
        for(int j=0; j<GRID;j++){
           w++;
           tmpx = tmpy = 0;
           if(w.valid()){
              for(int r=0;r<WN_SZ;r++){
                 for(int c=0;c<WN_SZ;c++){
                    tmpx += w(r-WN_SZ/2,c-WN_SZ/2)*cx[r][c];
                    tmpy += w(r-WN_SZ/2,c-WN_SZ/2)*cy[r][c];
                 }
              }
              Gx = tmpx;
              Gy = tmpy;
              dout.write(abs(Gx) + abs(Gy));
           }
           if(j==col_sz-1)
              break;
        }
        if(i==row_sz-1)
           break;
     }
}
```

Once again the code changes are similar to the 1-d case. The input array is pointed to using the ac_window_2d_stream::write member function. The current row and column sizes are also set using this function. Conditional breaks are added so that the data is read and written continuously and the ac_window_2d_stream::valid function is used to wait until the window pre-fill has finished. The test bench is similar to the 1-d case and is included in the examples.

Using ignore_memory_precedence

The 2-d window classes typically require the use of memories to map the window's large internal arrays to memories. As new row data is read into the window, the old row data is shifted through these internal memories. Thus the internal memories are read and written at the same time. Although this is the desired behavior Catapult cannot prove that the memory reads occur before the memory writes when the design is pipelined. This will cause a pipelining failure because Catapult detects a memory dependency between the reads and writes to the memory. A constraint must be added by the user to direct Catapult to ignore the memory dependency in order to pipeline the design. This constraint is called "ignore_memory_precedence" and can usually be wild-carded based on the window class internal array name:

```
ignore_memory_precedences -from read_mem_sobel_w_vWind* -to
write_mem_sobel_w_vWind*
ignore_memory_precedences -from write_mem_sobel_w_vWind* -to
read_mem_sobel_w_vWind*
ignore_memory_precedences -from write_mem_sobel_w_vWind* -to
write_mem_sobel_w_vWind*
```

3.6. ac_window_1d_array Class

The ac_window_1d_array class requires the least amount of code changes but is also the most restrictive in the way it can be used. It is defined in the ac_window.h header file. This class uses the constructor to both point to the data and to set the maximum array size. One on the limitations of this class is that making the maximum array size a variable will break loop merging when trying to cascade several windows together. This is because it requires adding conditional breaks into the main design loops.

```
template<class T, int AC_WN, int AC_NCOL, ac_window_mode AC_WMODE = AC_WIN>
class ac_window_1d_array{
public:
     ac_window_1d_array(T *s, int n);
     int operator++ ();
     int operator++ (int) { // non-canonical form (s.b. const window)
        return this->operator++(); }
     T & operator[]      (int i);
     const T & operator[](int i) const ;
     int pos(){
        return cnt_;
     }
     bool valid(){
        return w.valid();
     }
```

ac_window_1d_array Template Parameters

- **T** - Specifies the input/output data type.
- **AC_WN** - Specifies the window size.

- **AC_NCOL** - Specifies the maximum array size.
- **AC_WMODE** - Specifies the mode for handling the array boundaries.
  - **AC_WIN** - No boundary handling.
  - **AC_MIRROR** - Mirroring used on the boundary.
  - **AC_CLIP** - Clipping used on the boundary.

ac_window_1d_array Member Functions

- **ac_window_1d_array(T *s, int n)** - Points to input array and sets the maximum array size.
- **int operator++ ()** - Loads a new input into the window and advances the stored data through the window.
- **T & operator[](int i)** - Returns the window value indexed by "i". Currently this value must be specified as -AC_WN/2 < I < AC_WN/2 where AC_WN is the window size.
- **int pos()** - Returns the current position of the window. This is used to control the loop iterations in the main processing loop. pos() is initialized to -AC_WN/2.
- **bool valid()** - Returns true when the window has pre-filled.

Using the ac_window_1d_array Class

This class requires the least amount of code modifications to the original C++ source. A generic algorithmic description may look like:

```
#include "ac_window_1d_array_example.h"
int mirror(int i, int sz){
     return ((i<0)?(i*-1):((i<sz)?i:2*sz-i-2));
}
void window_ref (int din[MAXSZ], int dout[MAXSZ] )
{
     int ltmp[MAXSZ];
     for (int i=0; i<MAXSZ; i++) {
        ltmp[i] = ( din[mirror(i-2, MAXSZ)] +
                    din[mirror(i-1, MAXSZ)] +
                    din[mirror(i  , MAXSZ)] +
                    din[mirror(i+1, MAXSZ)] +
                    din[mirror(i+2, MAXSZ)]
                    );
     }
     for (int i=0; i<MAXSZ; i++) {
        dout[i] = ( ltmp[mirror(i-2, MAXSZ)] +
                    ltmp[mirror(i-1, MAXSZ)] +
```

```
                    ltmp[mirror(i  , MAXSZ)] +
                    ltmp[mirror(i+1, MAXSZ)] +
                    ltmp[mirror(i+2, MAXSZ)]
                    );
     }
}
```

The example shown above processes the input array din, stores it in a temporary array ltmp, and then processes ltmp and writes it to dout. This can be re-written using the ac_window_1d_array class as:

```
#pragma design top
void window_ccs(int din[MAXSZ], int dout[MAXSZ])
{
     int ltmp[MAXSZ];
     ac_window_1d_array<int, 5,MAXSZ, AC_MIRROR> w0(din,MAXSZ);

     for(int i = w0.pos() ; i<MAXSZ; i++){
        w0++;//advance w0
        if(w0.valid()){
           ltmp[i] = (w0[-2] + w0[-1] + w0[+0] + w0[+1] + w0[+2] );
        }
     }
     ac_window_1d_array<int, 5,MAXSZ, AC_MIRROR> w1(ltmp,MAXSZ);
     for(int i=w1.pos();i<MAXSZ;i++){
        w1++;//advance w1
        if(w1.valid()){
           dout[i] = (w1[-2] + w1[-1] + w1[+0] + w1[+1] + w1[+2] );
        }
     }
}
```

Chapter 4: AC Window 2.0 Blocks

The AC Window 2.0 libraries were added as an upgrade to the older AC Window libraries. They feature significant enhancements to the older libraries:

- The new libraries can consume multiple inputs and produce multiple output windows in parallel, thereby increasing throughput.
- New linebuffer mappings are supported, e.g., Mapping multiple linebuffers to one singleport RAM with write-masking.
- The new designs are very modular, with multiple sub-blocks that can be reused across multiple designs.
- Flushing support, i.e., the ability to read new inputs while flushing old outputs, is now supported.

The AC Window 2.0 Designs can be divided into three types:

- Standard Windowing. This is similar in function to most of the old AC Window designs. The window continuously reads inputs till the end of the input frame has been reached, after which it enters a flushing period which accounts for the initial lag incurred due to the window's ramp-up. As the name suggests, the window flushes outputs during this period and does not read any inputs. 1D and 2D Windows are supported. 2D Windows must have at least three rows.
- Flush/Flushing Support. This is similar in function to standard windowing, but with the ability to accept inputs for the next frame during the flushing period, thereby increasing throughput. 2D Windows with at least three rows are supported.
- Line Flushing. The window now has a flushing period at the end of each frame AND each line which correspond to the boundary padding at the bottom and the right of the image, respectively. The flushing period at the end of each line is also called "line flushing". This is useful for image processing algorithms where the input data has gaps between lines as these gaps can be aligned with the line flushing. 2D Windows with at least three rows are supported.

All the window designs uses sub-blocks of the following types:

- Synchronization Flag Generator.
- Linebuffer.
- Data and Flag Shifters.
- Boundary Processor.
- Output Flag Generator for Standard Flushing.

While some of the sub-blocks are reused between the three different design types mentioned above, some of the sub-blocks are designed specifically for a particular design. The following table details the sub-blocks used by all the designs.

Sub-Block Type	Standard Windowing	Line Flushing	Flush Support
Synchronization Flag Gen.	ac_flag_gen	ac_flag_gen_lflush	ac_flag_gen_flush
Linebuffer	ac_linebuffer		ac_linebuffer_flush
Data Shifter	ac_shift_N		
Flag Shifter	ac_flag_shift		ac_flag_shift_flush
Output Flag Generator	ac_flag_gen_out	ac_flag_gen_out_lflush	ac_flag_gen_out_flush
Boundary Processor	ac_boundary		

As can be seen in the table above, the ac_shift_N and ac_flag_shift sub-blocks are reused across all the designs. The other sub-blocks are more design-specific.

Out of all these sub-blocks, ac_flag_gen and ac_linebuffer are generic and can be used in other image processing algorithms. They have been described in the "Image Processing" section of this reference manual. The rest are specifically designed to be used with AC Window 2.0 and are described in this section, under the following headings:

- ac_window_1d_flag Class
- Boundary Processor (ac_boundary)
- Output Flag Generator for Standard Windowing (ac_flag_gen_out)
- Synchronization Flag Generator for Flush Support (ac_flag_gen_flush)
- Linebuffers for Flush Support (ac_linebuffer_flush)
- Flag Shifter for Flush Support (ac_flag_shift_flush)
- Output Flag Generator for Flush Support (ac_flag_gen_out_flush)
- Synchronization Flag Generator for Line Flushing (ac_flag_gen_lflush)
- Output Flag Generator for Line Flushing (ac_flag_gen_out_lflush)

Each design also has its own top-level block which integrates all the relevant sub-blocks in addition to providing an interface for the user to send inputs and receive windowed outputs without needing to handle the intermediate outputs that are passed from one sub-block to the other. They are described under the following headings:

- Top-Level Design for Standard Windowing (ac_window_v2)

- Top Level Design for Flush Support (ac_window_v2_flush)
- Top Level Design for Line Flushing (ac_window_v2_lflush)

4.1. Data and Flag Shifters for Standard Windowing (ac_shift_N and ac_flag_shift)

The data shift registers store the window data before it is sent to the boundary processing block. The input is shifted in to slide the window in the desired direction. Since we shift the contents by N—where N is a positive integer that denotes the number of words streamed in parallel on the bus interface—the name of the classes for the data shifters are ac_shift_N_1d and ac_shift_N_2d for one- and two-dimensional shifting, respectively.

We also need to align the input flags with the shifted data, so that the boundary processing block knows where the image starts and where it ends in context of the windowed output. The flag shift registers handle this alignment.

4.1.1. C++ Code Overview for Data Shifters

A snippet of the ac_shift_N_1d class is given below.

```
template<typename T, int AC_WIN_WIDTH, ac_padding_method AC_PMODE, int
AC_BUS_WORDS = 1>
class ac_shift_N_1d {
public:
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  // Other code

private:
  ac_array<T, AC_SHIFT_WORDS> regs;
};
```

The “regs” member variable stores the shift register data. AC_SHIFT_WORDS is the size of the shift register. More details on how this size is computed are given in Calculating Shifter Size.

```
template<typename T, int AC_WIN_HEIGHT, int AC_WIN_WIDTH, ac_padding_method
AC_PMODE, int AC_BUS_WORDS = 1>
class ac_shift_N_2d {
public:
  static_assert(AC_WIN_HEIGHT > 0, "AC_WIN_HEIGHT must be positive.");
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");
```

```
  typedef ac_shift_N_1d<T, AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS> LINE_TYPE;

  enum {
    USING_REFLECT101 = AC_PMODE == AC_REFLECT101,
    EXTRA_LINE = USING_REFLECT101 ? int(AC_WIN_HEIGHT%2 == 0) : 0,
    AC_SHIFT_HEIGHT = AC_WIN_HEIGHT + EXTRA_LINE
  };

  // Other code.

private:
  LINE_TYPE lines[AC_SHIFT_HEIGHT];
};
```

As can be seen in the snippet above, the ac_shift_N_2d class is built on top of the ac_shift_N_1d class, and uses lines, an array of ac_shift_N_1d objects serving to store the horizontal shift register data for each line. If we're using AC_REFLECT101 as the boundary padding mode and if AC_WIN_HEIGHT is even, we must add an extra line to the shift register to enable correct mirroring.

ac_shift_N_1d Template Parameters

- T – Specifies the base type, i.e. the type of each element in the shift register.
- AC_WIN_WIDTH – Window width, i.e. the number of columns in the window.
- AC_PMODE – Boundary padding mode. More details on this parameter are given in Boundary Processing Modes.
- AC_BUS_WORDS – Number of pixels sent in parallel over the bus interface. It is set to 1 by default.

Both the integer parameters, i.e. AC_WIN_WIDTH and AC_BUS_WORDS must be positive, failing which a static assertion is triggered.

ac_shift_N_2d Template Parameters

ac_shift_N_2d uses the same template parameters that ac_shift_N_1d does, along with an additional template parameter AC_WIN_HEIGHT for the window height, i.e. the number of rows in the window. AC_WIN_HEIGHT, AC_WIN_WIDTH and AC_BUS_WORDS must all be positive, failing which a static assertion is triggered.

Constructors

Constructor	Description
ac_shift_N_1d()	Default ac_shift_N_1d constructor. Calls the reset() function mentioned in Miscellaneous ac_shift_N_1d Methods.
ac_shift_N_2d()	Default ac_shift_N_2d constructor. Calls the reset() function mentioned in Miscellaneous ac_shift_N_2d Methods.

ac_shift_N_1d Operator Overloading

Method	Description
void operator << (ac_array<T, AC_BUS_WORDS> din)	Used to carry out left-shifting with din as the input.
void operator >> (ac_array<T, AC_BUS_WORDS> din)	Used to carry out right-shifting with din as the input.
T &operator[] (unsigned idx)	Subscript operator overloading. Returns a reference of the register element at index idx. Can be used to read or write individual elements. Will assert if idx >= AC_SHIFT_WORDS.

ac_shift_N_2d Operator Overloading

Method	Description
void operator << (ac_array<T, AC_SHIFT_HEIGHT, AC_BUS_WORDS> din)	Iteratively left-shifts all lines of din into all the horizontal shift registers.
void operator >> (ac_array<T, AC_SHIFT_HEIGHT, AC_BUS_WORDS> din)	Iteratively right-shifts all lines of din into the horizontal shift registers.
LINE_TYPE &operator[] (unsigned idx)	Subscript operator overloading. Returns a reference of the horizontal shift register at index idx. Can be used to read or write individual elements of the horizontal shift registers. Will assert if idx >= AC_SHIFT_HEIGHT.

Miscellaneous ac_shift_N_1d Methods

Method	Description
void lshift(ac_array<T, AC_BUS_WORDS> din, ac_array<T, AC_WORDS + EXTRA_WORD> &dout)	Uses the << operator to shift din into the shift register and copies the last (AC_WORDS + EXTRA_WORD) elements of the shift register to dout. Details on how AC_WORD and EXTRA_WORD are computed are given in Calculating Shifter Size.
void rshift(ac_array<T, AC_BUS_WORDS> din, ac_array<T, AC_WORDS + EXTRA_WORD> &dout)	Uses the >> operator to shift din into the shift register and copies the first (AC_WORDS + EXTRA_WORD) elements of the shift register to dout. Details on how AC_WORD and EXTRA_WORD are computed are given in Calculating Shifter Size.
void reset()	Resets all elements of the shift register to 0.

Miscellaneous ac_shift_N_2d Methods

Method	Description
void lshift(ac_array<T, AC_SHIFT_HEIGHT, AC_BUS_WORDS> din, ac_array<T, AC_SHIFT_HEIGHT, AC_WORDS + EXTRA_WORD>& dout)	Iteratively calls the lshift() function described in Miscellaneous ac_shift_N_1d Methods to write each line of din into the corresponding horizontal shift register and copy the corresponding shift register output to each line of dout.

void rshift(ac_array<T, AC_SHIFT_HEIGHT, AC_BUS_WORDS> din, ac_array<T, AC_SHIFT_HEIGHT, AC_WORDS + EXTRA_WORD>& dout)	Iteratively calls the rshift() function described in Miscellaneous ac_shift_N_1d Methods to write each line of din into the corresponding horizontal shift register and copy the corresponding shift register output to each line of dout.
void reset()	Resets all elements of the shift register to 0.

4.1.2. C++ Code Overview for Flag Shifters

A snippet of the ac_flag_shift_1d class is given below.

```
template<int AC_WIN_WIDTH, ac_padding_method AC_PMODE, int AC_BUS_WORDS = 1>
class ac_flag_shift_1d {
public:
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  // Other code

private:
  ac_int<AC_SHIFT_WORDS, false> sol_regs, eol_regs;
};
```

The sol_regs and eol_regs member variables store the shift register data. They are similar to the regs member variable mentioned in C++ Code Overview for Data Shifters, except that these registers are an ac_int vector of bits rather than ac_array objects. AC_SHIFT_WORDS is the size of the shift register, which is the same as the size of the horizontal data shifters. More details on how this number is computed are given in Calculating Shifter Size.

A snippet of the ac_flag_shift_2d class is given below.

```
template <int AC_WIN_HEIGHT, int AC_WIN_WIDTH, ac_padding_method AC_PMODE, int
AC_BUS_WORDS = 1, bool AC_REPEAT = false>
class ac_flag_shift_2d {
public:
  static_assert(AC_WIN_HEIGHT > 0, "AC_WIN_HEIGHT must be positive.");
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  typedef ac_flag_shift_1d<AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS> LINE_TYPE;

  enum {
    USING_REFLECT101 = AC_PMODE == AC_REFLECT101,
    EXTRA_LINE = USING_REFLECT101 ? int(AC_WIN_HEIGHT%2 == 0) : 0,
    AC_SHIFT_HEIGHT = AC_WIN_HEIGHT + EXTRA_LINE,
  };
```

```
  // Other code.

private:
  LINE_TYPE line;
  ac_array<bool, AC_SHIFT_HEIGHT, NUM_SHIFTS_FILL + EXTRA_WORD> sof_regs,
eof_regs;
  bool eof_old, eol_old;
};
```

As can be seen in the snippet above, the ac_flag_shift_2d class is built on top of the ac_flag_shift_1d class, and uses line, an ac_flag_shift_1d object serving to store and update the horizontal flag registers. It is similar to the “lines” object described in C++ Code Overview for Data Shifters, except we only need one such object to store the horizontal flag registers because it remains the same for each line of our window. The sof_regs and eof_regs shift registers store the and shift the SOF and EOF flags as the window slides over the image. The computation of AC_SHIFT_HEIGHT is the same as that for the data shifter. More details on how NUM_SHIFTS_FILL and EXTRA_WORD parameters are calculated are given in Calculating Shifter Size. The eof_old and eol_old member variables let us delay shifting the eof_regs register, so that it happens in the iteration right after the EOF pixel is received, which in turn allows for correct boundary processing. The eof_old flag is returned by the run function of the 2D flag shifter and used by the output flag generator when no padding is used, as explained in Output Flag Generator for Standard Windowing (ac_flag_gen_out).

ac_flag_shift_1d Template Parameters

- AC_WIN_WIDTH – Window width, i.e. the number of columns in the window.
- AC_PMODE – Boundary padding mode. More details on this parameter are given in Boundary Processing Modes.
- AC_BUS_WORDS – Number of pixels sent in parallel over the bus interface. It is set to 1 by default.

ac_flag_shift_2d Template Parameters

- AC_WIN_HEIGHT – Window height, i.e. the number of rows in the window.
- AC_WIN_WIDTH – Window width, i.e. the number of columns in the window.
- AC_PMODE – Boundary padding mode. More details on this parameter are given in Boundary Processing Modes.
- AC_BUS_WORDS – Number of pixels sent in parallel over the bus interface. It is set to 1 by default.
- AC_REPEAT – This parameter is set to true if the window is expected to iterate over the same set of lines for algorithms such as interpolation.

Constructors

Constructor	Description
ac_flag_shift_1d()	Default ac_flag_shift_1d constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_shift_1d Methods.

ac_flag_shift_2d()	Default ac_flag_shift_2d constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_shift_2d Methods.

Miscellaneous ac_flag_shift_1d Methods

Method	Description
void run(bool sol, bool eol, ac_int<AC_WORDS, false> &sol_flags, ac_int<AC_WORDS, false> &eol_flags)	Carries out right-shifting, updates the SOL and EOL flag registers, and writes their content to sol_flags and eol_flags respectively.
void reset()	Sets the sol_flags and eol_flags registers to 0.

Miscellaneous ac_flag_shift_2d Methods

Method	Description
bool run(bool sof, bool eof, bool sol, bool eol, ac_int<AC_WIN_HEIGHT, false> &sof_flags, ac_int<AC_WIN_HEIGHT, false> &eof_flags, ac_int<AC_WORDS, false> &sol_flags, ac_int<AC_WORDS, false> &eol_flags, bool repeat_line = false)	Carries out right-shifting for the horizontal flag registers. Updates the SOF, EOF, SOL and EOL flag registers, and writes their content to sof_flags, eof_flags, sol_flags and eol_flags, respectively. If the repeat_line function input and the AC_REPEAT template parameter are true, the flag shifter assumes that the window is iterating over the same set of lines. In such a case, the SOF/EOF flag registers do not accept new inputs, which in turn means that the window does not slide vertically. repeat_line can only be true if AC_REPEAT is also true, failing which an assertion is triggered. This function returns the value of eof_old, i.e. the eof value seen the last time the run function is called. This is sometimes used by the output flag generator to generate the vld_out flag, when no boundary padding is used. More details are given in Output Flags without Padding.
void reset()	Calls the reset() method for the line object. Also sets the eof_old, eol_old, sof_regs and eof_regs members to false/0.

4.1.3. Horizontal Shift Direction

Consider the snippet below that demonstrates the left- or right-shifting done on an unsigned 8-bit integer.

```
ac_int<8, false> x = 8;
ac_int<8, false> x_rshift = x >> 2; // x_rshift = 8 >> 2 = 2
ac_int<8, false> x_lshift = x << 2; // x_lshift = 8 << 2 = 32
```

Right-shifting, as done while computing x_rshift, shifts the bits from the MSB to the LSB. The MSB corresponds to the upper bound of the bit indices (7), while the LSB corresponds to the lower bound (0). The view of the right-shift operation through the binary representation of x is as follows:

```
(00001000)2 >> 2 = (00000010)2
```

Since the binary representation starts with the upper bound on the left and ends with the lower bound on the right, it seems that the bits are being shifted rightward and this operation is called right-shifting.

Similarly, in the left-shift operation, it seems like the bits are being shifted leftward:

```
(00001000)2 << 2 = (00100000)2
```

However, when we represent an array of pixel values as opposed to a vector of bits, the diagrammatic representation of the array is the inverse of what it is for the bit vector, i.e. it starts with the lower bound on the left and the upper bound on the right. Hence, “right”-shifting for an ac_array actually looks like we’re shifting leftward when we show it diagrammatically. Consider Illustration 42 where we shift in 2 data values at a time (AC_BUS_WORDS = 2) into a register with 8 elements (AC_SHIFT_WORDS = 8) using the >> operator overloading described in Miscellaneous ac_shift_N_1d Methods.

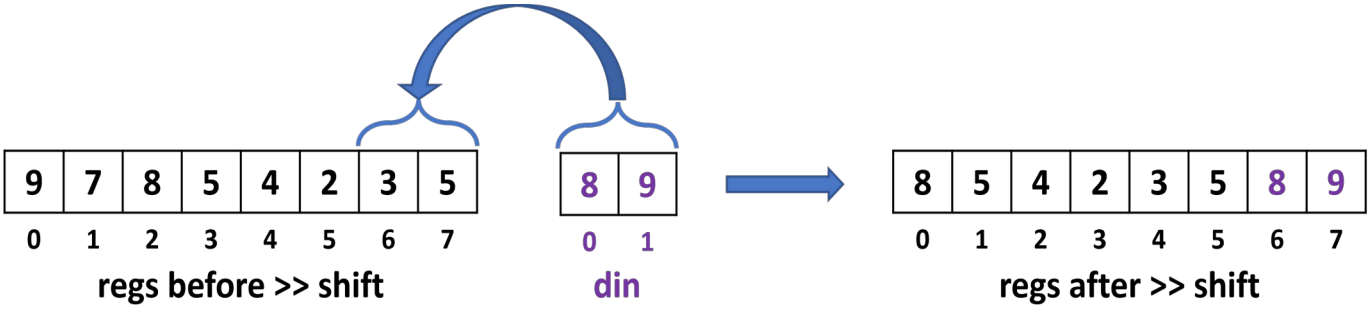

Illustration 42: Right-shifting Data Values

While this shifting operation is analogous to the right-shifting on the integer described above in terms of the relation between the shift direction and the indices, it actually looks like we’re shifting left instead of right based on the diagrammatic representation. The user should keep this discrepancy in mind. Note that the most recent pixels are stored at the upper bounds of the register array (in this case, indices 6 and 7) and the other pixels are shifted toward the lower bound.

Similarly, the diagrammatic representation of an equivalent “left”-shifting operation makes it look like we’re shifting right instead of left, with the most recent pixels stored at the lower bounds of the register array (in this case, indices 0 and 1).

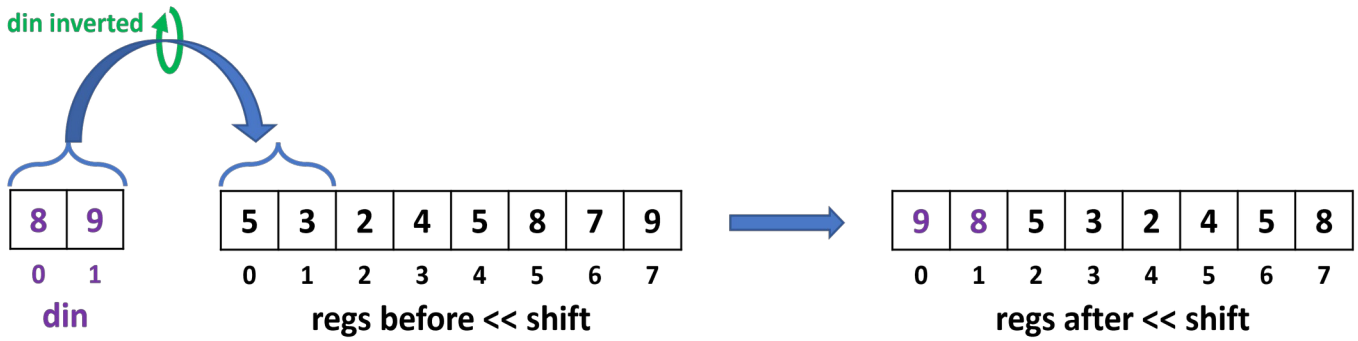

Illustration 43: Left-shifting Data Values

Note that to make sure that the most recent pixel—in this case, the value 9—gets stored at index 0, we invert din before storing it in the shifter, as represented by the green circular arrow in Illustration 43.

Since image processing applications generally store recently received pixels at the upper bounds of storage arrays, the AC Window 2.0 design uses right-shifting to update the registers. However, the ac_shift_N IP also has left-shifting capabilities in case the user wants to use data shifter standalone.

As mentioned earlier, the flag shifters use two ac_int bit vectors—sol_regs and eol_regs—to store the horizontal flags. These bit vectors are shifted by the run() function described in Miscellaneous ac_flag_shift_1d Methods, using the right shift assignment operator in C++:

```
sol_regs >>= AC_BUS_WORDS;
eol_regs >>= AC_BUS_WORDS;
```

The position at which the sol and eol input flags are stored corresponds to the pixels received by the data shifter. Assuming the same case depicted in Illustration 42, i.e. AC_SHIFT_WORDS = 8 and AC_BUS_WORDS = 2, the SOL pixel (when received) is located at the lower bound of din (index 0) and the EOL pixel (when received) is located at the upper bound of din (index 1 for AC_BUS_WORDS = 2).

Illustration 44 depicts how the SOL and EOL flags are stored when they're received. Note that the ac_int vectors in the illustration are represented in the array format, i.e. starting with the lower bound on the left and ending with the upper bound on the right. This is done to keep it consistent with the array representation in Illustration 42.

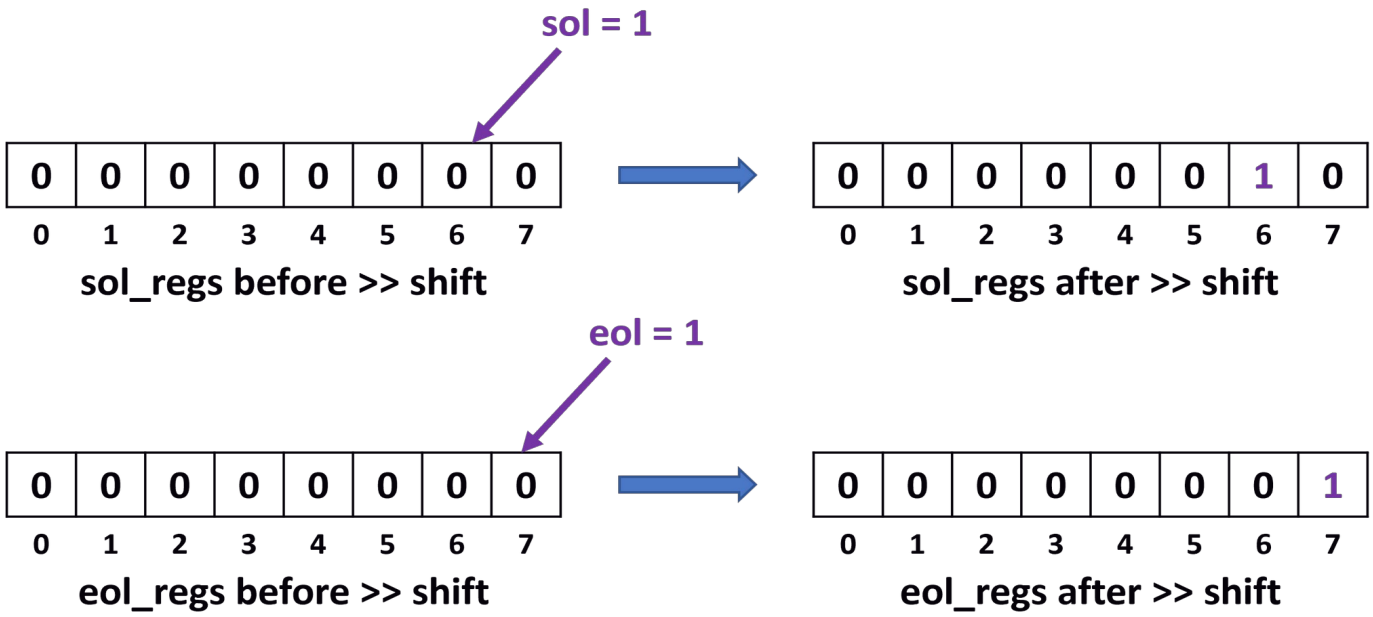

Illustration 44: Right-shifting SOL Flags

The SOL flag value is stored at the index given by AC_SHIFT_WORDS - AC_BUS_WORDS, which in the case described above is 6 (8 - 2). The EOL flag value is stored at the index given by AC_SHIFT_WORDS - 1, which in the case described above is 7 (8 – 1). The corresponding indices are highlighted in the snippet below.

```
    sol_regs >>= AC_BUS_WORDS;
    eol_regs >>= AC_BUS_WORDS;
    sol_regs[AC_SHIFT_WORDS - AC_BUS_WORDS] = sol;
    eol_regs[AC_SHIFT_WORDS - 1] = eol;
```

4.1.4. Calculating Shifter Size

The minimum size of the horizontal shift registers is given by AC_WORDS. It is the size required to store all the overlapped windows outputted by AC Window 2.0 in parallel. This number of overlapped windows is the same as the number of pixels sent in parallel to AC Window 2.0, i.e. AC_BUS_WORDS.

For a given value of AC_WIN_WIDTH and AC_BUS_WORDS, we compute AC_WORDS as follows:

```
AC_WORDS = AC_WIN_WIDTH + AC_BUS_WORDS - 1
```

Let us assume a case where AC_WIN_WIDTH = 5 and AC_BUS_WORDS = 3. In this case, AC_WORDS = 5 + 3 - 1 = 7. Now, the size of the shifter will depend on whether we're implementing boundary padding or not. Let use consider both cases separately.

Shifter Size with Boundary Padding

When using boundary padding, a key requirement for AC Window 2.0 is that all the windows should ramp up simultaneously in the horizontal direction. To figure out the length of the shift registers required to allow this to happen, we first subtract the number of pixels that are padded at the left boundary of the image (AC_WIN_WIDTH/2) from AC_WORDS, to find the number of pixels inside the image at the time of horizontal ramp-up:

```
AC_WORDS_INSIDE = AC_WORDS - AC_WIN_WIDTH/2
```

In this case, AC_WORDS_INSIDE = 7 - 5/2 = 5. This case is shown in Illustration 45, where the grayed-out pixels represent the left boundary pixels and each green curly brace represents a single window.

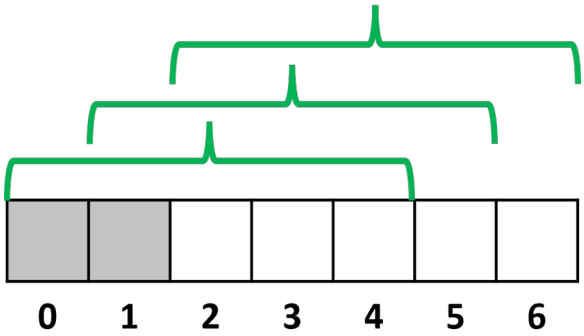

Illustration 45: AC_WORDS_INSIDE

We must then figure out how many data shifts are required before the window is ramped up horizontally. Since each data shift fills the shift register with AC_BUS_WORDS number of inputs and the minimum number of shift register elements we need to fill with inputs for horizontal rampup is given by AC_WORDS_INSIDE, the number of shifts required to fill the window up is given by the formula

ceil(AC_WORDS_INSIDE/AC_BUS_WORDS). This value can be computed at compile-time using the C++ expression below:

```
NUM_SHIFTS_FILL = AC_WORDS_INSIDE/AC_BUS_WORDS + int(AC_WORDS_INSIDE
%AC_BUS_WORDS != 0)
```

For this case, NUM_SHIFTS_FILL = ceil(5/3) = ceil(1.667) = 2.

The number of register elements that must be present to accommodate the inputs filled in after NUM_SHIFTS_FILLS number of data shifts without the loss of data is given by the formula NUM_SHIFTS_FILL*AC_BUS_WORDS.

If we're using an even window width with AC_REFLECT101 as the boundary padding mode and if we're passing in one pixel at a time (AC_BUS_WORDS = 1), we need to add an extra word to the horizontal shift registers to enable correct mirroring. Whether or not this extra word is needed is computed as follows:

```
USING_REFLECT101 = AC_PMODE == AC_REFLECT101
EXTRA_WORD = USING_REFLECT101 && AC_WIN_WIDTH%2 == 0 && AC_BUS_WORDS == 1
```

For this case, EXTRA_WORDS = 0 as we're passing multiple words in parallel.

Adding EXTRA_WORD and NUM_SHIFT_FILLS*AC_BUS_WORDS to the number of left boundary pixels gives us the total number of register elements, i.e. AC_SHIFT_WORDS:

```
AC_SHIFT_WORDS = NUM_SHIFTS_FILL*AC_BUS_WORDS + AC_WIN_WIDTH/2 + EXTRA_WORD
```

Note that if an extra word is not added and if AC_WORDS_INSIDE is divisible by AC_BUS_WORDS, AC_SHIFT_WORDS and AC_WORDS are the same.

For this case, AC_SHIFT_WORDS = 2*3 + 5/2 + 0 = 8. AC_SHIFT_WORDS and AC_WORDS are not the same because AC_WORDS_INSIDE (5) is not divisible by AC_BUS_WORDS (3). If we're using the rshift() function, the first AC_WORDS (7) number of elements are taken out of the shift register to give the overlapped windows output (dout), as represented by the orange curly brace in Illustration 46, where green curly braces continue to represent each individual window.

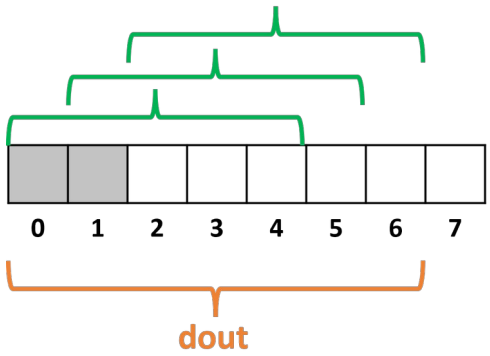

Illustration 46: Overlapped Windows Output

We can confirm that we need 2 (i.e. NUM_SHIFTS_FILL) shifts to ramp up the window by considering Illustration 47. At the end of the second shift, we can see that the SOL pixel value (1) is just right of the left boundary region, indicating that the linebuffer is ramped up and ready to produce outputs. The boundary region is now ready to get padded by the boundary processing block.

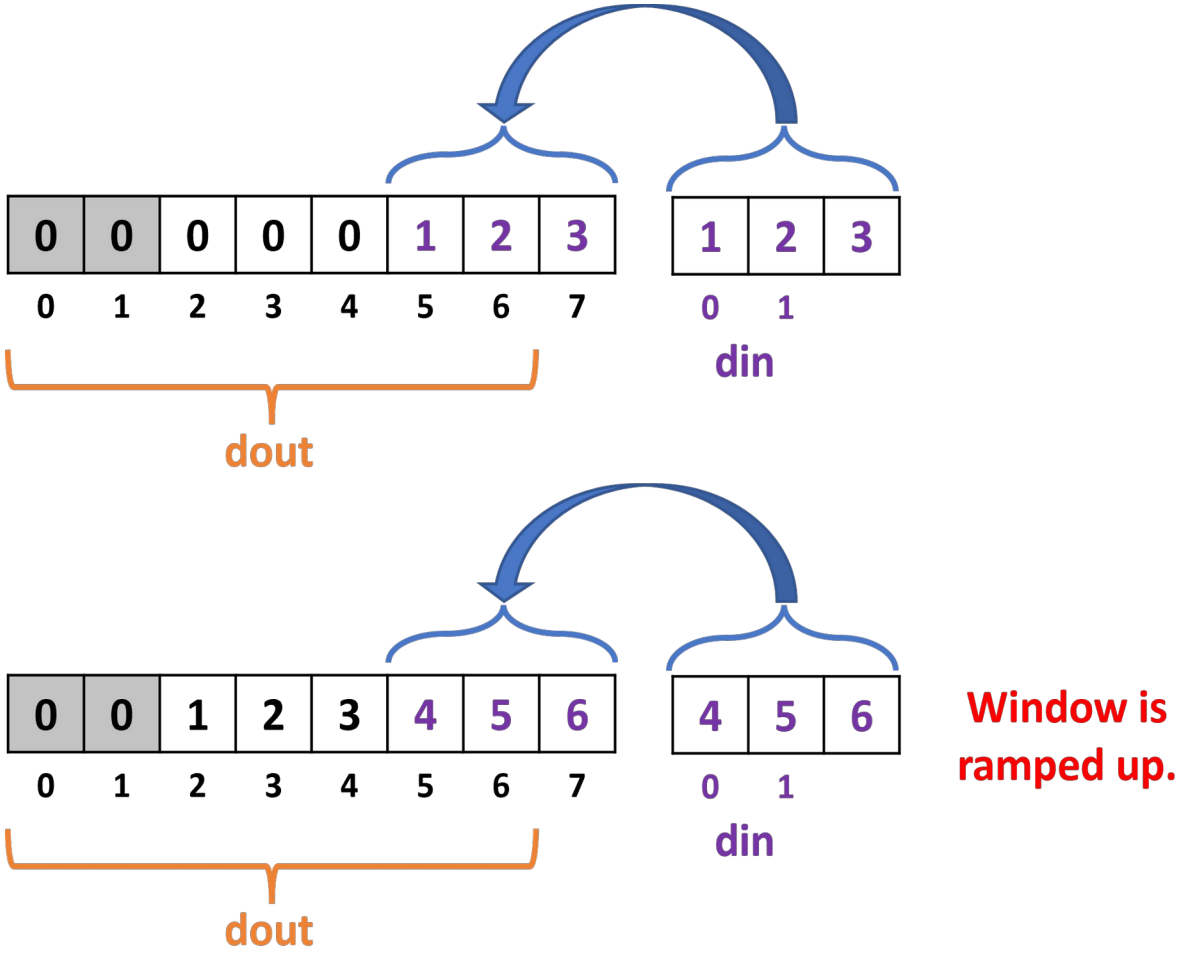

Illustration 47: Horizontal Window Rampup

Shifter Size without Boundary Padding

Without boundary padding, the assumptions are different. Now, if an individual window isn't fully inside the image, it's invalid. As a result, the window generally takes longer to ramp up. The number of shifts required to fill the window with enough pixels to ramp up is also ceil(AC_WORDS_INSIDE)/NUM_SHIFTS_FILL, with the difference being that AC_WORDS_INSIDE and AC_WORDS have the same value, as the window must be fully inside the image to have ramped up. The value of NUM_SHIFTS_FILL can hence be computed at compile time as follows:

```
AC_WORDS_INSIDE = AC_WORDS
NUM_SHIFTS_FILL = AC_WORDS_INSIDE/AC_BUS_WORDS + int(AC_WORDS_INSIDE
%AC_BUS_WORDS != 0)
```

In this case, the boundary padding mode is AC_NO_PADDING. Since this padding mode is mutually exclusive with AC_REFLECT101, EXTRA_WORD is always 0, i.e. no extra word is added. AC_SHIFT_WORDS can be computed by simply multiplying NUM_SHIFTS_FILL with AC_BUS_WORDS:

```
AC_SHIFT_WORDS = NUM_SHIFTS_FILL*AC_BUS_WORDS
```

When AC_WIN_WIDTH = 5 and AC_BUS_WORDS = 3, AC_WORDS_INSIDE = AC_WORDS = 7. Hence, NUM_SHIFTS_FILL = ceil(7/3) = 3. The width of the shifter is computed as:

AC_SHIFT_WORDS = NUM_SHIFTS_FILL*AC_BUS_WORDS = 3*3 = 9.

A visual representation of the horizontal rampup in such a case is shown in the following illustration:

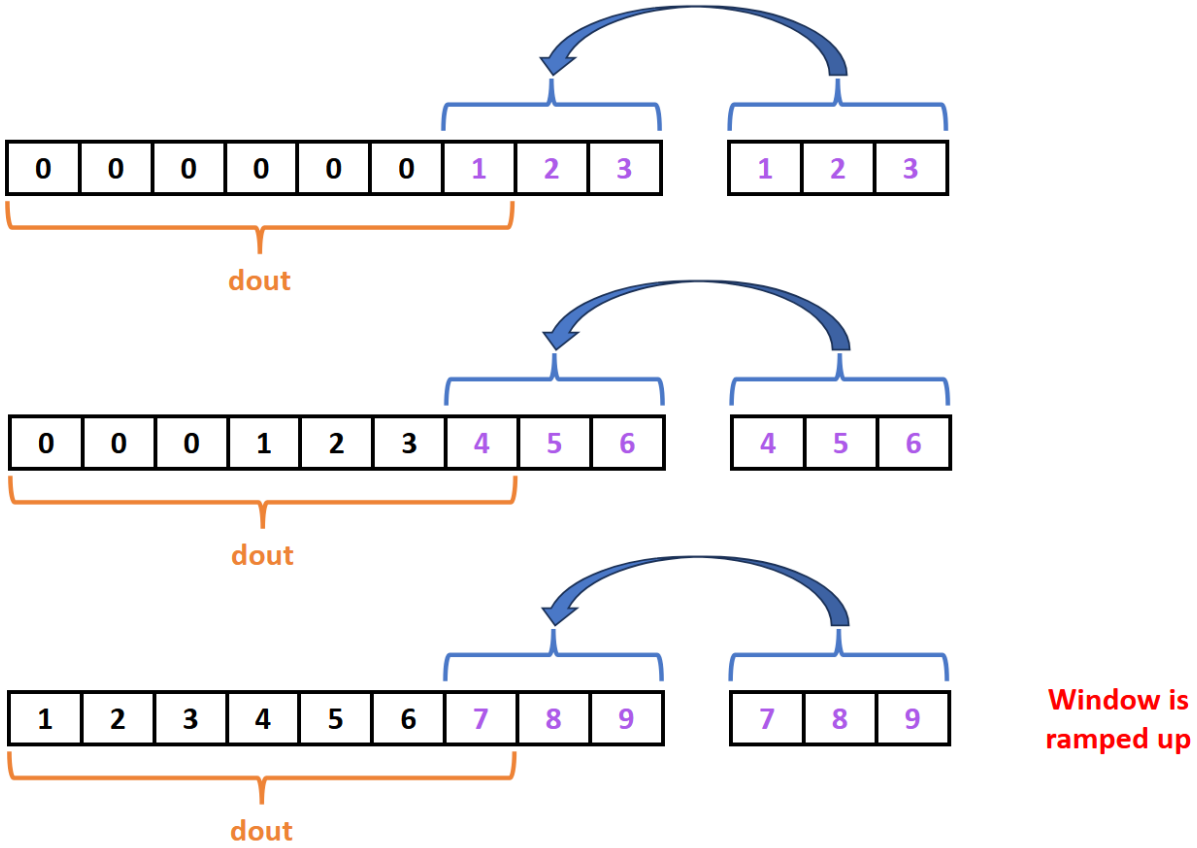

Illustration 48: Horizontal Window Rampup, without Padding

Note that in such a case, all the individual windows do ramp up at the same time, same as when we do have boundary padding. However, in cases where AC_WORDS is not a multiple of AC_BUS_WORDS—such as this one—the last column of the overlapped window, i.e. dout, does not align with the end-of-line pixel. As a result, all of the individual windows do not lie within the image and they do not ramp down at the same time. In other words, the misalignment causes some of the windows to be valid while others are not. If we assume an image width of 12, the view of the window when the end of line pixel is outputted is shown in Illustration 51. Note the pixels from the next line are denoted with the letter "x" as they are irrelevant for the windowed output at the end of the line. The window for the next line will take a while to be valid again, thereby resulting in a "blank" period between the lines where the output is invalid.

The information on when the overlapped window is valid and which individual window is valid is given by the vld_out flag, as explained in Output Flags without Padding.

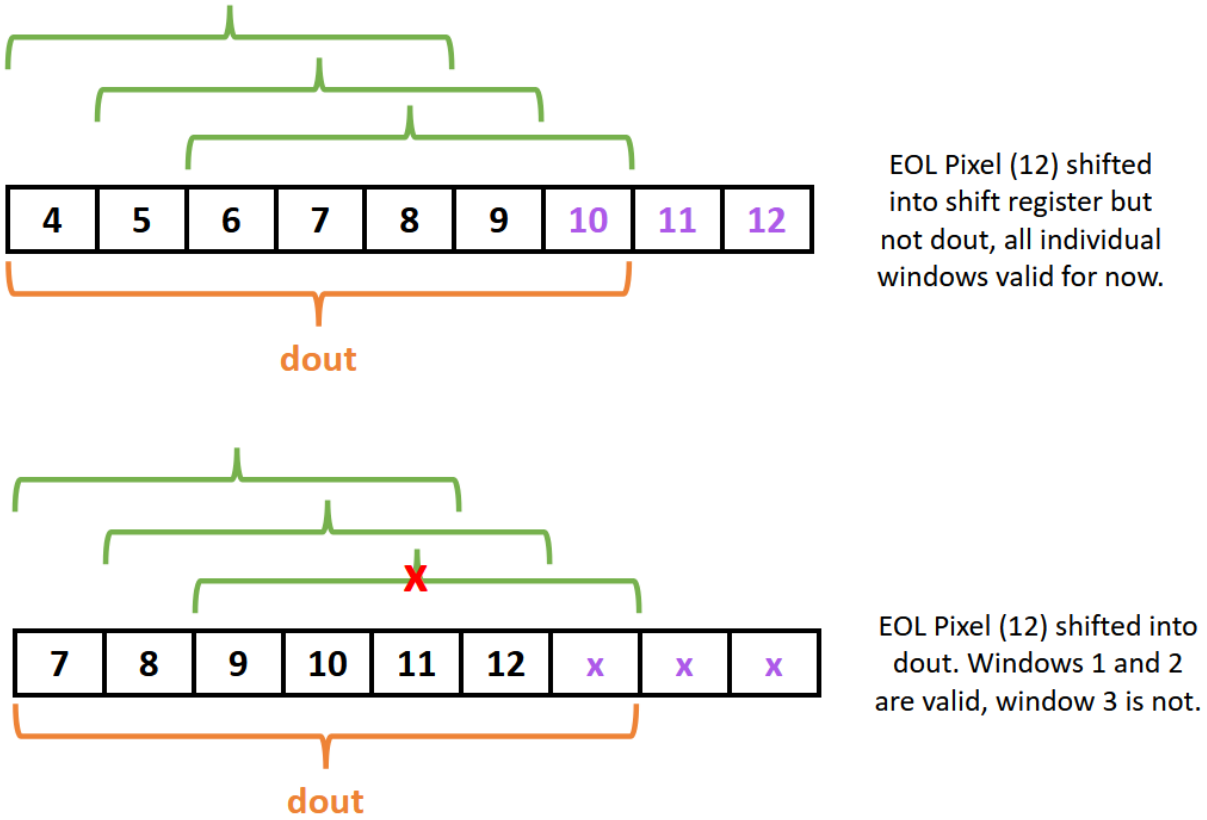

Illustration 49: Horizontal Window Rampdown, without Padding

Helper Struct to Compute Shifter Size

All the computations for the horizontal shifter size are contained within the calc_shifter_words struct inside ac_boundary_enums.h, as shown below:

```
template <int AC_WIN_WIDTH, ac_padding_method AC_PMODE, int AC_BUS_WORDS>
struct calc_shifter_words {
  enum {
    USING_PADDING = AC_PMODE != AC_NO_PADDING,
    USING_REFLECT101 = AC_PMODE == AC_REFLECT101,
    AC_WORDS = AC_WIN_WIDTH + AC_BUS_WORDS - 1,
    // LPAD_WORDS and RPAD_WORDS: Number of words padded on the left and right
    LPAD_WORDS = USING_PADDING ? AC_WIN_WIDTH/2 : 0,
    RPAD_WORDS = USING_PADDING ? AC_WIN_WIDTH/2 - int(AC_WIN_WIDTH%2 == 0) : 0,
    AC_WORDS_INSIDE = AC_WORDS - LPAD_WORDS,
    // Number of shifts required to fill up the shift register for horizontal
rampup = ceil(AC_WORDS_INSIDE/AC_BUS_WORDS)
    NUM_SHIFTS_FILL = AC_WORDS_INSIDE/AC_BUS_WORDS + int(AC_WORDS_INSIDE
%AC_BUS_WORDS != 0),
```

```
    // If the window has an even width, you're using REFLECT101 and if you're
passing a pixel at time, you'll need an extra word in the shift register
    // to ensure correct boundary processing later.
    EXTRA_WORD = USING_REFLECT101 && AC_WIN_WIDTH%2 == 0 && AC_BUS_WORDS == 1,
    AC_SHIFT_WORDS = NUM_SHIFTS_FILL*AC_BUS_WORDS + LPAD_WORDS + EXTRA_WORD
  };
};
```

Note that if boundary padding is not used, LPAD_WORDS = 0, AC_WORDS_INSIDE = AC_WORDS and EXTRA_WORD = 0.

4.2. Boundary Processor (ac_boundary)

Image windowing generally requires some type of boundary padding to make sure that the number of output windows is the same as the number of input pixels. The boundary processor block accepts inputs from the data and flag shifters to perform boundary padding. The data shifter provides the actual window data, while the flag shifter lets the boundary processing block know where the image boundaries lie so it can pad the boundary appropriately.

4.2.1. C++ Code Overview

A snippet of the ac_boundary_1d class is given below.

```
template <typename T, int AC_WIN_WIDTH, ac_padding_method AC_PMODE, int
AC_BUS_WORDS = 1, bool HIGH_SPEED_REPLICATE = false>
class ac_boundary_1d
{
public:
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  // Other code

private:
  T padded_val;
};
```

The padded_val private member variable stores the boundary value to be used when the window is set to implement constant padding. For any other boundary padding modes, this variable is ignored. It is set to zero by default.

A snippet of the ac_boundary_2d class is given below.

```
template <typename T, int AC_WIN_HEIGHT, int AC_WIN_WIDTH, ac_padding_method
AC_PMODE, int AC_BUS_WORDS = 1, bool HIGH_SPEED_REPLICATE = false>
class ac_boundary_2d
```

```
{
public:
  static_assert(AC_WIN_HEIGHT > 0, "AC_WIN_HEIGHT must be positive.");
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  // Other code

private:
  T padded_val;
};
```

This class also uses the padded_val data member to store the boundary value for constant padding, and this member is also set to zero by default.

ac_boundary_1d Template Parameters

The following list describes the template parameters for ac_boundary_1d.

- T – Specifies the base type, i.e. the type of each pixel element in the shift register. It must also contain a parameterized constructor that accepts integer arguments. Refer to Base Type Constructor Requirements for more details.
- AC_WIN_WIDTH – Window width, i.e. the number of columns in the window.
- AC_PMODE – Boundary padding mode. More details on this parameter are given in Boundary Processing Modes.
- AC_BUS_WORDS – Number of pixels sent in parallel over the bus interface. It is set to 1 by default.
- HIGH_SPEED_REPLICATE – If this parameter is set to true and replication is the chosen boundary mode, the architecture for boundary processing uses a shorter critical path with 1-hot MUXes instead of a chain of binary MUXes. Depending on the synthesis libraries used, window dimensions and the number of pixels packed in parallel, this can allow the user to synthesize the design at a higher frequency and still meet timing.

AC_WIN_WIDTH and AC_BUS_WORDS must be positive, failing which a static assertion is triggered.

ac_boundary_2d Template Parameters

ac_boundary_2d uses the same template parameters as ac_boundary_1d, along with an addition template parameter AC_WIN_HEIGHT which specifies the window height, i.e. the number of rows in the window. AC_WIN_HEIGHT, AC_WIN_WIDTH and AC_BUS_WORDS must all be positive, failing which a static assertion is triggered.

Constructors

Constructor	Description
`ac_boundary_1d()`	Default ac_boundary_1d constructor. Calls the set_pval() function mentioned

	in Miscellaneous ac_boundary_1d Methods to set the padded_val data member to 0.
`ac_boundary_1d(T pval)`	Parameterized ac_boundary_1d constructor. Calls the set_pval() function mentioned in Miscellaneous ac_boundary_1d Methods to set the padded_val data member to the pval input.
`ac_boundary_2d()`	Default ac_boundary_2d constructor. Calls the set_pval() function mentioned in Miscellaneous ac_boundary_2d Methods to set the padded_val data member to 0.
`ac_boundary_2d(T pval)`	Parameterized ac_boundary_2d constructor. Calls the set_pval() function mentioned in Miscellaneous ac_boundary_2d Methods to set the padded_val data member to the pval input.

Miscellaneous ac_boundary_1d Methods

Method	Description
`void run ( ac_array<T, AC_WORDS + EXTRA_WORD> din, ac_int<AC_WORDS, false> sol_flags, ac_int<AC_WORDS, false> eol_flags, ac_array<T, AC_WORDS> &dout)`	Performs horizontal boundary processing and writes the boundary padded output to dout. din is the output of the ac_shift_N_1d data shifter, while sol_flags and eol_flags are the outputs of the ac_flag_shift_1d flag shifter. A description on how AC_WORDS and EXTRA_WORDS are calculated is given in Calculating Shifter Size.
`void set_pval(T pval)`	Sets the padded_val data member to the pval input.

The ac_boundary_1d class has a nested struct with various specializations. Each specialization has a method that handles a different type of horizontal boundary padding. For brevity's sake, these aren't listed here, but the user can look at the code of ac_boundary.h for more details.

Miscellaneous ac_boundary_2d Methods

The following table describes the other ac_boundary_2d methods:

Method	Description
`void run( ac_array<T, AC_SHIFT_HEIGHT, AC_WORDS + EXTRA_WORD> din, ac_int<AC_WIN_HEIGHT, false> sof_flags, ac_int<AC_WIN_HEIGHT, false> eof_flags, ac_int<AC_WORDS, false> sol_flags, ac_int<AC_WORDS, false> eol_flags, ac_array<T, AC_WIN_HEIGHT, AC_WORDS> &dout)`	Performs horizontal and vertical boundary processing and writes the boundary padded output to dout. din is the output of the ac_shift_N_2d data shifter, while the sof_flags, eof_flags, sol_flags and eol_flags integer vectors are the outputs of the ac_flag_shift_2d flag shifter. The output after boundary processing is written to dout. A description on how AC_WORDS and EXTRA_WORDS are calculated is given in Calculating Shifter Size. The horizontal boundary processing is performed by iteratively calling objects of the ac_boundary_1d class.
`void set_pval(T pval)`	Sets the padded_val data member to the pval input.

The ac_boundary_2d class also has a nested struct with various specializations. Each specialization has a method that handles a different type of vertical boundary padding. For brevity's sake, these aren't listed here, but the user can look at the code of ac_boundary.h for more details.

4.2.2. Base Type Constructor Requirements

The base type is the type of the pixel elements in the data shifters. The boundary processing code requires that the base type support construction with the integer literal "0" as the input. To enable this, the base type definition must have a parameterized constructor that accepts an int argument. An example for a user-defined type is given in the snippet below:

```
struct user_def_type {
  ac_int<16, false> data[3][3]; // Actual data: 3x3 ac_int array.

  // Parameterized constructor: accepts int input and assigns it to all
  // elements of data array.
  user_def_type(int in) {
    #pragma hls_unroll yes
    INIT_TO_INT_ROW: for (int i = 0; i < 3; i++)
      #pragma hls_unroll yes
      INIT_TO_INT_COL: for (int j = 0; j < 3; j++)
        data[i][j] = in;
  }

  // Default constructor: Explicitly defined. This one's empty but feel
  // free to modify your default constructor as appropriate.
  user_def_type() {}
};
```

4.2.3. Boundary Processing Modes

The following table demonstrates the image padding methods supported by the ac_boundary classes:

Method	Examples of padded pixels		
	Pad	**Image**	**Pad**
Replicate	a a a a	a b c d e f g h i	i i i i
Reflect	d c b a	a b c d e f g h i	i h g f
Reflect101	e d c b	a b c d e f g h i	h g f e
Constant	m m m m	a b c d e f g h i	m m m m

Note that this example assumes that 4 pixels are padded to the left and the right of the image, which is the case when AC_WIN_WIDTH = 9.

In addition to the above padding modes, AC Window 2.0 also supports cases with no boundary padding. In cases like this, any individual window must be completely inside the image to be valid.

The AC_PMODE template parameter determines which boundary padding method is used. This template parameter is a value from the ac_padding_method enumeration, which is defined in the ac_boundary_enums.h header file as follows:

```
enum ac_padding_method {
  AC_NO_PADDING,
  AC_REPLICATE,
  AC_REFLECT101,
  AC_CONSTANT,
  AC_REFLECT
};
```

If the constant boundary padding is chosen, the value m in the table above is the same as the padded_val data member. Note that you can specify AC_NO_PADDING for cases where no image padding is to be used. If no padding is used, the input is simply passed through to the output.

4.3. Output Flag Generator for Standard Windowing (ac_flag_gen_out)

Since the overlapped window needs to ramp up, there is a period of delay between when the inputs are sent in and when windowed outputs are produced. The Windowed Output Flag Generator takes that lag into account and generates various flags that give the user more information on the validity and position of the output windows.

4.3.1. C++ Code Overview

A snippet of the ac_flag_gen_out_1d class is given below.

```
template<int AC_WIN_WIDTH, int AC_BUS_WORDS = 1, bool USING_PADDING = true,
bool IS_STATEFUL = false>
class ac_flag_gen_out_1d {
public:
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");
  typedef ac_int<AC_BUS_WORDS, false> VOUT_TYPE;

  // Other code

private:
  bool ramped_up;
};
```

The IS_STATEFUL flag determines whether the 1D output flag generator is stateful or not. This template parameter and the ramped_up flag are described in more detail in ac_flag_gen_out_1d Template Parameters.

The ramped_up data member is set to true when the 1D window has ramped up. It is set to false once the windowed outputs are no longer valid. When we're not using padding and outputting multiple individual windows (AC_BUS_WORDS > 1), we may get a case where some of the individual windows are valid and some aren't at the EOL output. In such a case, a simple boolean vld_out flag is not enough, and we need a vld_out that's of type ac_int<AC_BUS_WORDS, false> so as to accommodate multiple flag bits (assuming AC_BUS_WORDS > 1), as specified by the VOUT_TYPE member type shown in the snippet. To keep the vld_out representations consistent between the implementations which do and do not use boundary padding, VOUT_TYPE remains the same. More information on how the vld_out flag bits are computed is given in Output Flags without Padding.

A snippet of the ac_flag_gen_out_2d class is given below.

```
template<int AC_WIN_HEIGHT, int AC_WIN_WIDTH, int AC_BUS_WORDS = 1, bool
USING_PADDING = true>
class ac_flag_gen_out_2d {
public:
  static_assert(AC_WIN_HEIGHT > 0, "AC_WIN_HEIGHT must be positive.");
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");
  static_assert(!USING_PADDING || AC_WIN_HEIGHT >= 3, "AC_WIN_HEIGHT must not
be less than 3, if boundary padding is used.");
  typedef ac_flag_gen_out_1d<AC_WIN_WIDTH, AC_BUS_WORDS, USING_PADDING>
LINE_TYPE;
  typedef typename LINE_TYPE::VOUT_TYPE VOUT_TYPE;

  // Other code.

private:
  LINE_TYPE line_np;
  bool ramped_up_v;
};
```

The vld_out type (VOUT_TYPE) is the same as that used by the 1D flag generator, i.e. ac_int<AC_BUS_WORDS, false>.

This class also has a ramped_up_v data member that is set to true once the design is at the SOF window. If we're using boundary padding, the window will remain ramped up until the design gets to the EOF window. As a result, we don't need to store state for the horizontal rampup and we don't need persistant storage for the 1D flag generator object. It is hence declared as a variable local to the run function described in Miscellaneous ac_flag_gen_out_2d Methods. However, if no boundary padding is used, there are periods between the output lines where none of the individual windows are inside the image, and the windowed data is hence invalid. In such a case, we do need to store state for the horizontal rampup and have persistent storage for it in the form of the line_np data member.

ac_flag_gen_out_1d Template Parameters

- AC_WIN_WIDTH – Specifies the window width, i.e. the number of columns in the window.
- AC_BUS_WORDS – Specifies the number of input pixels sent in parallel on the data bus.

- USING_PADDING – Specifies whether boundary padding is used or not.
- IS_STATEFUL – Specifies whether the generator is stateful and the ramped_up data member is used.

AC_WIN_WIDTH and AC_BUS_WORDS must be positive, failing which a static assertion is triggered.

The 1D output flag generator should be stateful in the following cases:

(a) 1D Windowing is used.

(b) 2D Windowing is used **AND** USING_PADDING = false **AND** AC_WIN_WIDTH - 1 >= AC_BUS_WORDS.

In both the cases described above, there are blank periods around the start and end of line positions where the window doesn't produce valid outputs. Condition (a) has blank periods because 1D windows need to ramp up at the start of each line. Condition (b) has blank periods because all of the output windows being produced in such a case are outside the image for one more iterations around the start of each line. In either case, the ramped_up flag is used to keep track of these blank periods and is set to true only when we're no longer in them.

For all other conditions, ac_flag_gen_out_1d is being used to generate output flags for a 2D window. In such cases, the outputs windows are completely valid when the first output window for that frame is produced, and the windows are either partially or completely valid until after the last output window for that frame is produced. ac_flag_gen_out_2d can keep track of this information. In addition, the windows are only partially valid when USING_PADDING = false and eol_out = 1 (more information on eol_out is given in Miscellaneous ac_flag_gen_out_1d Methods). Both these factors can be determined without using the ramped_up flag, and ac_flag_gen_out_1d does not need to have state.

ac_flag_gen_out_2d Template Parameters

The ac_flag_gen_out_2d class uses the same template parameters as the ac_flag_gen_out_1d class, in addition to the template parameter AC_WIN_HEIGHT which specifies the window height, i.e. the number of rows in the window. AC_WIN_HEIGHT must also be positive, failing which a static assertion is triggered.

The 2D output flag generator currently only supports windows that are at least 3 rows tall if boundary padding is used, i.e. AC_WIN_HEIGHT must never be less than 3 while using boundary padding. This is done to avoid an out-of-bounds array access while computing the eof_out flag.

If boundary padding is not used, any positive value of AC_WIN_HEIGHT is allowed, resources permitting.

Constructors

Constructor	Description
`ac_flag_gen_out_1d()`	Default ac_flag_gen_out_1d() constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_gen_out_1d Methods to set the ramped_up member variable to false.
`ac_flag_gen_out_2d()`	Default ac_flag_gen_out_2d() constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_gen_out_2d Methods to set the ramped_up member variable to false.

Miscellaneous ac_flag_gen_out_1d Methods

Method	Description
`void run ( ac_int<AC_WORDS, false> sol_flags, ac_int<AC_WORDS, false> eol_flags, bool &sol_out, bool &eol_out, VOUT_TYPE &vld_out)`	Accepts the sol_flags and eol_flags outputs from the 1D flag shifter and produces sol_out, eol_out and vld_out flags. More information on these flags is given in Output Flags with Padding and Output Flags without Padding. Calculating Shifter Size describes how the AC_WORDS parameter is calculated.
`void reset()`	Sets the ramped_up member variable to false.

Miscellaneous ac_flag_gen_out_2d Methods

Method	Description
`void run ( ac_int<AC_WIN_HEIGHT, false> &sof_flags, ac_int<AC_WIN_HEIGHT, false> &eof_flags, ac_int<AC_WORDS, false> sol_flags, ac_int<AC_WORDS, false> eol_flags, bool &sof_out, bool &eof_out, bool &sol_out, bool &eol_out, VOUT_TYPE &vld_out, ac_int<2, false> eof_vals = 0)`	Accepts the sof_flags, eof_flags, sol_flags and eol_flags outputs from the 1D flag shifter and produces sof_out, eof_out, sol_out, eol_out and vld_out flags. More information on these flags is given in Output Flags with Padding and Output Flags without Padding. Calculating Shifter Size describes how the AC_WORDS parameter is calculated. eof_vals is needed when no padding is used. The first bit is the EOF flag as generated by the input flag generator, while the second bit is the EOF flag seen on the previous iteration, i.e. the eof_old parameter stored in the 2D flag generator and returned by its run function when needed. Refer to C++ Code Overview for Flag Shifters for more details.
`void reset()`	Sets the ramped_up member variable to false.

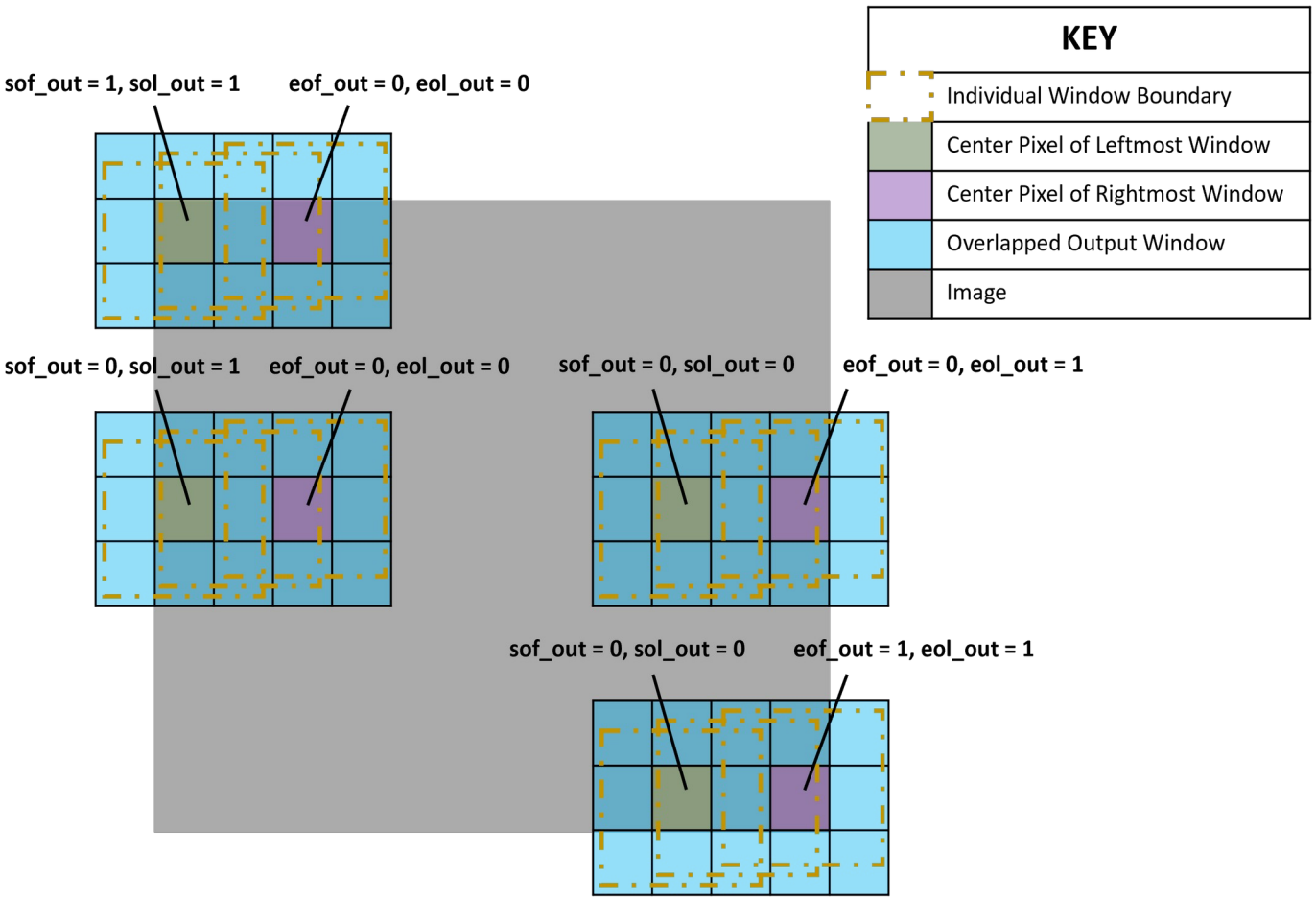

Illustration 50: Output Flags for Different Window Positions, with Boundary Padding

4.3.2. Output Flags with Padding

As mentioned earlier, the AC Window 2.0 design can output multiple overlapped windows. If boundary padding is used, the window ramps up when the leftmost window ramps up, and it ramps down when the rightmost window ramps down.

Taking this into account, the position flags generated by ac_flag_gen_out_2d can be described as follows:

- sof_out: Set to true when the center pixel of the first individual window is the SOF pixel.
- eof_out: Set to true when the center pixel of the last individual window is the EOF pixel.
- sol_out: Set to true when the center column of the first individual window coincides with the SOL.
- eol_out: Set to true when the center column of the last individual window coincides with the EOL.

Refer to Illustration 50 for a representation of how these flags change at different window positions, for three overlapped 3x3 output windows.

The vld_out flags are all set to true—in other words, vld_out is all 1s—when the overlapped windows have collectively ramped up, and set to false—in other words, vld_out is all 0s—when the overlapped windows have collectively ramped down.

While this description is applicable to the 2D output flag generator, the 1D generator also produces outputs similarly. The only major difference is that the 1D generator does not produce sof_out and eof_out output flags because it is designed to work with a single line of inputs and not a 2D image.

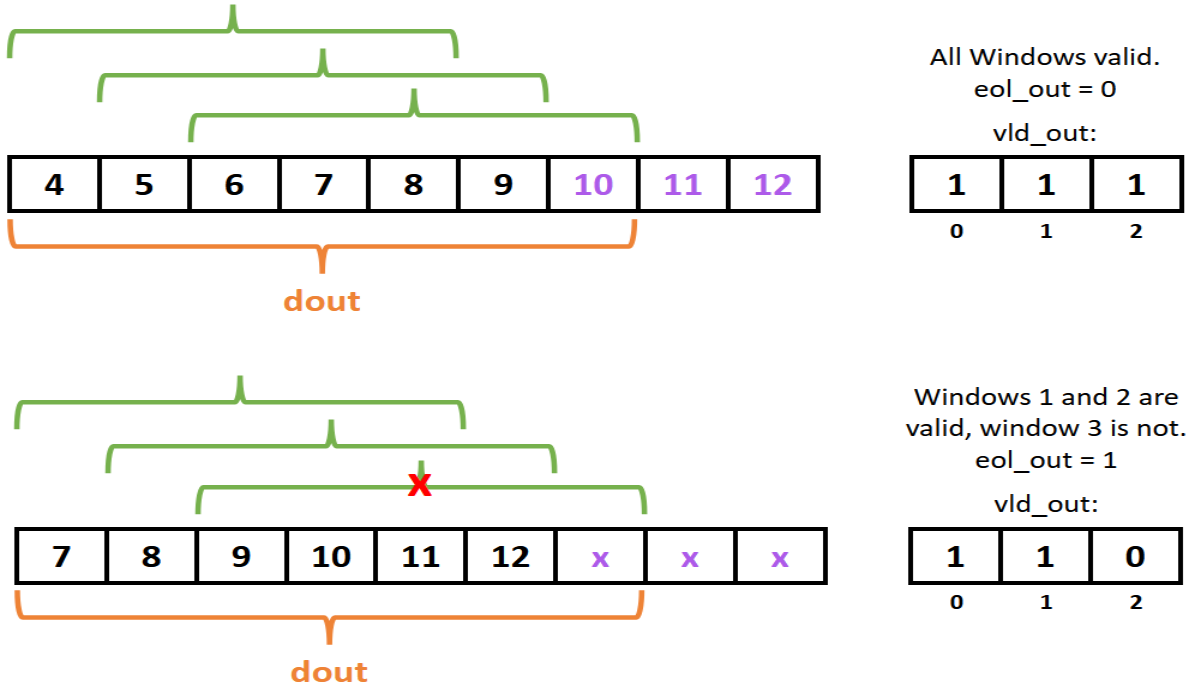

Illustration 51: vld_out Generation at EOL, without Padding

4.3.3. Output Flags without Padding

When no padding is implemented, an individual window must be inside the image to be valid. This changes the positional significance of the output flags. The sol_out flag now corresponds to the leftmost column of the first individual window, while the sof_out flag corresponds to the top left pixel of the first individual window.

The eol_out and eof_out flags are a little more complicated and are explained as follows.

As mentioned in Shifter Size without Boundary Padding, we may run into scenarios where all windows are not valid at the EOL position. If we take into account the same scenario presented in that example (AC_WIN_WIDTH = 5, AC_BUS_WORDS = 3, input image width = 12), we can see that the first and second windows are valid at the EOL position, but not the third one, thanks to the fact that AC_WORDS is not divisible by AC_BUS_WORDS. To represent this, we set the first and second bits of vld_out to 1, while the third bit is set to 0. In other words, vld_out = (011)2. This is diagrammatically represented in Illustration 52. Note that for the diagrammatic representation of vld_out, we're starting with the LSB and ending with the MSB, as opposed to starting with the MSB like what's generally done for bitwise representations of integers. This is done in order to make sure that the diagrammatic representation of vld_out is consistent with that of the shift register, which also starts with lower indices. Note that eol_out corresponds to the rightmost pixel of the last valid window for the line.

Also note that eol_out lags behind eol by a cycle, i.e. the output EOL window is produced a cycle after the EOL pixel is received. While this lag can be accommodated for by indexing into the eol_flags register appropriately, we also get a corresponding lag for the EOF pixel (assuming that we're using 2D windows) which cannot be resolved that way due to the way the eof_flags register is updated. We must instead look at the eof_old output from the flag register, which is obtained from the eof_vals input to the run function, as explained in Miscellaneous ac_flag_gen_out_2d Methods.

Note that the mixing of valid and invalid windows, as well as the lag in the eol_out/eof_out signal are both not applicable if AC_WORDS is divisible by AC_BUS_WORDS. In such a case, the last column of the last individual window aligns with the EOL pixel, and all the windows are either valid or invalid at the same time. Hence, vld_out is all 0s or all 1s. The eof_out flag is generated directly with the eof flag from the input flag generator.

Generalizing this, the position flags generated by ac_flag_gen_out_2d can be described as follows:

- sof_out: Set to true when the top left pixel of the first window is the SOF pixel.
- eof_out: Set to true when the bottom right pixel of the last valid window is the EOF pixel.
- sol_out: Set to true when the left column of the first window coincides with the SOL.
- eol_out: Set to true when the right column of the last valid window coincides with the EOL.

Refer to Illustration 48 for a representation of how these flags, along with the vld_out flags, change at different window positions, for three overlapped 3x3 windows. Note that in this case, only the first window is valid at the EOL position, while the second and third are invalid.

From a C++ coding perspective, which window is valid and which isn't can be determined by using the ac_int subscript operator on vld_out. This is explained in more detail in Using the vld_out Signal when Boundary Padding is Disabled.

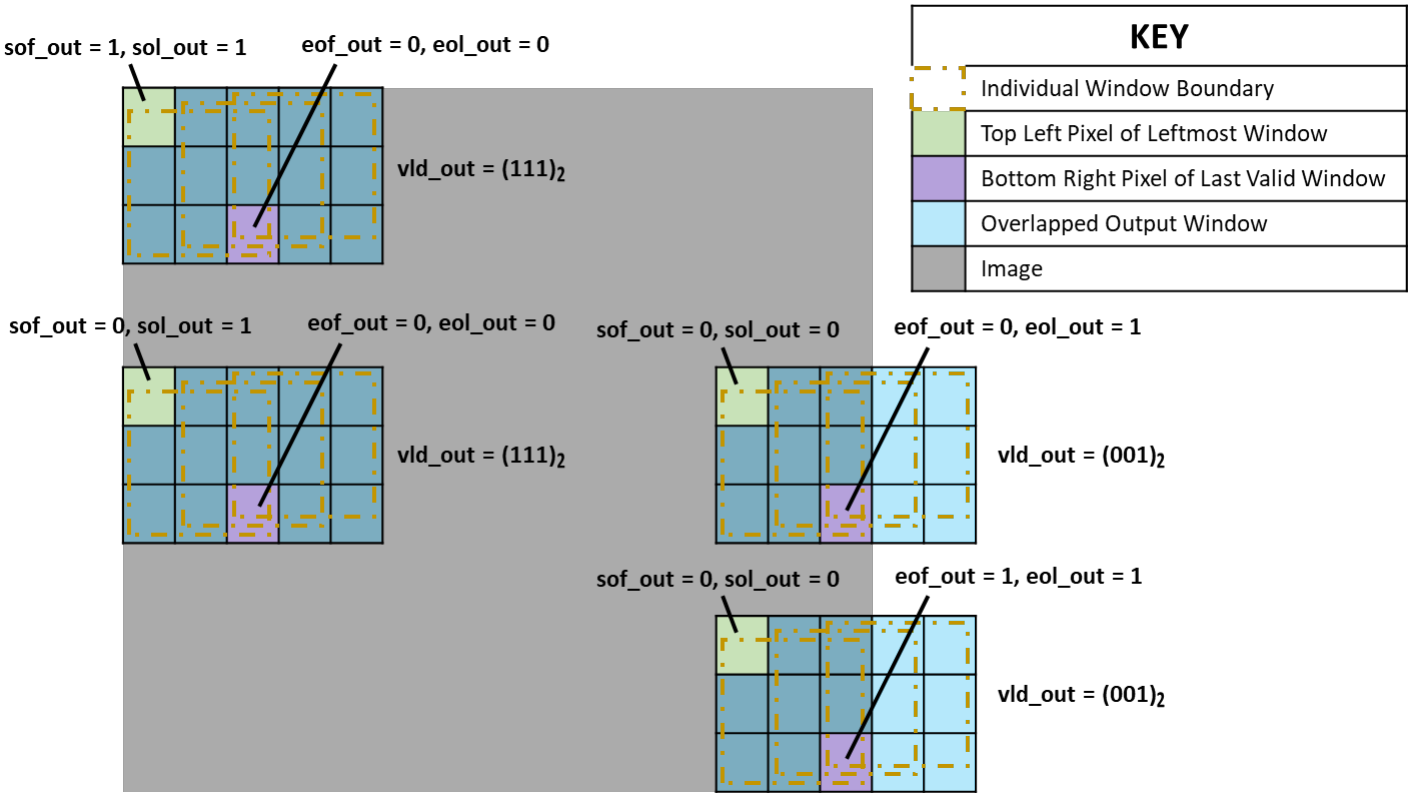

Illustration 52: Output Flags for Different Window Positions, without Boundary Padding

4.4. Top-Level Design for Standard Windowing (ac_window_v2)

All the sub-blocks used by Standard Windowing are instantiated and integrated by the top-level design. The design is meant to be pipelined with an II of 1 and can accept multiple inputs/produce multiple outputs for enhanced throughput.

4.4.1. C++ Code Overview

A snippet of the ac_window_v2_1d class is given below.

```
template <class PIX_TYPE, int AC_IMG_WIDTH, int AC_WIN_WIDTH, ac_padding_method
AC_PMODE, int AC_BUS_WORDS = 1, bool HIGH_SPEED_REPLICATE = false>
class ac_window_v2_1d {
  typedef ac_int<ac::nbits<AC_IMG_WIDTH>::val, false> W_TYPE;
  typedef ac_int<AC_BUS_WORDS, false> VOUT_TYPE;

  enum { USING_PADDING = AC_PMODE != AC_NO_PADDING };

  // Other code
```

```
private:
  ac_flag_gen_1d<AC_IMG_WIDTH, AC_BUS_WORDS> fgen_1d;
  ac_flag_shift_1d<AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS> fshift_1d;
  ac_shift_N_1d<PIX_TYPE, AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS> dshift_1d;
  ac_flag_gen_out_1d<AC_WIN_WIDTH, AC_BUS_WORDS, USING_PADDING, true>
fgen_out_1d;
};
```

The following sub-blocks are instantiated as members of the class:

- 1D Synchronization Flag Generator – Instantiated as fgen_1d.
- 1D Flag Shifter – Instantiated as fshift_1d.
- 1D Data Shifter – Instantiated as dshift_1d.
- 1D Windowed Output Flag Generator – fgen_out_1d.

Since the 1D Boundary Processor does not need any static storage, it is instantiated as a variable local to the run() function described in Miscellaneous ac_window_v2_1d Methods.

W_TYPE is the type of the width input to the run() function. VOUT_TYPE is the type of the vld_out output of the run() function.

A snippet of the ac_window_v2_2d class is given below.

```
template <class PIX_TYPE, int AC_IMG_HEIGHT, int AC_IMG_WIDTH, int
AC_WIN_HEIGHT, int AC_WIN_WIDTH, ac_buff_type BUFF_TYPE, ac_padding_method
AC_PMODE, int AC_BUS_WORDS = 1, bool HIGH_SPEED_REPLICATE = false, bool AC_RE-
PEAT = false>
class ac_window_v2_2d {
public:
  typedef ac_int<ac::nbits<AC_IMG_WIDTH>::val, false> W_TYPE;
  typedef ac_int<ac::nbits<AC_IMG_HEIGHT>::val, false> H_TYPE;
  typedef ac_int<AC_BUS_WORDS, false> VOUT_TYPE;

  typedef ac_boundary_2d<PIX_TYPE, AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_PMODE,
AC_BUS_WORDS, HIGH_SPEED_REPLICATE> BOUND_2D_TYPE;

  enum {
    AC_WORDS = BOUND_2D_TYPE::AC_WORDS,
    EXTRA_WORD = BOUND_2D_TYPE::EXTRA_WORD,
    AC_SHIFT_HEIGHT = BOUND_2D_TYPE::AC_SHIFT_HEIGHT,
    USING_PADDING = AC_PMODE != AC_NO_PADDING
  };

  typedef ac_packed_vector<PIX_TYPE, AC_BUS_WORDS> LB_IN_TYPE;
```

```
  typedef ac_array<PIX_TYPE, AC_SHIFT_HEIGHT, AC_BUS_WORDS> LB_OUT_TYPE;
  typedef ac_linebuffer<LB_IN_TYPE, LB_OUT_TYPE, AC_IMG_WIDTH, AC_SHIFT_HEIGHT,
BUFF_TYPE, AC_REPEAT> LB_TYPE;

  // Other code.

private:
  ac_flag_gen_2d<AC_IMG_WIDTH, AC_IMG_HEIGHT, AC_BUS_WORDS> fgen_2d;
  LB_TYPE linebuf_2d;
  ac_flag_shift_2d<AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS, AC_RE-
PEAT> fshift_2d;
  ac_shift_N_2d<PIX_TYPE, AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS>
dshift_2d;
  ac_flag_gen_out_2d<AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_BUS_WORDS> fgen_out_2d;
};
```

The following sub-blocks are instantiated as members of the class:

- 2D Synchronization Flag Generator – Instantiated as fgen_2d.
- Linebuffer – Instantiated as linebuf_2d.
- 2D Flag Shifter – Instantiated as fshift_2d.
- 2D Data Shifter – Instantiated as dshift_2d.
- 2D Windowed Output Flag Generator – fgen_out_2d.

Since the 2D Boundary Processor does not have state, it does not need any persistent storage. It is instantiated as a variable local to the run() function described in Miscellaneous ac_window_v2_2d Methods.

Note that while most of the sub-blocks instantiated for 2D windowing are the same as the 1D counterpart, the 2D design also instantiates the linebuffer class, as we need to store multiple image lines to implement 2D windowing. The linebuffer is instantiated using the LB_IN_TYPE and LB_OUT_TYPE typedefs to specify the input and output to the linebuffer.

W_TYPE and H_TYPE are the types used for the width and height inputs to the run() function, respectively. VOUT_TYPE is the type for the vld_out output of the same function.

ac_window_v2_1d Template Parameters

The following list describes the 1D template parameters.

- PIX_TYPE – Pixel type. It must contain a parameterized constructor that accepts integer arguments, to support typecasting in the boundary processor. Refer to Base Type Constructor Requirements for more details.
- AC_IMG_WIDTH – Maximum possible image width.

- AC_WIN_WIDTH – Window width, i.e. number of columns in the window.
- AC_PMODE – Boundary processing mode. For more details, refer to Boundary Processing Modes.
- AC_BUS_WORDS – Number of pixels packed in parallel on the data bus.
- HIGH_SPEED_REPLICATE – Refer to ac_boundary_1d Template Parameters for more details.

ac_window_v2_2d Template Parameters

The following list describes the 2D template parameters.

- PIX_TYPE – Pixel type. This type must be compatible with the ac_packed_vector container class. It must also contain a parameterized constructor that accepts integer arguments, to support typecasting in the boundary processor. Refer to Base Type Constructor Requirements for more details.
- AC_IMG_HEIGHT – Maximum possible image height.
- AC_IMG_WIDTH – Maximum possible image width.
- AC_WIN_HEIGHT – Window height, i.e. number of rows in the window.
- AC_WIN_WIDTH – Window width, i.e. number of columns in the window.
- BUFF_TYPE – Specifies the linebuffer architecture used.
- AC_PMODE – Boundary processing mode. For more details, refer to Boundary Processing Modes.
- AC_BUS_WORDS – Number of pixels packed in parallel on the data bus.
- HIGH_SPEED_REPLICATE – Refer to ac_boundary_1d Template Parameters for more details.
- AC_REPEAT – This parameter is set to true if the window is expected to iterate over the same set of lines for algorithms such as interpolation.

Constructors

Constructor	Description
`ac_window_v2_1d()`	Default ac_window_v2_1d() constructor. Calls the reset() function mentioned in Miscellaneous ac_window_v2_1d Methods and also sets the padded_val value used by the boundary processor to 0.

`ac_window_v2_1d(`
`  PIX_TYPE pval`
`)`
	Parameterized ac_window_v2_1d() constructor. Calls the reset() function mentioned in Miscellaneous ac_window_v2_1d Methods and also sets the padded_val value used by the boundary processor to pval.

`ac_window_v2_2d()`	Default ac_window_v2_2d() constructor. Calls the reset() function mentioned in Miscellaneous ac_window_v2_2d Methods and also sets the padded_val value used by the boundary processor to 0.

`ac_window_v2_2d(`
`  PIX_TYPE pval`
`)`
	Parameterized ac_window_v2_2d() constructor. Calls the reset() function mentioned in Miscellaneous ac_window_v2_2d Methods and also sets the padded_val value used by the boundary processor to pval.

Miscellaneous ac_window_v2_1d Methods

Method	Description

`void run(`
`  ac_array<PIX_TYPE, AC_BUS_WORDS> din,`
`  W_TYPE width,`
`  bool &read_data,`
`  ac_array<PIX_TYPE, AC_WORDS> &dout,`
`  bool &sol_out,`
`  bool &eol_out,`
`  VOUT_TYPE &vld_out)`
	Performs 1D Windowing. din is the input pixel data. The width dimension input specifies the width of the line. read_data is an output flag that can be used to gate against input reads when the window is flushing outputs and cannot accept image inputs.
	dout is the overlapped output window from the boundary processor. sol_out, eol_out and vld_out are the output flags from the output flag generator.

void reset()	Calls the reset() methods of all the sub-blocks except for the boundary processor.

Miscellaneous ac_window_v2_2d Methods

Method	Description

`void run (`
`  ac_array<PIX_TYPE, AC_BUS_WORDS> din,`
`  W_TYPE width, H_TYPE height,`
`  bool &read_data,`
`  ac_array<PIX_TYPE, AC_WIN_HEIGHT,`
`  AC_WORDS> &dout,`
`  bool &sof_out, bool &eof_out,`
`  bool &sol_out, bool &eol_out,`
`  VOUT_TYPE &vld_out,`
`  bool repeat_line = false)`
	Performs 2D Windowing. din is the input pixel data. The width and height dimension inputs specify the width and height of the image, respectively.
	read_data is a flag that can be used to gate against input reads when the window is flushing outputs and cannot accept image inputs.
	dout is the overlapped output window from the boundary processor. sof_out, eof_out, sol_out, eol_out and vld_out are the outputs of the output flag generator. Refer to Line Repetition for a detailed description of the repeat_line input.

`void reset_flags()`	Resets various flags internal to the AC Window 2.0 design. If AC_REPEAT = false, reset_flags() is called implicitly by the run() function once the design encounters the output EOF. If AC_REPEAT = true, reset_flags() must be called explicitly by the user before they send in a frame for processing. Refer to Line Repetition for more details.
`void reset()`	Calls the reset() methods of all the sub-blocks except for the boundary processor. Also resets the class member variables.

4.4.2. Block Diagram

Refer to Illustration 53 for a block diagram showing how the sub-blocks are integrated in the top-level AC Window Design for 2D windowing.

The pixel inputs are fed to the linebuffer. The linebuffer outputs multiple lines of buffered data, which are fed to the data shifter. The input flag generator is fed the input dimensions and produces flags for the input image. The flag shifter shifts and aligns these flags with the output of the data shifter. The boundary processing block uses the shifted flags to ascertain where the image boundaries are, so that it can perform boundary processing (if applicable) on the shifted data to produce the windowed data output. The output flag generator accepts the shifted flags and produces output flags that give the user information regarding the validity and the position of the windowed data.

1D windowing works similarly, with the only major differences being:

- 1D Windowing doesn't use a linebuffer as it works on a single line of inputs.
- The output window is only one row tall.
- No SOF or EOF flag information is needed for the input or for the output.

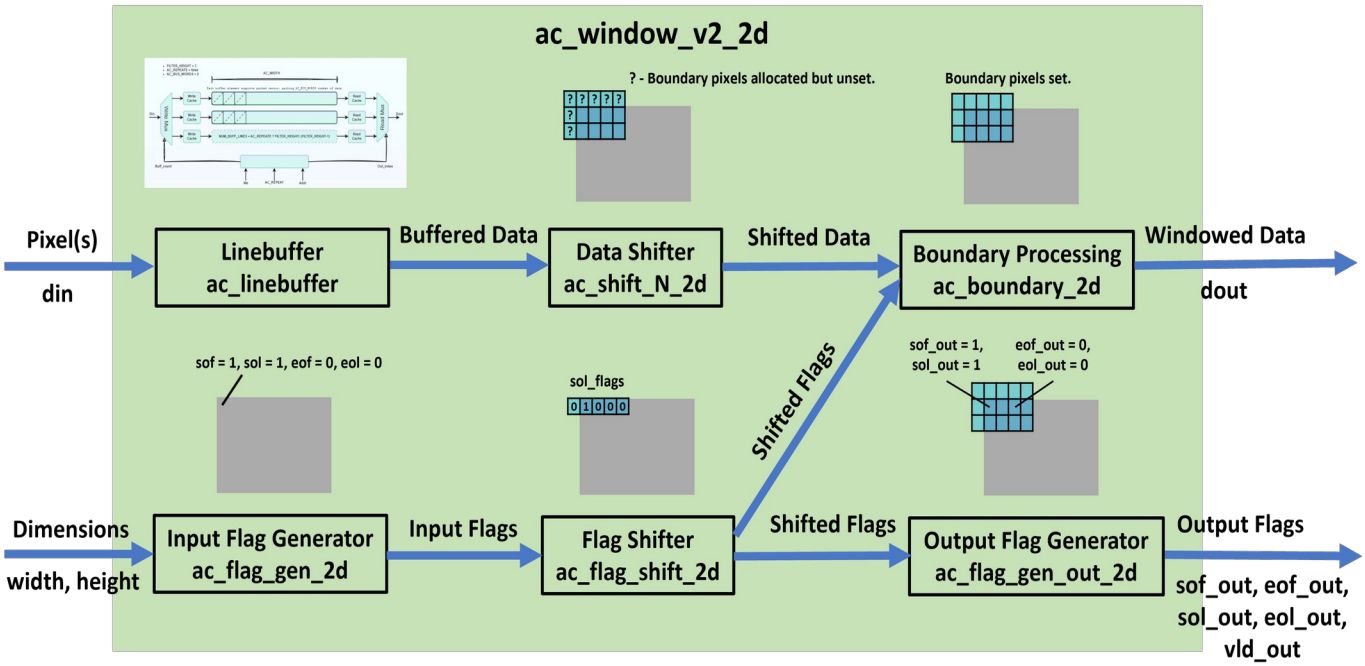

Illustration 53: Block Diagram for Standard Windowing

4.4.3. Line Repetition

Image scaling and interpolation in two dimensions require the window to iterate over the same set of lines when the height of the resized output image is greater than the height of the original image. To accommodate this requirement, the window must not slide vertically while it goes over the repeated lines. To stop the window from sliding vertically in such a case, the design does the following:

- The only_update_line input to the input flag generator is set to false. The flag generator does not update the SOF and EOF flags.

- The write enable (“we”) input to the linebuffers is set to false. The linebuffers do not accept new inputs and only output previously stored data.

- The repeat_line input to the flag shifter block is set to true. Like the linebuffers, the vertical flag shifters, i.e. the sof_regs and eof_regs data members in ac_flag_shift_2d, do not accept new inputs.

Support for line repetition calls for unique hardware requirements that are not expected when performing standard 2D convolution. Setting the AC_REPEAT template parameter to true will ensure that the appropriate hardware is synthesized. The design must also know when to repeat the same set of lines and when to process new lines at run-time. If the repeat_line input to the run() function is set to true, the design will perform line repetition. If it is set to false, the design will process new lines, as per its default behavior.

If AC_REPEAT = true, it is assumed that the design is resizing the input image. In such a case, the output EOF might not be a reliable indicator of when to call reset_flags() and the design does not implicitly call it as a result. The user must explicitly call reset_flags() right before they send a frame in for processing.

Setting repeat_line to true without setting AC_REPEAT to true first is invalid, as it would imply that the user expects the design to perform line repetition despite not having synthesized the appropriate hardware. A software assert (AC_ASSERT) is provided to guard against this condition during C simulation.

4.4.4. Using the vld_out Signal when Boundary Padding is Disabled

As mentioned in Output Flags without Padding, the overlapped output window may have a mix of valid and invalid individual windows, if padding is disabled. Consider a case with the following parameters:

- 8-bit unsigned ac_int input and output pixels.

- 5x5 window.

- AC_SPWRMASK linebuffers.

- AC_BUS_WORDS = 3.

- 24x24 input image. Note that the width must be a multiple of 2*AC_BUS_WORDS = 6, since singleport linebuffers are used.

- Inputs and outputs are streamed on ac_channels. din_ch is the input channel while dout_ch is the output channel.

In such a case, AC_WORDS = AC_WIN_WIDTH + AC_BUS_WORDS - 1 = 5 + 3 - 1 = 7. Since AC_WORDS is not divisible by AC_BUS_WORDS, not all the windows are valid when eol_out = true. To determine which windows are valid, we use the subscript operator on vld_out, as shown in the snippet below:

```
ac_window_v2_2d<uint8, 24, 24, 5, 5, AC_SPWRMASK, AC_NO_PADDING, 3> window;
constexpr int AC_WORDS = window.AC_WORDS;
bool in_read = true, eof_out = false;

do {
  typedef ac_array<uint8, 3> DIN_TYPE;
```

```
  DIN_TYPE din = in_read ? din_ch.read() : DIN_TYPE(0);

  // Call window's run() function.
  ac_array<uint8, 5, AC_WORDS> window_out;
  bool sof_out, eof_out, sol_out, eol_out;
  ac_int<3, false> vld_out;
  window.run(din, 24, 24, in_read, window_out, sof_out, eof_out, sol_out,
eol_out, vld_out);

  if (vld_out) {
    ac_array<uint8, 3> dout;
    for (int k = 0; k < 3; k++) { // Iterate over the windows.
      // Use the subscript operator to examine vld_out. The kth window is
      // valid if kth bit of vld_out is 1.
      if (vld_out[k]) {
        // Perform convolution on the window, write the output to dout_temp.
        dout[k] = dout_temp;
      } else {
        // Since this window is invalid, fill the corresponding dout index
        // with a dummy value (zero in this case).
        // Do not leave it uninitialized.
        dout[k] = 0;
      }
    }
    dout_ch.write(dout);
  }
} while (!eof_out);
```

Not using boundary padding means that the input and output dimensions are not the same (assuming that we're not using a 1x1 window). For a generalized case, the output dimensions, including the total number of output pixels, can be computed as shown below:

```
// Finding the max possible output image height and width let us constrain the
// bitwidths for the out_height and out_width variables, if needed.
constexpr int MAX_OUT_HEIGHT = AC_IMG_HEIGHT - (AC_WIN_HEIGHT - 1);
constexpr int MAX_OUT_WIDTH = AC_IMG_WIDTH - (AC_WIN_WIDTH - 1);
constexpr int OUTH_BITS = ac_nbits<MAX_OUT_HEIGHT>::val;
constexpr int OUTW_BITS = ac_nbits<MAX_OUT_WIDTH>::val;

// Compute the output height and width.
ac_int<OUTH_BITS, false> out_height = height - (AC_WIN_HEIGHT - 1);
ac_int<OUTW_BITS, false> out_width = width - (AC_WIN_WIDTH - 1);
```

In this case, out_height = out_width = 24 - (5 - 1) = 20. If you don't need to store the output width and height in bit-accurate variables – which is probably the case if you're computing them in your testbench or other

code that isn't synthesized – you can also simply store them in int variables as shown below, assuming that height and width are ac_int variables which are appropriately bitwidth-constrained:

```
int out_height = height.to_int() - (AC_WIN_HEIGHT - 1);
int out_width = width.to_int() - (AC_WIN_WIDTH - 1);
```

4.4.5. Architectural Overview

The AC Window 2.0 design is meant to be pipelined with an II of 1. All the loops internal to the AC Window 2.0 sub-blocks are fully unrolled via pragmas to allow for this pipelining and maximize throughput. The linebuffer sub-block also lets the user pick between different linebuffer architectures, such as singleport memories with write-masking. The boundary processor lets the user choose an architecture that can potentially enable the use of higher clock frequencies for AC_REPLICATE boundary padding, through the HIGH_SPEED_REPLICATE template parameter described in ac_boundary_1d Template Parameters.

4.4.6. Limitations

- If using 2D windows, the pixel type must be compatible with the ac_packed_vector container class. Currently, only the ac_int, ac_fixed and RGB_pv (in ac_pixels.h) datatypes as well as the ac_packed_vector type itself have this compatibility. Support for floating point datatypes might be provided with a future release.
- Windows less than three rows tall are not supported if boundary padding is used, i.e. AC_WIN_HEIGHT must not be less than 3 in such a case.
- The design currently only supports a stride of 1. Strides greater than 1 might be supported in a future release.

4.5. Linebuffers for Flush Support (ac_linebuffer_flush)

Flushing support requires a different method of accessing linebuffer elements and hence a different interface than the one used by the ac_linebuffer blocks. Hence, a separate set of linebuffers was designed for this purpose.

4.5.1. C++ Code Overview

Like the other set of linebuffers, i.e. those which don’t have flush support enabled, the top-level linebuffer class (ac_linebuffer_flush) is a wrapper which instantiates a specific linebuffer architecture based on a class template parameter. A snippet of this class is given below.

```
template<typename INPUT_TYPE, typename OUTPUT_TYPE, int AC_WIDTH, int
AC_NUM_LINES, ac_buff_arch_flush AC_BUFF_ARCH, bool AC_REPEAT = false>
class ac_linebuffer_flush
{
public:
  enum {
    AC_BUS_WORDS = INPUT_TYPE::packed_words,
    PACKED_WIDTH = AC_WIDTH/AC_BUS_WORDS,
```

```
  };
  typedef typename INPUT_TYPE::base_type BASE_TYPE;
  typedef ac_int<ac::nbits<PACKED_WIDTH - 1>::val, false> ADDR_TYPE;
  static_assert(AC_WIDTH > 0, "AC_WIDTH must be positive.");
  static_assert(AC_WIDTH%AC_BUS_WORDS == 0, "AC_WIDTH should be divisible by
AC_BUS_WORDS.");

  // Class methods

private:
  // The DUMMY template parameter ensures that buffer_type_struct is always
  // partially specialized and that the code will pass C++ compilation.
  template <bool DUMMY, ac_buff_arch_flush BUFF_ARCH>
  struct buffer_type_struct {};

  template <bool DUMMY>
  struct buffer_type_struct<DUMMY, AC_SPWRMASK_FLUSH> {
    static_assert(PACKED_WIDTH%2 == 0, "AC_WIDTH should be divisible by
2*AC_BUS_WORDS if you're using singleport buffers.");
    typedef ac_linebuffer_spwrmask_flush<INPUT_TYPE, PACKED_WIDTH,
AC_NUM_LINES, AC_REPEAT> BUFF_TYPE;
  };

  template <bool DUMMY>
  struct buffer_type_struct<DUMMY, AC_SPCIR_FLUSH> {
    static_assert(PACKED_WIDTH%2 == 0, "AC_WIDTH should be divisible by
2*AC_BUS_WORDS if you're using singleport buffers.");
    typedef ac_linebuffer_spcircular_flush<INPUT_TYPE, PACKED_WIDTH,
AC_NUM_LINES, AC_REPEAT> BUFF_TYPE;
  };

  template <bool DUMMY>
  struct buffer_type_struct<DUMMY, AC_1R1W_FLUSH> {
    typedef ac_linebuffer_1r1w_flush<INPUT_TYPE, PACKED_WIDTH, AC_NUM_LINES,
AC_REPEAT> BUFF_TYPE;
  };

  static constexpr bool DUMMY = false;
  typedef typename buffer_type_struct<DUMMY, AC_BUFF_ARCH>::BUFF_TYPE
BUFF_TYPE;
  BUFF_TYPE lb;
};
```

The AC_BUFF_ARCH template parameter is used to pick between different linebuffer architectures. In C++, the different architectures are coded as separate classes. More details on the AC_BUFF_ARCH template parameter, as well as the classes/architectures it picks, are given in Template Parameters.

Template Parameters

- INPUT_TYPE – Specifies the input type for the linebuffer. Like the other linebuffers, ac_linebuffer_flush only supports ac_packed_vector objects on the input.

- OUTPUT_TYPE – Specifies the output type for the linebuffer. The types supported are:
  (a) A 2D array, i.e. ac_array<BASE_TYPE, AC_NUM_LINES, AC_BUS_WORDS>, where BASE_TYPE is the base type of the input packed vector and AC_BUS_WORDS is the number of packed words.
  (b) An ac_array with packed vector elements, i.e. ac_array<INPUT_TYPE, AC_NUM_LINES>.
  All the linebuffer sub-classes produce an output of the type specified in (b), i.e. an ac_array with packed vector elements. If the OUTPUT_TYPE is also (b), no type conversion or unpacking is required, but if it is (a), the packed vector elements need to be unpacked before they're converted to OUTPUT_TYPE. The linebuffer_access() methods specified in Class Methods handle this conversion.

- AC_WIDTH – Width of the linebuffer after unpacking. This value must be divisible by the number of words packed in parallel at the input, i.e. INPUT_TYPE::packed_words/AC_BUS_WORDS.

- AC_NUM_LINES – Number of rows/lines in the output window. For standard 2D convolution, the number of linebuffers is one less than AC_NUM_LINES.

- AC_BUFF_ARCH – Linebuffer architecture. This is an enum of type ac_buff_arch, as specified below:

```
enum ac_buff_arch_flush {
  AC_1R1W_FLUSH,
  AC_SPWRMASK_FLUSH,
  AC_SPCIR_FLUSH };
```

  Specifying AC_1R1W_FLUSH for this template parameter selects instantiates the linebuffer as an ac_linebuffer_1r1w_flush object, which results in the linebuffers being synthesized as 1R1W memories. AC_SPWRMASK_FLUSH instantiates the linebuffer as an ac_linebuffer_spwrmask_flush object, which results in the linebuffers being synthesized as one singleport memory with write-masking. AC_SPCIR_FLUSH instantiates the linebuffer as an ac_linebuffer_spcircular_flush, which results in the linebuffers being synthesized as multiple singleport memories. The design switches between memories as though they were elements of a circular buffer.

  If we're using singleport linebuffers, AC_WIDTH must be divisible by (2*AC_BUS_WORDS).

- AC_REPEAT – set to true if the design is expected to iterate over the same set of lines. This feature is under development for flush support and setting the parameter to true will trigger a static assertion. For the default case, i.e. standard 2D convolution, AC_REPEAT is set to false.

Class Methods

Method	Description

`void run(`
`  INPUT_TYPE din,`
`  ADDR_TYPE write_addr,`
`  ADDR_TYPE read_addr,`
`  bool write_enabled,`
	Writes data inputs to the linebuffers and reads data outputs from it. din is the input data and dout is the output data. The write_addr and read_addr inputs specify the write and read addresses of the linebuffers, respectively. write_enabled is used to specify whether writes to the linebuffers are enabled or not. More details on this

`OUTPUT_TYPE &dout,` `bool repeat_line = false)`	input, as well as the read_addr and write_addr inputs, are given in Controlling Writes to the Linebuffer. repeat_line is set to true when the design is iterating over a previous set of lines. A description of all the changes this entails is given in Linebuffer Reads and Writes During Line Repetition.
`void reset()`	Sets all elements of the linebuffers to zero.

`void linebuffer_access(`
`ac_array<INPUT_TYPE,`
`AC_NUM_LINES> packed_input,`
`ac_array<BASE_TYPE,`
`AC_NUM_LINES,`
`AC_BUS_WORDS> &output)`
	This method uses two unrolled loops to convert the linebuffer output to a 2D ac_array. It is called by the run() function to handle output conversion when the OUTPUT_TYPE is specified as ac_array<BASE_TYPE, AC_NUM_LINES, AC_BUS_WORDS> by the user.

`void linebuffer_access(`
`ac_array<INPUT_TYPE,`
`AC_NUM_LINES> packed_input,`
`ac_array<INPUT_TYPE,`
`AC_NUM_LINES> &output)`
	This method is used to convert the linebuffer output to an ac_array with packed_vector elements. It is called by the run() function when the OUTPUT_TYPE is specified as ac_array<INPUT_TYPE, AC_NUM_LINES> by the user.

`void align_buf_cnts()`	This method aligns the internal counter which is responsible for reading from the linebuffer in the correct order. It should be called when the window shifts from the flushing state to the idle state to account for any misalignments which might have happened during the former.
`bool misaligned()`	Single-port memory linebuffers require reads to occur on even cycles and writes to occur on odd cycles. Certain patterns of input stalling during the flushing state can result in misalignments in this order of reads and writes. When transitioning from the flushing period to the idle state, the design might need to stall for an extra cycle so that the misalignment is resolved. The misaligned() function returns true to signal this need to stall. For more details of how this method is used, refer to the section on the Top Level Design for Flush Support (ac_window_v2_flush).

4.5.2. Controlling Writes to the Linebuffer

Since the AC Window 2.0 libraries with flush support are designed to operate with non-blocking interfaces, we must account for the chance that they may not accept new input data while sliding the window. This is true in the following cases:

- If the design is in the flushing state but no new input is available.
- If the design is using singleport memories and it must resolve misalignments at the beginning of the idle state.

In these cases, the user must set the write_enabled input of the run() function to false. These cases will also result in a difference between the write and read addresses of the linebuffers. Hence, the run() function lets the user specify the read and write address separately through the read_addr and write_addr inputs.

Note that if no new input is available during the active or idle state (barring cases where misalignments must be resolved), the run() function must not be called, so as to stop the internal pointers in the linebuffer class from advancing. This can be done by gating the call to the run() function with a conditional statement.

More details on the different window states (e.g. idle and flushing states) is given in Window States for Flush Support.

4.5.3. Linebuffer Reads and Writes During Line Repetition

The linebuffer supports line repetition, i.e. iterating across the same set of lines, for applications such as interpolation. To enable line repetition, instantiate the linebuffer class template with AC_REPEAT = true and set the repeat_line input to the run function to true as well. repeat_line can only be true if AC_REPEAT is also true, failing which a runtime software assertion will be triggered.

During line repetition, the buffer counts (read_buf_cnt and write_buf_cnt) do not change, to make sure that the design can get back to reading from and writing to the correct buffers once we're not repeating lines any more. Additionally, no inputs are written to the linebuffer. The design now reads from all the buffers, as it relies exclusively on stored image data to produce the outputs. Setting AC_REPEAT to true also allocates an extra buffer to make sure that the linebuffer block can rely solely on stored data during repetition.

4.6. Synchronization Flag Generator for Flush Support (ac_flag_gen_flush)

Similar to ac_flag_gen, ac_flag_gen_flush also generates synchronization flags to indicate the start of line, end of line, start of frame and end of frame. However, the since the flag generator must support flushing, it must also maintain two sets of flags. One of these sets will handle flag generation for the primary frame, i.e. the frame that is being windowed. The other set will handle flag generation for the secondary frame, i.e. the frame that is being read while outputs from the primary frame are being flushed out.

Once the design transitions from the flushing state to the idle state for the next frame, the secondary frame becomes the primary frame, and the set of flags associated with the secondary frame are now associated with the primary frame. Similarly, the set of flags associated with the primary frame will be associated with the secondary frame during the next flushing period. This alternation is handled by the top-level window design using a ping-pong scheme. To support this scheme, the synchronization flag generator must take into account which state the window is in and output two sets of flags—the “ping” set and the “pong” set. ac_flag_gen_flush provides this functionality and the interface needed to implement it.

This library has two classes ac_flag_gen_flush_1d and ac_flag_gen_flush_2d, which generate synchronization flags in one and two dimensions, respectively.

4.6.1. C++ Code Overview

A snippet of the ac_flag_gen_flush_1d class is given below.

```
template<int AC_WIDTH, int AC_BUS_WORDS = 1>
class ac_flag_gen_flush_1d
{
public:
```

```
  static_assert(AC_WIDTH > 0, "AC_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");
  static_assert(AC_WIDTH%AC_BUS_WORDS == 0, "AC_WIDTH must be perfectly divisi-
ble by AC_BUS_WORDS.");

  typedef ac_int<ac::nbits<AC_WIDTH>::val, false> W_TYPE;

  // Class methods

private:
  ac_int<ac::nbits<AC_WIDTH - AC_BUS_WORDS>::val, false> cnt_ping, cnt_pong;
};
```

Since the design generates two sets of flags, it must also have counters cnt_ping and cnt_pong, one corresponding to each set of flags. The counters help in tracking the horizontal position of pixels from the primary and secondary frames.

A snippet of the ac_flag_gen_flush_2d class is given below.

```
template<int AC_WIDTH, int AC_HEIGHT, int AC_BUS_WORDS = 1>
class ac_flag_gen_flush_2d
{
public:
  static_assert(AC_WIDTH > 0, "AC_WIDTH must be positive.");
  static_assert(AC_HEIGHT > 0, "AC_HEIGHT must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  typedef ac_int<ac::nbits<AC_WIDTH>::val, false> W_TYPE;
  typedef ac_int<ac::nbits<AC_HEIGHT>::val, false> H_TYPE;

  // Class methods

private:
  ac_flag_gen_flush_1d<AC_WIDTH, AC_BUS_WORDS> line;
  ac_int<ac::nbits<AC_HEIGHT - 1>::val, false> cnt_ping, cnt_pong;
  bool incorrect_dim_change;
  W_TYPE frame_width;
  H_TYPE frame_height;
};
```

Similar to the ac_flag_gen_2d class, ac_flag_gen_flush_2d also uses an object of the 1D class to compute the SOL and EOL flags. Like ac_flag_gen_flush_1d, this class also has two counters cnt_ping and cnt_pong for each set of flags. These counters help in tracking the vertical position of pixels from the primary and secondary frames. The frame_width and frame_height variables are updated to store the width and height dimensions when an SOF pixel is received in the idle state. Like the frame_width and frame_height variables in ac_flag_gen, these variables also help in detecting any unexpected changes in dimension inputs. More

details are given in Incorrect Dimension Checking. W_TYPE and H_TYPE are the types for the width and height input to the run() function, respectively.

ac_flag_gen_flush_1d Template Parameters

- AC_WIDTH – Specifies the maximum line width.
- AC_BUS_WORDS – Number of pixels that are sent in parallel over the bus interface. It is set to 1 by default.

Both these template parameters must be positive. AC_WIDTH must also be perfectly divisible by AC_BUS_WORDS. If any of these conditions are violated, a static assertion is triggered.

ac_flag_gen_flush_2d Template Parameters

- AC_WIDTH – Specifies the maximum frame width.
- AC_HEIGHT – Specifies the maximum frame height.
- AC_BUS_WORDS – Number of pixels that are sent in parallel over the bus interface. It is set to 1 by default.

Similar to the ac_flag_gen_flush_1d Template Parameters, these parameters must also be positive and AC_WIDTH must also be perfectly divisible by AC_BUS_WORDS, with static assertions being added to guard against any violation of these conditions.

Constructors

Constructor	Description
ac_flag_gen_flush_1d()	Default ac_flag_gen_flush_1d constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_gen_1d Methods.
ac_flag_gen_flush_2d()	Default ac_flag_gen_flush_2d constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_gen_flush_2d Methods.

Miscellaneous ac_flag_gen_flush_1d Methods

Method	Description

void run (
  W_TYPE width,
  ac_window_v2_flush_state window_state,
  bool ping,
  bool flush_write_enabled,
  bool &sol_ping, bool &eol_ping,
  bool &sol_pong, bool &eol_pong)
	Class interface run function. This should be called every time the design receives a pixel. The width dimension input specifies the width of the line. The window_state input specifies the state the window is in. For more detail on window states, refer to Window States for Flush Support. The ping input variable is alternatively set to true and false to switch between ping and pong sets of flags using the ping-pong scheme described earlier. The flush_write_enabled input is set to true if the design processes new inputs during the flushing state. sol_ping and eol_ping are the “ping” set of SOL and EOL flags, while sol_pong and eol_pong are the “pong” set.

void reset_primary_cnt(bool ping)	This function must be called just before a 2D window design switches from the idle state to the flush state, so that the counter associated with

	primary frame is reset to zero. This prepares the window for the next flush state. The ping input selects which the primary registers are, and hence figure out which registers to reset.
void reset()	Resets cnt_ping and cnt_pong to zero.

Miscellaneous ac_flag_gen_flush_2d Methods

Method	**Description**

void run (
  W_TYPE width, H_TYPE height,
  ac_window_v2_flush_state window_state,
  bool ping, bool flush_write_enabled,
  bool &sof_ping, bool &eof_ping, bool &sol_ping, bool &eol_ping,
  bool &sof_pong, bool &eof_pong, bool &sol_pong, bool &eol_pong,
  bool only_update_line = false)
	Class interface run function. This should be called every time the design receives a valid input pixel and/or produces an output window. The width and height dimension inputs specify the width and height of the frame, respectively. The width input must be perfectly divisible by AC_BUS_WORDS, failing which an AC_ASSERT is triggered. The window_state input specifies the state the window is in. For more detail on window states, refer to Window States for Flush Support. The ping input variable is alternatively set to true and false to switch between ping and pong sets of flags using the ping-pong scheme described earlier.
	Similar to the 1D flag generator, the 2D flag generator's run() function also outputs a "*_ping" and a "*_pong" set of flags for the SOF, EOF, SOL and EOL flags.
	only_update_line should be set to true during line repetition. This makes sure that the SOF and EOF flags and the associated counters aren't updated but the SOL and EOL flags as well as the associated counters are updated.

void reset_primary_line_cnt(bool ping)	This method calls the reset_primary_cnt() method of the 1D flag generator that was declared as a member of the class. For more details on this method, refer to Miscellaneous ac_flag_gen_1d Methods. It is called by the switch_frames() method described in Miscellaneous ac_window_v2_flush_2d Methods.
void reset()	Does the following: - Calls the reset() function of the line member variable. - Resets cnt_ping and cnt_pong to 0. - Resets the incorrect_dim_change flag to false. - Resets frame_width and frame_height to 0.
bool dim_change_error()	Returns the incorrect_dim_change member variable. More details on this variable are given in Incorrect Dimension Checking.

4.6.2. Incorrect Dimension Checking

Similar to ac_flag_gen_2d, ac_flag_gen_flush_2d also provides functionality to check for incorrect changes to the dimension inputs, i.e. width and height. Unlike in ac_flag_gen, however, the dimension inputs can only change if the next frame is received in the idle state. If the design processes the next frame while flushing outputs from the primary frame, the dimension of the next frame must be the same as the primary frame. If the next frame is received in the idle state, however, it means that the SOF pixel will be associated with the primary frame. The dimensions can only change when the aforementioned SOF pixel is received. The following snippet highlights how the design checks for this condition:

```
    // sof_primary is true only when SOF pixel is received in the idle state.
    if (sof_primary) {
      frame_width = width;
      frame_height = height;
    } else {
      incorrect_dim_change = (width != frame_width) || (height !=
frame_height);
      #ifndef DISABLE_AC_FLAG_GEN_FLUSH_ASSERT
      AC_ASSERT(!incorrect_dim_change, "Dimension input(s) changed unexpect-
edly. Make sure the dimensions only change when an SOF pixel is received in the
idle state.");
      #endif
    }
```

When sof_primary is received, the width and height for the primary frame are stored in the frame_width and frame_height class member variables. For every other iteration, the width and height dimension inputs will be compared against frame_width and frame_height, respectively. If either of these comparisons fails, the incorrect_dim_change member variable will be set to true. By default, this will trigger an AC_ASSERT. This assert can be disabled by defining the DISABLE_AC_FLAG_GEN_FLUSH_ASSERT macro, similar to the corresponding assert in ac_flag_gen.

It is important to remember that the assert is only available in software. The user can use the dim_change_error() method described in Miscellaneous ac_flag_gen_flush_2d Methods to return the incorrect_dim_change flag and carry out error checking in hardware with that.

4.7. Flag Shifter for Flush Support (ac_flag_shift_flush)

As described earlier, flushing support requires the window to operate on two sets of flags. As covered in ac_window_1d_flag Class, we need a flag shifter to align input flags with the shifted data. Since we're using two sets of flags instead of one, there are special considerations to be taken while shifting flags for flush support. Note that these do not apply for data shifting. As a result, the flushing support window can use the same data shifter as designs where flushing support is not incorporated, i.e. ac_shift_N.

4.7.1. C++ Code Overview for Flag Shifters

A snippet of the ac_flag_shift_flush_1d class is given below.

```
template<int AC_WIN_WIDTH, ac_padding_method AC_PMODE, int AC_BUS_WORDS = 1>
class ac_flag_shift_flush_1d
{
public:
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  // Other code

private:
```

```
  ac_int<AC_SHIFT_WORDS, false> sol_regs_ping, eol_regs_ping, sol_regs_pong,
eol_regs_pong;
};
```

Like the ac_flag_shift_1d class, this class also has registers for shifting SOL and EOL flags. These registers are duplicated for the ping and pong storage, i.e. sol_regs_ping, eol_regs_ping, sol_regs_pong and eol_regs_pong. AC_SHIFT_WORDS is computed the same way as described in Calculating Shifter Size. A snippet of the ac_flag_shift_flush_2d class is given below.

```
template <int AC_WIN_HEIGHT, int AC_WIN_WIDTH, ac_padding_method AC_PMODE, int
AC_BUS_WORDS = 1, bool AC_REPEAT = false>
class ac_flag_shift_flush_2d
{
public:
  static_assert(AC_WIN_HEIGHT > 0, "AC_WIN_HEIGHT must be positive.");
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  typedef ac_flag_shift_flush_1d<AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS>
LINE_TYPE;

  enum {
    USING_REFLECT101 = AC_PMODE == AC_REFLECT101,
    EXTRA_LINE = USING_REFLECT101 ? int(AC_WIN_HEIGHT%2 == 0) : 0,
    AC_SHIFT_HEIGHT = AC_WIN_HEIGHT + EXTRA_LINE,
  };

  // Other code.

private:
  LINE_TYPE line;
  ac_array<bool, AC_SHIFT_HEIGHT, NUM_SHIFTS_FILL + EXTRA_WORD> sof_regs_ping,
eof_regs_ping, sof_regs_pong, eof_regs_pong;
  bool eof_old_ping, eol_old_ping, eof_old_pong, eol_old_pong;
};
```

The ac_flag_shift_flush_2d class is built on top of the ac_flag_shift_flush_1d class and uses line, an ac_flag_shift_flush_1d object serving to store and update the horizontal flag registers. Like the ac_flag_shift_2d class, this class also has registers used to shift SOF and EOF flags. These registers are also duplicated for the ping and pong set of flags, i.e. sof_regs_ping, eof_regs_ping, sof_regs_pong and eof_regs_pong. AC_SHIFT_HEIGHT is computed in the manner described in C++ Code Overview for Data Shifters. Like ac_flag_shift_2d, this class also has variables to let us delay shifting the eof_regs* registers by one iteration. These variables are also duplicated for ping and pong flags, i.e. eof_old_ping, eol_old_ping, eof_old_pong and eol_old_pong.

ac_flag_shift_flush_1d Template Parameters

These are the same as those described in ac_flag_shift_1d Template Parameters.

ac_flag_shift_flush_2d Template Parameters

These are the same as those described in ac_flag_shift_2d Template Parameters.

Constructors

Constructor	Description
ac_flag_shift_flush_1d()	Default ac_flag_shift_flush_1d constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_shift_flush_1d Methods.
ac_flag_shift_flush_2d()	Default ac_flag_shift_flush_2d constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_shift_flush_2d Methods.

Miscellaneous ac_flag_shift_flush_1d Methods

Method	Description
void run(bool sol_ping, bool eol_ping, bool sol_pong, bool eol_pong, ac_window_v2_flush_state window_state, bool ping, bool flush_write_enabled, ac_int<AC_WORDS, false> &sol_flags_ping, ac_int<AC_WORDS, false> &eol_flags_ping, ac_int<AC_WORDS, false> &sol_flags_pong, ac_int<AC_WORDS, false> &eol_flags_pong)	Carries out right-shifting and updates the internal SOL and EOL flag registers. The content of these registers is written to sol_flags_ping, eol_flags_ping, sol_flags_pong and eol_flags_pong. sol_ping, eol_ping, sol_pong and eol_pong are the outputs from the ac_flag_gen_flush block. The window_state input specifies the state the window is in. For more detail on window states, refer to Window States for Flush Support. The ping input variable is alternatively set to true and false to switch between ping and pong sets of flags using the ping-pong scheme described earlier. The flush_write_enabled input is set to true if the design processes new inputs during the flushing state.
void reset_primary_regs(bool ping)	Reset SOL and EOL registers associated with the primary frame. The ping input selects which the primary registers are, and hence figure out which registers to reset.
void reset()	Resets all internal registers of the class.

Miscellaneous ac_flag_shift_flush_2d Methods

Method	Description
void run(bool sof_ping, bool eof_ping, bool sol_ping, bool eol_ping, bool sof_pong, bool eof_pong, bool sol_pong, bool eol_pong, ac_window_v2_flush_state window_state, bool ping, bool flush_write_enabled, ac_int<AC_WIN_HEIGHT, false> &sof_flags_ping,	Carries out right-shifting and updates the internal registers. The contents of the registers are written to sol_flags_ping, eol_flags_ping, sof_flags_ping, eof_flags_ping, sol_flags_pong, eol_flags_pong, sof_flags_pong and eof_flags_pong. The sof_ping, eof_ping, sol_ping, eol_ping, sof_pong, eof_pong, sol_pong and eol_pong inputs are outputs from ac_flag_gen_flush_2d.

ac_int<AC_WIN_HEIGHT, false> &eof_flags_ping,
  ac_int<AC_WORDS, false> &sol_flags_ping,
ac_int<AC_WORDS, false> &eol_flags_ping,
  ac_int<AC_WIN_HEIGHT, false> &sof_flags_pong,
ac_int<AC_WIN_HEIGHT, false> &eof_flags_pong,
  ac_int<AC_WORDS, false> &sol_flags_pong,
ac_int<AC_WORDS, false> &eol_flags_pong,
  bool repeat_line = false)
	The window_state input specifies the state the window is in. For more detail on window states, refer to Window States for Flush Support. The ping input variable is alternatively set to true and false to switch between ping and pong sets of flags using the ping-pong scheme described earlier. The flush_write_enabled input is set to true if the design processes new inputs during the flushing state.
	
	repeat_line is intended to be used for applications such as interpolation, where line repetition is required. This ensures that no outputs are read to the last column of the internal SOF/EOF registers. repeat_line can only be set to true if AC_REPEAT == true, failing which a runtime software assertion will be triggered.

void reset_primary_line_regs(bool ping)	Calls the reset_primary_regs function described in Miscellaneous ac_flag_shift_flush_1d Methods to reset the primary line flag registers. This function must be called just before a 2D window design switches from the idle state to the flush state, so that the window is prepared for the next flush state.
void reset()	Resets all internal registers and data members used to store the old EOL and EOF flags.

4.8. Output Flag Generator for Flush Support (ac_flag_gen_out_flush)

Windowed designs need a delay to ramp up. As covered in Output Flag Generator for Standard Windowing (ac_flag_gen_out), the output flag generator must take this delay into account to produce the output flags. In addition—because the window for flushing support operates with non-blocking inputs—the corresponding output flag generator must account for invalid inputs.

4.8.1. C++ Code Overview

A snippet of the ac_flag_gen_out_flush_1d class is given below.

```
template<int AC_WIN_WIDTH, int AC_BUS_WORDS = 1>
class ac_flag_gen_out_flush_1d
{
public:
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  // Class methods
};
```

A snippet of the ac_flag_gen_out_flush_2d class is given below.

```
template<int AC_WIN_HEIGHT, int AC_WIN_WIDTH, int AC_BUS_WORDS = 1>
class ac_flag_gen_out_flush_2d
{
public:
  static_assert(AC_WIN_HEIGHT > 0, "AC_WIN_HEIGHT must be positive.");
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");
  static_assert(AC_WIN_HEIGHT >= 3, "AC_WIN_HEIGHT must not be less than 3.");

  // Class methods
};
```

The template parameters for the 1D and 2D output flag generators are the same as those mentioned in ac_flag_gen_out_1d Template Parameters and ac_flag_gen_out_2d Template Parameters, respectively.

4.8.2. Class Methods

Unlike the ac_flag_gen_out classes, the ac_flag_gen_out_flush classes do not have member variables, including a variable that stores information about whether the window has ramped up or not. As a result, there are no member variables to initialize/reset. The classes do not have any reset methods, and the default constructors included in both are empty. Information about the state of the window is passed through an input variable to the run() functions, which are described below.

A snippet of the run() function for ac_flag_gen_out_flush_1d is given below.

```
void run (
  ac_int<AC_WORDS, false> sol_flags, ac_int<AC_WORDS, false> eol_flags,
  ac_window_v2_flush_state window_state, bool write_enabled,
  bool &sol_out, bool &eol_out, bool &vld_out)
```

Since the output flag generator is concerned with the output window, we only need the primary set of shifted flags, which must be selected beforehand in accordance with ping-pong scheme, i.e. the sol_flags and eol_flags inputs. The input variable window_state is the one that conveys information about the state of the window. For more detail on window states, refer to Window States for Flush Support. The write_enabled input specifies whether the input is valid and whether it should be written to the internal storage of the window design. In context of the output flag generator, the write_enabled input is used to generate the vld_out flag. More details on the same are given in vld_out Flag for Flushing Support. The sol_out and eol_out outputs give information about the horizontal position of the overlapped windows as specified in Output Flags with Padding.

A snippet of the run() function for ac_flag_gen_out_flush_2d is given below.

```
void run (
  ac_int<AC_WIN_HEIGHT, false> sof_flags, ac_int<AC_WIN_HEIGHT, false>
eof_flags,
  ac_int<AC_WORDS, false> sol_flags, ac_int<AC_WORDS, false> eol_flags,
```

```
  ac_window_v2_flush_state window_state, bool write_enabled,
  bool &sof_out, bool &eof_out, bool &sol_out, bool &eol_out, bool &vld_out)
```

Similar to the 1D output flag generator, the 2D version also operates on the shifted flags for the primary frame, i.e. sof_flags, eof_flags, sol_flags and eol_flags. An object of ac_flag_gen_out_flush_1d is declared as a local variable of the run() function and computes the sol_out and eol_out outputs. The sof_out and eof_out outputs give information on the vertical position of the overlapped windows as specified in Output Flags with Padding. The window_state and write_enabled inputs serve the same function as they do for the 1D output flag generator.

4.8.3. vld_out Flag for Flushing Support

Similar to the ac_flag_gen_out classes, the vld_out flag for the ac_flag_gen_out_flush classes is set to true when the design has ramped up and the output window is valid. For flushing support, this output depends on the state the window is in:

- If a 2D window is in the idle state, the output flag generator monitors the sof_out flag. Once sof_out is true, i.e. once the center pixel of the first individual window is the SOF pixel, vld_out is set to true. This denotes a change from the idle to active state. When such a transition happens, the vld_out flag is used externally to change the state of the window from idle to active. For more details, refer to Window States for Flush Support. For a 1D Window, this transition depends on the sol_out flag instead of sof_out.

- If the window is in the active state, vld_out is set to true whenever the input is valid, i.e. whenever write_enabled is true. This is because the presence of valid data indicates that the windows has slid to a new position and that a new valid output window is available.

- The flushing states of the window necessitate that output data is always flushed, regardless of whether the input data is valid or not. As a result, a new valid output window is always available when the window is flushing, and the vld_out flag is always set to true in such a case.

4.9. Top Level Design for Flush Support (ac_window_v2_flush)

Similar to the AC Window 2.0 Design without Flush Support, the top-level design with flush support also instantiates all the sub-blocks and brings them together for windowing. This design is also meant to be pipelined with and II of 1 and capable of producing multiple output windows at the same time.

Note that only two-dimensional windows are supported. More details on this and other limitations of the flushing support windows are given in Limitations.

4.9.1. C++ Code Overview

A snippet of the ac_window_v2_flush_2d class is given below.

```
template <class PIX_TYPE, int AC_IMG_HEIGHT, int AC_IMG_WIDTH, int
AC_WIN_HEIGHT, int AC_WIN_WIDTH, ac_buff_arch_flush AC_BUFF_ARCH,
```

```
ac_padding_method AC_PMODE, int AC_BUS_WORDS = 1, bool HIGH_SPEED_REPLICATE =
false, bool AC_REPEAT = false>
class ac_window_v2_flush_2d
{
public:
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_WIN_HEIGHT > 0, "AC_WIN_HEIGHT must be positive.");
  static_assert(AC_WIN_HEIGHT >= 3, "AC_WIN_HEIGHT must not be less than 3.");

  typedef ac_boundary_2d<PIX_TYPE, AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_PMODE,
AC_BUS_WORDS, HIGH_SPEED_REPLICATE> BOUND_TYPE;
  typedef ac_int<ac::nbits<AC_IMG_WIDTH>::val, false> W_TYPE;
  typedef ac_int<ac::nbits<AC_IMG_HEIGHT>::val, false> H_TYPE;

  enum { AC_SHIFT_HEIGHT = BOUND_TYPE::AC_SHIFT_HEIGHT };

  typedef ac_packed_vector<PIX_TYPE, AC_BUS_WORDS> LB_IN_TYPE;
  typedef ac_array<PIX_TYPE, AC_SHIFT_HEIGHT, AC_BUS_WORDS> LB_OUT_TYPE;
  typedef ac_linebuffer_flush<LB_IN_TYPE, LB_OUT_TYPE, AC_IMG_WIDTH,
AC_SHIFT_HEIGHT, AC_BUFF_ARCH, AC_REPEAT> LB_TYPE;

  // Other code.

private:
  ac_flag_gen_flush_2d<AC_IMG_WIDTH, AC_IMG_HEIGHT, AC_BUS_WORDS> fgen;
  LB_TYPE linebuf;
  ac_flag_shift_flush_2d<AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS,
AC_REPEAT> fshift;
  ac_shift_N_2d<PIX_TYPE, AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS>
dshift;

  ac_window_v2_flush_state window_state;
  bool ping, dont_read_data_;
  typename LB_TYPE::ADDR_TYPE linebuf_read_addr, linebuf_write_addr;
  PIX_TYPE padded_val;
};
```

The following sub-blocks are instantiated as members of the class:

- 2D Synchronization Flag Generator for Flush Support – Instantiated as fgen.
- Linebuffer for Flush Support – Instantiated as linebuf.
- 2D Flag Shifter for Flush Support – Instantiated as fshift.
- 2D Data Shifter – Instantiated as dshift.

Since the 2D Boundary Processor and the Output Flag Generator for Flush Support do not need any static storage, they are instantiated as variables local to the run() function described in Miscellaneous ac_window_v2_flush_2d Methods.

The linebuffer is instantiated using the LB_IN_TYPE and LB_OUT_TYPE typedefs to specify the input and output to the linebuffer. W_TYPE and H_TYPE are the types used for the width and height inputs to the run() function, respectively.

window_state is an enumeration from the ac_window_v2_flush_state enumeration list which specifies the state the window is in. It is passed as an input to all the sub-blocks associated with flag handling, i.e. the ac_flag_gen_flush, ac_flag_shift_flush and ac_flag_gen_out_flush sub-blocks. More details on the window states are given in Window States for Flush Support. The ping boolean member helps alternate between two sets of flags using a ping-pong scheme. It is also passed as an input to all the flag handling sub-blocks. The window_state and ping variables hence allow the flag handling sub-blocks to correctly associate each set of flags with the primary and secondary frames.

dont_read_data_ is a member variable used to stall reads to the window to avoid misalignments while using a singleport memory. It is set to the value returned by the misaligned() class method in ac_linebuffer_flush, during the a transition out of either of the flushing states for the window, i.e. AC_WINDOW_FLUSH or AC_WINDOW_FLUSH_LB_ADVANCE. If it is set to true, the window will not read the input in the next cycle/call to the run() function, even if it is a valid input, and it will instead focus on writing the previous input to the linebuffer at the correct address to resolve any potential misalignments.

linebuf_read_addr and linebuf_write_addr store the read and write addresses to the linebuffer, respectively.

padded_val is initialized during class construction, either by the default or parameterized constructor listed in Constructors. It is the constant value used to pad the window boundary when AC_CONSTANT is the boundary processing mode.

ac_window_v2_flush_2d Template Parameters

- PIX_TYPE — Pixel type. It must contain a parameterized constructor that accepts integer arguments, to support typecasting in the boundary processor. Refer to Base Type Constructor Requirements for more details. It must also be compatible with the ac_packed_vector container class.
- AC_IMG_HEIGHT – Maximum possible image height.
- AC_IMG_WIDTH – Maximum possible image width.
- AC_WIN_HEIGHT – Window height, i.e. number of rows in the window.
- AC_WIN_WIDTH – Window width, i.e. number of columns in the window.
- AC_BUFF_ARCH – Specifies the linebuffer architecture used.
- AC_PMODE – Boundary processing mode. For more details, refer to Boundary Processing Modes.
- AC_BUS_WORDS – Number of pixels packed in parallel on the data bus.
- HIGH_SPEED_REPLICATE – Refer to ac_boundary_1d Template Parameters for more details.

- AC_REPEAT – set to true if the design is expected to iterate over the same set of lines. This feature is under development for flush support and setting the parameter to true will trigger a static assertion. For the default case, i.e. standard 2D convolution, AC_REPEAT is set to false.

Constructors

Constructor	Description
ac_window_v2_flush_2d()	Default ac_window_v2_flush_2d() constructor. Calls the reset() function mentioned in Miscellaneous ac_window_v2_flush_2d Methods and also sets the padded_val value used by the boundary processor to 0.
ac_window_v2_flush_2d(PIX_TYPE pval)	Parameterized ac_window_v2_flush_2d() constructor. Calls the reset() function mentioned in Miscellaneous ac_window_v2_flush_2d Methods and also sets the padded_val value used by the boundary processor to pval.

Miscellaneous ac_window_v2_flush_2d Methods

The following table describes other ac_window_v2_flush_2d methods:

Constructor	Description
void run (ac_array<PIX_TYPE, AC_BUS_WORDS> din, bool write, W_TYPE width, H_TYPE height, ac_array<PIX_TYPE, AC_WIN_HEIGHT, AC_WORDS> &dout, bool &sof_out, bool &eof_out, bool &sol_out, bool &eol_out, bool &vld_out, bool &dont_read_data, bool repeat_line = false)	Performs 2D Windowing. din is the input pixel data. write specifies whether the input is valid or not and hence specifies whether it should be written to the internal storage of the window. The width and height dimension inputs specify the width and height of the image, respectively. dout is the overlapped output window from the boundary processor. sof_out, eof_out, sol_out and eol_out and vld_out are the outputs of the output flag generator. dont_read_data is set to true by the window sub-block if the window has to stall for a cycle to resolve alignment issues. In such a case, the window will not read any input data, even if it is valid. repeat_line is set to true if the user desires line repetition. Refer to Line Repetition for more details.
bool switch_frames()	This is called by the run() function whenever the window switches from a flushing state to the idle state. It does the following: - Sets the window_state flag to AC_WINDOW_IDLE. - Calls the reset_primary_line_cnt() and reset_primary_line_regs() methods of the flag generator and shifter, respectively. - Toggles the ping flag. - Sets dont_read_data to the value returned by the misaligned() method of ac_linebuffer_flush. The net effect is that the ping and pong set of flags alternate, and the flags associated with the secondary window will now be associated with the primary window.

	The return value is the dont_read_data flag, which is useful for line repetition as this method must be explicitly called by the user at the start of the idle state. More details are given in Line Repetition.
void reset()	Does the following: - Calls the reset() methods of the flag generator, flag shifter and data shifter. - Sets window_state to AC_WINDOW_IDLE. - Sets linebuf_read_addr and linebuf_write_addr to 0. - Sets ping to true. - Sets dont_read_data_ to false.

4.9.2. Block Diagram

Refer to Illustration 54 for a block diagram showing how the sub-blocks are integrated in the top-level window design for flushing support.

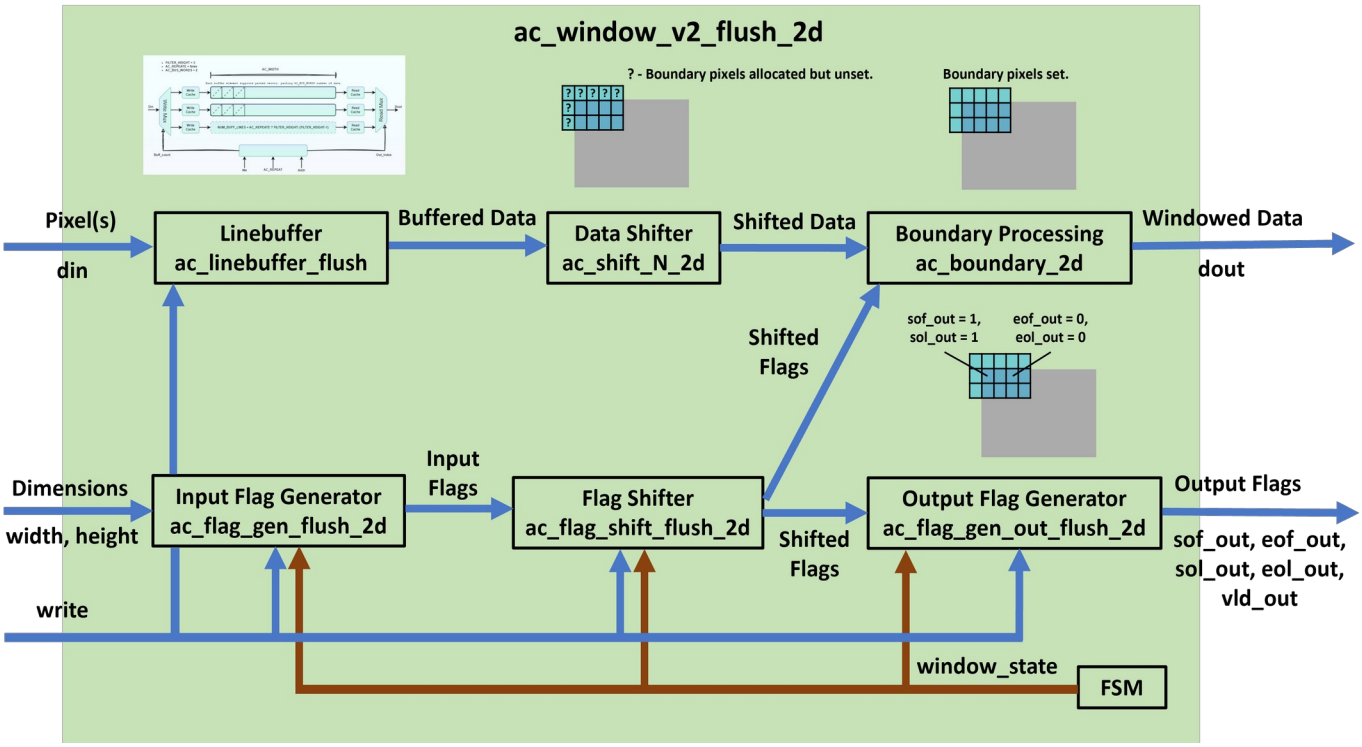

Illustration 54: Block Diagram for Flushing Support

The block diagram is largely the same as that for Standard Windowing, but there are two key differences:

- All the sub-blocks coded specifically for flushing support accept the write input, as it allows them to process inputs in the flushing region appropriately.

- All the flag handling sub-blocks are given the window_state variable from the Finite State Machine (FSM) as an input, because they need it to process the flags appropriately. More details on the window_state variable are given in Window States for Flush Support.

4.9.3. Window States for Flush Support

As mentioned earlier, the window_state variable is responsible for communicating the state information to all the flag handling sub-blocks in the design. It is declared as a data member of the window class and is of type ac_window_v2_flush_state, an enumeration declared in the header file ac_window_flush_enums.h:

```
enum ac_window_v2_flush_state {
  AC_WINDOW_IDLE,
  AC_WINDOW_ACTIVE,
  AC_WINDOW_FLUSH,
  AC_WINDOW_FLUSH_LB_ADVANCE
};
```

The significance of each of the enumerators and the state they represent is given below:

- AC_WINDOW_IDLE – Idle state enumerator. In this state, the window's generally busy ramping up with primary frame data (if valid) and doesn't produce outputs. The only exception is at the last cycle of the idle state, when sof_out is set to true by the output flag and the rampup is complete, as mentioned in vld_out Flag for Flushing Support.

- AC_WINDOW_ACTIVE – Active state enumerator. In this state, the window simultaneously consumes data inputs (if valid) from the primary frame and carries out windowing on it.

- AC_WINDOW_FLUSH – Flush state enumerator. In this state, the window flushes outputs from the primary frame. If valid input data from the secondary frame is available, it will also ramp the window up with it.

- AC_WINDOW_FLUSH_LB_ADVANCE – Enumerator for flushing with linebuffer advance. "Linebuffer Advance" in this case means that the linebuffer now advances and outputs data from the secondary frame instead of the primary frame. However, due to how EOL boundary processing works, the final windowed data will be from the current/primary frame, not the secondary frame. As a result, this still counts as a flushing state. This state is encountered if and when the window has ramped up with (AC_WIN_HEIGHT/2) rows of pixels from the secondary frame during AC_WINDOW_FLUSH, and there are still outputs left to be flushed.

Taking these states into account, we can define the primary and secondary frames as follows:

- The primary frame is the frame being processed at the input during AC_WINDOW_IDLE, the output during AC_WINDOW_FLUSH or AC_WINDOW_FLUSH_LB_ADVANCE and the input/output during AC_WINDOW_ACTIVE.

- The secondary frame is the next frame being written to the internal storage (i.e. linebuffers and shift register) of the window design while the window outputs data from the primary frame in the flushing states, i.e. AC_WINDOW_FLUSH and AC_WINDOW_FLUSH_LB_ADVANCE.

If AC_REPEAT == false, the secondary frame turns into the primary frame and the window will produce outputs from it, once the flushing period has ended. If AC_REPEAT == true, the frame switching must be done explicitly. Refer to Line Repetition for more details. By processing the secondary frame while the window is still flushing outputs from the primary frame, we pre-emptively ramp the window up and reduce the time it spends in the idle state, which in turn increases the throughput of the design.

4.9.4. Line Repetition

Image scaling and interpolation require the window to iterate over the same set of lines when the height of the resized output image is greater than the height of the original image. To accommodate this requirement, the window must not slide vertically while it goes over the repeated lines. To facilitate this, the design does the following:

- The only_update_line input to the input flag generator is set to false. The flag generator does not update the SOF and EOF flags.
- The repeat_line inputs to the linebuffers are set to false. A detailed description of what the linebuffer does is given in Linebuffer Reads and Writes During Line Repetition.

Support for line repetition calls for unique hardware requirements that are not expected when performing standard 2D convolution. Setting the AC_REPEAT template parameter to true will ensure that the appropriate hardware is synthesized. The design must also know when to repeat the same set of lines and when to process new lines at run-time. If the repeat_line input to the run() function is set to true, the design will perform line repetition. If it is set to false, the design will process new lines, as per its default behavior.

If AC_REPEAT == true, it is assumed that the design is resizing the input image. In such a case, the output EOF might not be a reliable indicator of when to switch frames, and the user must explicitly call switch_frames() right before the idle state starts. If using singleport buffers, the user will need to use the return value from the switch_frames() function to determine if the window needs to stall for a cycle to resolve alignment issues at the start of the idle state. This is necessary because the user can't use the dont_read_data output from the run function interface as they normally would to determine if the design needs to stall or not.

Setting repeat_line to true without setting AC_REPEAT to true first is invalid, as it would imply that the user expects the design to perform line repetition despite not having synthesized the appropriate hardware. A software assert (AC_ASSERT) is provided to guard against this condition during C simulation.

4.9.5. Limitations

- Only 2D windows are supported.
- Boundary padding of some form is always enabled. A static assertion is triggered if the user tries to instantiate the window with AC_NO_PADDING as the padding mode.
- If using 2D windows, the pixel type must be compatible with the ac_packed_vector container class. Currently, only the ac_int, ac_fixed and RGB_pv (in ac_pixels.h) datatypes as well as the ac_packed_vector type itself have this compatibility. Support for floating point datatypes might be provided with a future release.

- Windows less than three rows tall are not supported if boundary padding is used, i.e. AC_WIN_HEIGHT must not be less than 3 in such a case.

- The design currently only supports a stride of 1. Strides greater than 1 might be supported in a future release.

4.10. Synchronization Flag Generator for Line Flushing (ac_flag_gen_lflush)

Similar to standard windowing, line flushing also requires the generation of synchronization flags to indicate the start of line, end of line, start of frame and end of frame. Unlike standard windowing, however, the synchronization flags and counters must not be updated during line flushing to ensure correct functioning. To account for this, we use a separate sub-block for flag generation, called ac_flag_gen_lflush, rather than reusing ac_flag_gen.

This library has two classes ac_flag_gen_flush_1d and ac_flag_gen_flush_2d, which generate synchronization flags in one and two dimensions, respectively.

4.10.1. C++ Code Overview

A snippet of the ac_flag_gen_lflush_1d class is given below.

```
template<int AC_WIDTH, int AC_BUS_WORDS = 1>
class ac_flag_gen_lflush_1d
{
public:
  static_assert(AC_WIDTH > 0, "AC_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");
  static_assert(AC_WIDTH%AC_BUS_WORDS == 0, "AC_WIDTH must be perfectly divisi-
ble by AC_BUS_WORDS.");

  typedef ac_int<ac::nbits<AC_WIDTH>::val, false> W_TYPE;

  // Class methods

private:
  ac_int<ac::nbits<AC_WIDTH - AC_BUS_WORDS>::val, false> cnt;
  W_TYPE line_width;
  bool incorrect_dim_change;
};
```

The cnt member variable is responsible for counting the number of pixels and gets reset to zero after the EOL pixel. The line_width variable is updated at the start of every line and is compared with the width dimension input to the run() function for every pixel that isn't an SOL pixel. This is done to check for unexpected changes to the dimension input. If the dimension input does change unexpectedly, the incorrect_dim_change

flag is set to true. More details are given in ac_flag_gen_lflush_1d Checking. W_TYPE is the type for the width input to the run() function.

A snippet of the ac_flag_gen_lflush_2d class is given below.

```
template<int AC_WIDTH, int AC_HEIGHT, int AC_BUS_WORDS=1, bool AC_REPEAT=false>
class ac_flag_gen_lflush_2d
{
public:
  static_assert(AC_WIDTH > 0, "AC_WIDTH must be positive.");
  static_assert(AC_HEIGHT > 0, "AC_HEIGHT must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  typedef ac_int<ac::nbits<AC_WIDTH>::val, false> W_TYPE;
  typedef ac_int<ac::nbits<AC_HEIGHT>::val, false> H_TYPE;

  // Class methods

private:
  ac_flag_gen_lflush_1d<AC_WIDTH, AC_BUS_WORDS> line;
  ac_int<ac::nbits<AC_HEIGHT - 1>::val, false> cnt;
  bool incorrect_dim_change;
  W_TYPE frame_width;
  H_TYPE frame_height;
};
```

Like ac_flag_gen_2d, ac_flag_gen_lflush_2d class is built on top of the 1D flag generator class and uses an object of the latter, i.e. "line", as a member variable, which computes the SOL and EOL flags. The cnt member variable is responsible for counting the number of lines received and gets reset to zero after the EOF pixel. The frame_width and frame_height variables are updated at the start of every frame and are compared with the width and height dimension inputs to the run() function, respectively, for every pixel that isn't an SOF pixel. This is done to check if one or both of the dimension inputs changed unexpectedly. If the dimension input(s) did change unexpectedly, the incorrect_dim_change flag is set to true. More details are given in ac_flag_gen_lflush_2d Checking. W_TYPE and H_TYPE are the types for the width and height input to the run() function, respectively.

ac_flag_gen_lflush_1d Template Parameters

- AC_WIDTH – Specifies the maximum line width.
- AC_BUS_WORDS – Number of pixels that are sent in parallel over the bus interface. It is set to 1 by default.

Both these template parameters must be positive. AC_WIDTH must also be perfectly divisible by AC_BUS_WORDS. If any of these conditions are violated, a static assertion is triggered.

ac_flag_gen_lflush_2d Template Parameters

- AC_WIDTH – Specifies the maximum frame width.
- AC_HEIGHT – Specifies the maximum frame height.
- AC_BUS_WORDS – Number of pixels that are sent in parallel over the bus interface. It is set to 1 by default.
- AC_REPEAT – This parameter is set to true if the window is expected to iterate over the same set of lines for algorithms such as interpolation.

Like the ac_flag_gen_lflush_1d Template Parameters, these parameters must also be positive and AC_WIDTH must also be perfectly divisible by AC_BUS_WORDS, with static assertions being added to guard against any violation of these conditions.

Constructors

Constructor	Description
ac_flag_gen_lflush_1d()	Default ac_flag_gen_lflush_1d constructor. Calls the reset() function mentioned in Miscellaneous ac_flag_gen_lflush_1d Methods.
ac_flag_gen_lflush_2d()	Default ac_flag_gen_lflush_2d constructor. Does the following: (a) Sets cnt, frame_width and frame_height to 0. (b) Sets incorrect_dim_change to false.

Miscellaneous ac_flag_gen_lflush_1d Methods

Method	Description
void run (W_TYPE width, ac_window_v2_lflush_state window_hstate, bool &sol, bool &eol)	Class interface run function. This should be called every time the design receives a pixel. The width dimension input specifies the width of the line. The window_hstate input specifies the state the window is in with regards to its horizontal position. Please refer to Window States for Line Flushing for more detail on window states. The sol and eol flag outputs are set to true when SOL and EOL pixels are received, respectively.
void reset()	Resets class member variables: (a) Sets cnt and line_width to 0. (b) Sets incorrect_dim_change to false.
bool dim_change_error()	Returns the incorrect_dim_change member variable. More details on this variable are given in ac_flag_gen_lflush_1d Checking.

Miscellaneous ac_flag_gen_lflush_2d Methods

Method	Description
void run (W_TYPE width, H_TYPE height,	Class interface run function. This should be called every time the design receives a pixel. The width and height

ac_window_v2_lflush_state window_hstate, ac_window_v2_lflush_state window_vstate, bool &sof, bool &eof, bool &sol, bool &eol, bool repeat_line = false)	dimension inputs specify the width and height of the frame, respectively. The window_hstate and window_vstate inputs specify the state the window is in with regards to its horizontal and vertical position, respectively. Please refer to Window States for Line Flushing for more detail on window states. The sof, eof, sol and eol flag outputs are set to true when the SOF, EOF, SOL and EOL pixels are received, respectively. If the repeat_line function input and the AC_REPEAT template parameter are true, neither the SOF or EOF flags or the associated counter is updated, as the window is assumed to iterate over the same set of lines. If the repeat_line input is set to true without setting AC_REPEAT to true, an assertion is triggered.
void reset()	Resets class member variables: (a) Sets cnt, frame_width and frame_height to 0. (b) Sets incorrect_dim_change to false. (c) Calls the reset method of the line member variable.
bool dim_change_error()	Returns the incorrect_dim_change member variable. More details on this variable are given in ac_flag_gen_lflush_2d Checking.

4.10.2. Flag Generation During Extension Regions and Line Repetition

Line Flushing assumes the existence of "extension regions" which lie to the right and bottom of the image and correspond to the window's rampup. More details on these are given in Extension Regions and Window States. The extension regions preclude some or all of the input boundary pixels. It is hence important to handle flag generation in these regions appropriately.

Since the Right Extension Region (RER) lies outside the line boundary (i.e. start and end of line), ac_flag_gen_lflush_1d sets all generated flags to false in this region. This is done using the not_in_RER flag and a comparison involving window_hstate:

```
    bool not_in_RER = (window_hstate != AC_WINDOW_ER_LF);
    bool sol_temp = not_in_RER && (cnt == 0);
    bool eol_temp = not_in_RER && (cnt == width - AC_BUS_WORDS);
```

Since the bottom extension region (BER) lies outside the frame boundaries (i.e. start and end of frame), ac_flag_gen_lflush_2d sets the sof and eof outputs to false in this region. These outputs are also set to false when the window repeats over the same set of lines for algorithms like interpolation.

All of this is done through the update_vflags boolean, as shown below:

```
    bool not_in_BER = (window_vstate != AC_WINDOW_ER_LF);
    bool dont_repeat = !(AC_REPEAT && repeat_line);
    bool update_vflags = dont_repeat && not_in_BER;
    bool sof_temp = update_vflags && (cnt == 0) && sol_temp;
    bool eof_temp = update_vflags && (cnt == height - 1) && eol_temp;
```

As mentioned earlier, window_hstate and window_vstate specify the state the window is in. Please refer to Window States for Line Flushing for more details on window states.

4.10.3. Incorrect Dimension Checking

As mentioned earlier, the run() functions for both ac_flag_gen_lflush_1d and ac_flag_gen_lflush_2d accept dimension inputs, and there's a chance that these inputs could change incorrectly in between function calls. Like ac_flag_gen, ac_flag_gen_lflush also notifies the user when the dimensions are changed incorrectly.

ac_flag_gen_lflush_1d Checking

The first method is to throw a software assert whenever the width is changed mid-line. The following snippet highlights how and when this assert is triggered in the run() function:

```
    if (sol_temp) {
      line_width = width;
    } else {
      incorrect_dim_change = (width != line_width);
      #ifndef DISABLE_AC_FLAG_GEN_LFLUSH_ASSERT
      AC_ASSERT(!incorrect_dim_change, "Dimension input changed unexpectedly.
Make sure width only changes when SOL pixel is received.");
      #endif
    }
```

For every pixel that isn't an SOL pixel, the run() function compares the width input against the line_width member variable that was updated at the SOL. If the two aren't the same, the incorrect_dim_change flag is set to true and an AC_ASSERT is triggered. This assert can be disabled by defining the DISABLE_AC_FLAG_GEN_LFLUSH_ASSERT macro.

The dim_change_error() method returns the incorrect_dim_change variable for external error-checking. Unlike the AC_ASSERT, this method is synthesizable and can be used for error-checking in hardware.

ac_flag_gen_lflush_2d Checking

Like ac_flag_gen_lflush_1d, ac_flag_gen_lflush_2d also provides a software assert and sets the incorrect_dim_change variable to true when the dimensions change incorrectly, in the run() function:

```
    if (sof_temp) {
      frame_width = width;
      frame_height = height;
    } else {
      incorrect_dim_change = (width != frame_width) || (height !=
frame_height);
      #ifndef DISABLE_AC_FLAG_GEN_LFLUSH_ASSERT
      AC_ASSERT(!incorrect_dim_change, "Dimension input(s) changed unexpect-
edly. Make sure the dimensions only change when SOF pixel is received.");
      #endif
    }
```

In this case, the dimensions are only expected to change when an SOF pixel is received.

4.11. Output Flag Generator for Line Flushing (ac_flag_gen_out_lflush)

Windowed designs need a delay to ramp up. As covered in Output Flag Generator for Standard Windowing (ac_flag_gen_out), the output flag generator must take this delay into account to produce the output flags. In addition—because line flushing also has gaps in input data reads between lines—the corresponding output flag generator must take these gaps into account while generating the output valid flag. We hence use a different output flag generator than the default, with changes in the interface that take the state of the window into account.

4.11.1. C++ Code Overview

A snippet of the ac_flag_gen_out_lflush_1d class is given below.

```
template<int AC_WIN_WIDTH, int AC_BUS_WORDS = 1>
class ac_flag_gen_out_lflush_1d
{
public:
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");

  // Class methods
};
```

A snippet of the ac_flag_gen_out_lflush_2d class is given below.

```
template<int AC_WIN_HEIGHT, int AC_WIN_WIDTH, int AC_BUS_WORDS = 1>
class ac_flag_gen_out_lflush_2d
{
public:
  static_assert(AC_WIN_HEIGHT > 0, "AC_WIN_HEIGHT must be positive.");
  static_assert(AC_WIN_WIDTH > 0, "AC_WIN_WIDTH must be positive.");
  static_assert(AC_BUS_WORDS > 0, "AC_BUS_WORDS must be positive.");
  static_assert(AC_WIN_HEIGHT >= 3, "AC_WIN_HEIGHT must not be less than 3.");

  // Class methods
};
```

The template parameters for the 1D and 2D output flag generators are the same as those mentioned in ac_flag_gen_out_1d Template Parameters and ac_flag_gen_out_2d Template Parameters, respectively.

4.11.2. Output Flag Generator Class Methods

Unlike the ac_flag_gen_out classes, the ac_flag_gen_out_lflush classes do not have any member variables. As a result, there are no member variables to initialize/reset. The classes do not have any reset methods, and the default constructors included in both are empty. Information about the state of the window is passed through an input variable to the run() functions, which are described below.

A snippet of the run() function for ac_flag_gen_out_lflush_1d class is given below.

```
void run (
  ac_int<AC_WORDS, false> sol_flags, ac_int<AC_WORDS, false> eol_flags,
  ac_window_v2_lflush_state window_hstate,
  bool &sol_out, bool &eol_out, bool &vld_out)
```

sol_flags and eol_flags are the flags shifted by the ac_flag_shift library. The window_hstate input specifies the state the window is in with regards to its horizontal position. Please refer to Window States for Line Flushing for more detail on window states. The sol_out and eol_out outputs give information about the horizontal position of the overlapped windows as specified in Output Flags with Padding. The vld_out flag is set to true when the window data is valid.

A snippet of the run() function for ac_flag_gen_out_lflush_2d is given below.

```
void run (
  ac_int<AC_WIN_HEIGHT, false> sof_flags, ac_int<AC_WIN_HEIGHT, false>
eof_flags,
  ac_int<AC_WORDS, false> sol_flags, ac_int<AC_WORDS, false> eol_flags,
  ac_window_v2_lflush_state window_hstate,
  ac_window_v2_lflush_state window_vstate,
  bool &sof_out, bool &eof_out, bool &sol_out, bool &eol_out, bool &vld_out)
```

sof_flags, eof_flags, sol_flags and eol_flags are the flags shifted by the ac_flag_shift library. An object of ac_flag_gen_out_lflush_1d is declared as a local variable of the run() function and computes the sol_out and eol_out outputs. The sof_out and eof_out outputs give information on the vertical position of the overlapped windows as specified in Output Flags with Padding. The window_state and write_enabled inputs serve the same function as they do for the 1D output flag generator. The window_hstate and window_vstate inputs specify the state the window is in with regards to its horizontal and vertical position, respectively. Please refer to Window States for Line Flushing for more detail on window states. The vld_out flag is set to true when the windowed output data is valid. This flag is set depending on the window_hstate and window_vstate state variables. More details on when the output data is valid is given in Extension Regions and Window States.

4.12. Top Level Design for Line Flushing (ac_window_v2_lflush)

Similar to the AC Window 2.0 Design without Flush Support, the top-level design for line flushing also instantiates all the sub-blocks and brings them together for windowing. It can be pipelined with an II of 1 and is capable of producing multiple output windows at the same time.

Note that only two-dimensional windows are supported. For 1D Windowing, the user is advised to use the ac_window_v2_1d class described in Top-Level Design for Standard Windowing (ac_window_v2) as that has a flushing period after every line too.

4.12.1. C++ Code Overview

A snippet of the ac_window_v2_lflush_2d class is given below.

```
template <class PIX_TYPE, int AC_IMG_HEIGHT, int AC_IMG_WIDTH, int
AC_WIN_HEIGHT, int AC_WIN_WIDTH, ac_buff_type BUFF_TYPE, ac_padding_method
AC_PMODE, int AC_BUS_WORDS = 1, bool HIGH_SPEED_REPLICATE = false, bool AC_RE-
PEAT = false>
class ac_window_v2_lflush_2d
{
public:
  typedef ac_boundary_2d<PIX_TYPE, AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_PMODE,
AC_BUS_WORDS, HIGH_SPEED_REPLICATE> BOUND_2D_TYPE;
  typedef ac_int<ac::nbits<AC_IMG_WIDTH>::val, false> W_TYPE;
  typedef ac_int<ac::nbits<AC_IMG_HEIGHT>::val, false> H_TYPE;
  typedef ac_flag_shift_2d<AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS,
AC_REPEAT> FSHIFT_2D_TYPE;

  // Enum declarations.

  typedef ac_packed_vector<PIX_TYPE, AC_BUS_WORDS> LB_IN_TYPE;
  typedef ac_array<PIX_TYPE, AC_SHIFT_HEIGHT, AC_BUS_WORDS> LB_OUT_TYPE;
  typedef ac_linebuffer<LB_IN_TYPE, LB_OUT_TYPE, AC_IMG_WIDTH, AC_SHIFT_HEIGHT,
BUFF_TYPE, AC_REPEAT> LB_TYPE;

  // Other code

private:
  ac_flag_gen_lflush_2d<AC_IMG_WIDTH, AC_IMG_HEIGHT, AC_BUS_WORDS, AC_REPEAT>
fgen_2d;
  LB_TYPE linebuf_2d;
  FSHIFT_2D_TYPE fshift_2d;
  ac_shift_N_2d<PIX_TYPE, AC_WIN_HEIGHT, AC_WIN_WIDTH, AC_PMODE, AC_BUS_WORDS>
dshift_2d;

  ac_window_v2_lflush_state window_hstate, window_vstate;
};
```

The template parameters used to instantiate the class are the same as those listed in ac_window_v2_2d Template Parameters.

The following sub-blocks are instantiated as members of the class:

- 2D Synchronization Flag Generator – Instantiated as fgen_2d.
- Linebuffer – Instantiated as linebuf_2d.
- 2D Flag Shifter – Instantiated as fshift_2d.
- 2D Data Shifter – Instantiated as dshift_2d.

Since the 2D Boundary Processor and Output Flag Generator do not need any static storage, they are instantiated as variables local to the run() function described in Miscellaneous ac_boundary_2d Methods.

W_TYPE and H_TYPE are the types used for the width and height inputs to the run() function, respectively.

window_hstate and window_vstate give information about the window's state. More details on these member variables are given in Window States for Line Flushing.

Some of the enums used by the class are declared as follows:

```
  enum {
    AC_WORDS = BOUND_2D_TYPE::AC_WORDS,
    EXTRA_WORD = BOUND_2D_TYPE::EXTRA_WORD,
    AC_SHIFT_HEIGHT = BOUND_2D_TYPE::AC_SHIFT_HEIGHT,
    BER_IT = AC_WIN_HEIGHT/2 - int(AC_WIN_HEIGHT%2 == 0) + FSHIFT_2D_TYPE::EX-
TRA_LINE,
    RER_IT = FSHIFT_2D_TYPE::NUM_SHIFTS_FILL + FSHIFT_2D_TYPE::EXTRA_WORD - 1,
  };
```

AC_WORDS and EXTRA_WORD specify the width of the final windowed output as well as the width of the shift registers. An explanation on how they are computed is given in Calculating Shifter Size.
AC_SHIFT_HEIGHT specifies the height of the 2D shift register.

BER_IT and RER_IT give the number of iterations required to cover the bottom and right extension regions, respectively. A detailed description of the extension regions is given in Extension Regions and Window States. These enumerators can be useful for designs which use two for loops to process the input and output data.

Constructors

Constructor	Description
ac_window_v2_lflush_2d()	Default ac_window_v2_lflush_2d() constructor. Calls the reset() function mentioned in Miscellaneous ac_window_v2_lflush_2d Methods and also sets the padded_val value used by the boundary processor to 0.
ac_window_v2_lflush_2d(PIX_TYPE pval)	Parameterized ac_window_v2_lflush_2d() constructor. Calls the reset() function mentioned in Miscellaneous ac_window_v2_lflush_2d Methods and also sets the padded_val value used by the boundary processor to pval.

Miscellaneous ac_window_v2_lflush_2d Methods

Method	Description
void run (ac_array<PIX_TYPE, AC_BUS_WORDS> din, W_TYPE width, H_TYPE height, ac_array<PIX_TYPE, AC_WIN_HEIGHT, AC_WORDS> &dout, bool &sof_out, bool &eof_out, bool &sol_out, bool &eol_out, bool &vld_out, bool &read_data bool repeat_line = false)	Performs 2D Windowing. din is the input pixel data. The width and height dimension inputs specifies the width and height of the frame, respectively. dout is the overlapped output window from the boundary processor. sol_out, eol_out and vld_out are the output flags from the output flag generator. read_data is an output flag that can be used to gate against input reads when the window is flushing outputs in the extension regions and cannot accept inputs. Refer to Line Repetition for a detailed description of the repeat_line input.
void reset_flags()	Resets various flags internal to the design. If AC_REPEAT = false, reset_flags() is called implicitly by the run() function once the design encounters the output EOF. If AC_REPEAT = true, reset_flags() must be called explicitly by the user before they send in a frame for processing. Refer to Line Repetition for more details.
void reset()	Calls the reset() methods of all the sub-blocks except for the boundary processor and output flag generator. Also resets class member variables.

4.12.2. Block Diagram

Refer to Illustration 55 for a block diagram showing how the sub-blocks are integrated in the top-level Window Design for Line Flushing.

The block diagram is largely the same as that for Standard Windowing, but there is one key difference: The two sub-blocks coded specifically for line flushing, i.e. ac_flag_gen_lflush_2d and ac_flag_gen_out_lflush_2d, are supplied inputs from the FSMs that specify the state the window is in with respect to its horizontal and vertical position, respectively. More details on the state variables window_hstate and window_vstate are given in Window States for Line Flushing.

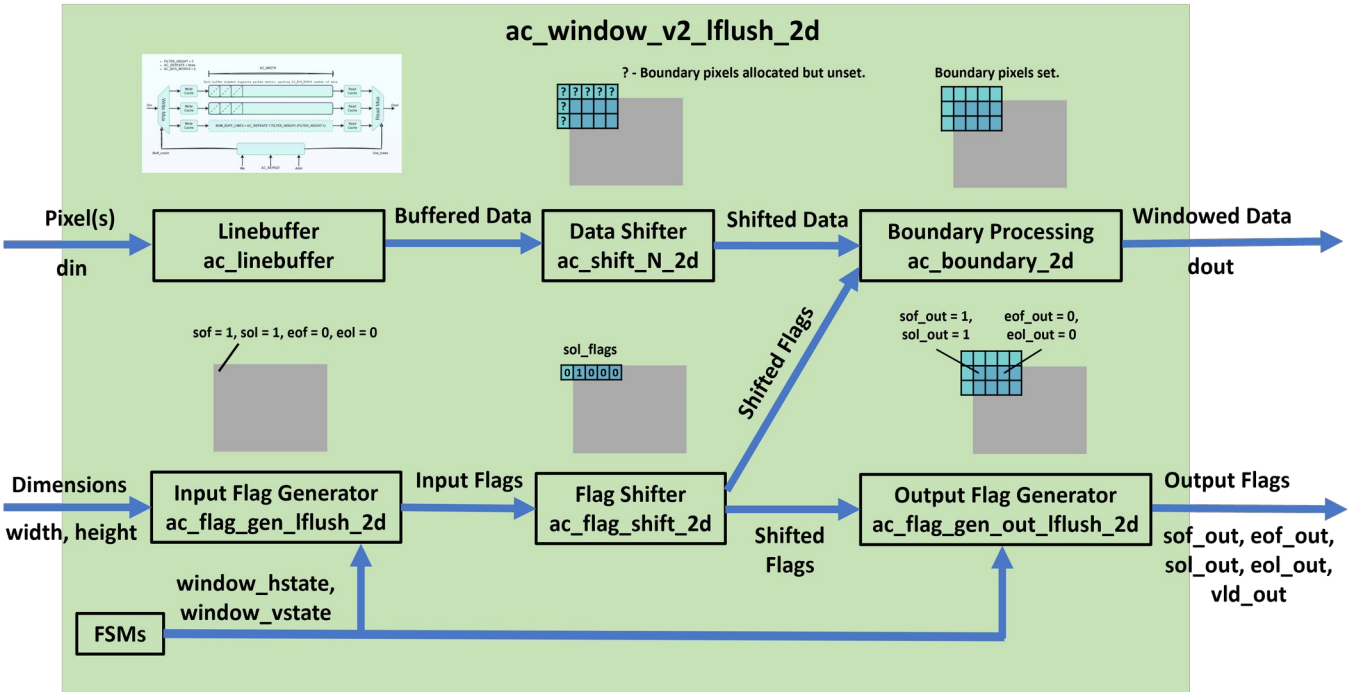

Illustration 55: Block Diagram for Line Flushing

4.12.3. Extension Regions and Window States

Standard 2D Windowing uses a single ramp-up period at the beginning of the image. During this period, it loads inputs into the linebuffer to prepare for the first output window. After this ramp-up period, it produces outputs continuously till the last output is flushed out.

Line Flushing operates differently in regards to the window ramp-up. Instead of a single ramp-up period at the beginning of the image, the window has to ramp-up in the horizontal and vertical direction. The vertical ramp-up occurs once per image and is roughly equal to half the number of window rows. The horizontal ramp-up occurs at the start of every line and is roughly equal to half the number of window columns. Since the vertical ramp-up only occurs once, before the start of the image, it corresponds to a single flushing region at the end of the image. Since the horizontal ramp-up occurs at the start of every line, it requires a corresponding flushing period at the end of every line.

Visualizing Extension Regions with for Loops

To understand the concept of extension regions, consider a simple example with the following parameters:

- 8x8 input image.
- 8-bit unsigned ac_int input and output pixels.
- 5x5 window.
- AC_CONSTANT boundary processing.

- AC_SPWRMASK linebuffers.
- AC_BUS_WORDS = 1, i.e. the default value of AC_BUS_WORDS is used.
- Inputs and outputs are streamed on ac_channels. din_ch is the input channel while dout_ch is the output channel.

```
ac_window_v2_lflush_2d<uint8, 8, 8, 5, 5, AC_SPWRMASK, AC_CONSTANT> window;
constexpr int AC_WORDS = window.AC_WORDS;
bool read_data = true;

ROW_LOOP: for (int i = 0; i < 8 + 5/2; i++) {
  COL_LOOP: for (int j = 0; j < 8 + 5/2; j++) {
    // Read input data.
    ac_array<uint8, 1> din;
    din[0] = (i < 8 && j < 8) ? din_ch.read() : uint8(0);

    // Call window's run() function.
    ac_array<PIX_TYPE, 5, AC_WORDS> window_out;
    bool sof_out, eof_out, sol_out, eol_out, vld_out;
    window.run(din, 8, 8, window_out, sof_out, eof_out, sol_out, eol_out,
vld_out, read_data);

    // Write output data.
    if (i >= 5/2 && j >= 5/2) {
      uint8 dout;
      // Code to carry out convolution with window_out and write the result of
      // the convolution to dout.
      dout_ch.write(dout);
    }
  }
}
```

As we can see above, the condition (i < 8 && j < 8) gates the input data reads. In other words, the input stream is only read when i = 0, 1, 2 ... 7 and j = 0, 1, 2 ... 7. This range of i and j values where the input data is valid is diagrammatically represented as the "Valid Input Region" in Illustration 56. For all other i and j values, a dummy input value (in this case, zero) is used.

The exit conditions of the for loops above specify that the processing will exit ROW_LOOP when i = 8 + 5/2 and COL_LOOP when j = 8 + 5/2 = 10. Hence, the range of valid i and j values to iterate within the for loops are i = 0, 1, 2 ... 9 and j = 0, 1, 2 ... 9. We can see that both loop boundaries have been extended past the Valid Input Region by 5/2 = 2 iterations. These extensions are represented diagrammatically as "Extension Regions" in Illustration 56. In these extension regions, the window will not read inputs but will flush outputs. The right extension region (RER) corresponds to the horizontal ramp-up lag, while the bottom extension region (BER) corresponds to the vertical ramp-up lag. Line flushing happens in the RER.

The vertical ramp-up is complete after the design reads AC_WIN_HEIGHT/2 rows of input data. In other words, the window is vertically ramped up when i >= 5/2. Similarly, the horizontal ramp-up is complete after

the design reads the first AC_WIN_WIDTH/2 pixels in each line, i.e. when j >= 5/2. The window is valid only when the design has ramped up vertically and horizontally, hence the output writes are gated by the condition (i >= 5/2 && j >= 5/2), i.e. (i >= 2 && j >= 2). This means that the range of i and j values for valid output data is i = 2, 3, 4 ... 9 and j = 2, 3, 4 ... 9. This range of i and j values is represented as the "Valid Output Region" in Illustration 56.

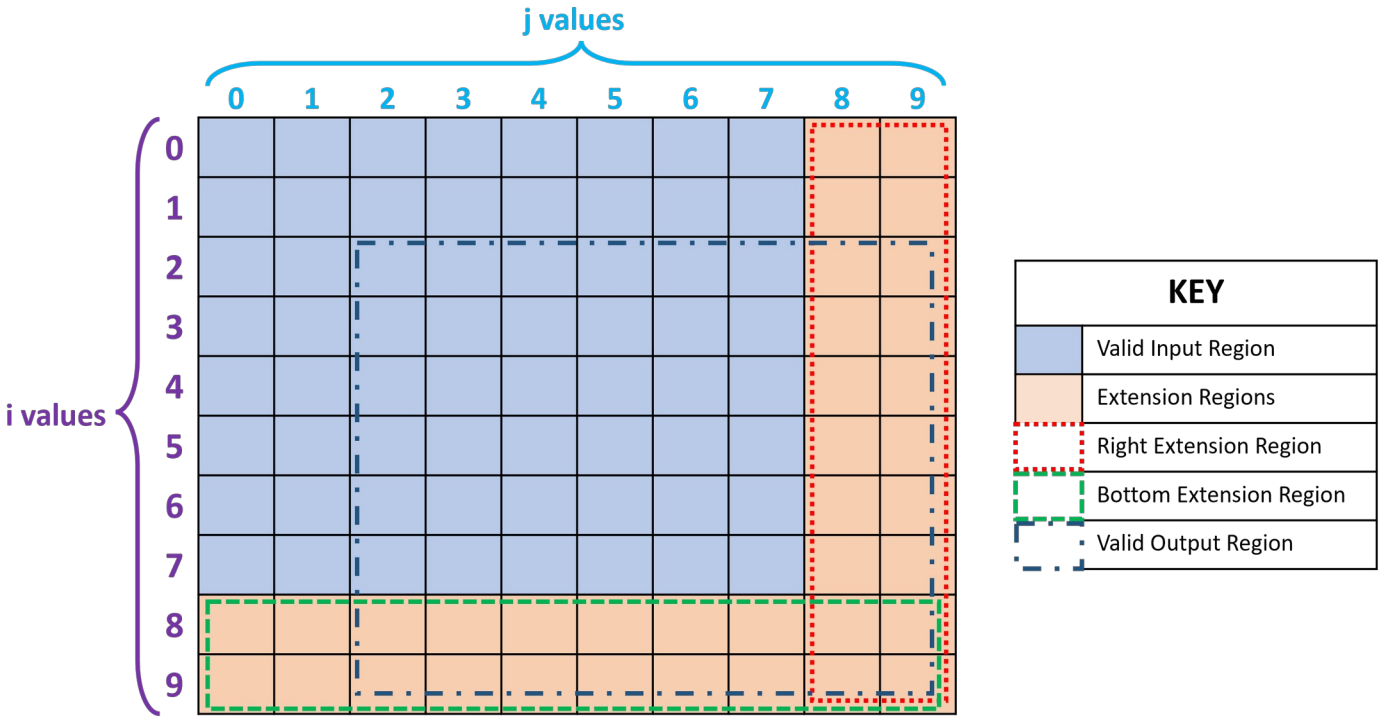

Illustration 56: Valid Input/Output and Extension Regions.

***Note**: The above coding style is not recommended and is only used for demonstration.* The following coding changes are recommended, if the user wants to adopt the coding style for their designs:

- The number of extra iterations to add for the extension regions can change depending on the type of boundary handling used, and the window size. To make sure you add the correct number of iterations in all cases, use the BER_IT and RER_IT enumerators that are supplied within the window class to add the extensions to ROW_LOOP and COL_LOOP, respectively:

```
constexpr int RER_IT = window.RER_IT;
constexpr int BER_IT = window.BER_IT;

ROW_LOOP: for (int i = 0; i < 8 + BER_IT; i++) {
  COL_LOOP: for (int j = 0; j < 8 + RER_IT; j++) {
```

  The number of column iterations can also change depending upon the AC_BUS_WORDS parameter. For the above example, AC_BUS_WORDS = 1 and the loop exit condition operates with that assumption. However, a more generic exit condition would look like this:

```
  COL_LOOP: for (int j = 0; j < 8/AC_BUS_WORDS + RER_IT; j++) {
```

- Notice that the read_data variable is never really used in the demonstration snippet. However, the user is encouraged to use it to gate input reads for their designs instead of the (i < 8 && j < 8) condition, to keep the design as generalized as possible:

```
    din[0] = read_data ? din_ch.read() : uint8(0);
```

  Since read_data must retain its value between iterations, the user must declare it outside the for loops. Since the window starts off reading inputs, read_data must be initialized to true before the for loops, ideally at its declaration. Provided that these guidelines are followed, the read_data flag will always be set to true inside the Valid Input Region and false inside the Extension Regions.

- The gating of output writes can also be generalized by using the vld_out output flag:

```
    if (vld_out) {
      uint8 dout;
      // Convolution code.
      dout_ch.write(dout);
    }
```

  The vld_out flag will be set to true inside the Valid Output Region and false outside it. It is obtained from the interface of the output flag generator described in Output Flag Generator Class Methods.

Using do-while Loops

Instead of using two for loops, a single do-while loop can also be used to process the inputs and outputs:

```
ac_window_v2_lflush_2d<uint8, 8, 8, 5, 5, AC_SPWRMASK, AC_CONSTANT> window;
constexpr int AC_WORDS = window.AC_WORDS;
bool read_data = true, eof_out = false;

do {
  // Read input data.
  ac_array<uint8, 1> din;
  din[0] = read_data ? din_ch.read() : uint8(0);

  // Call window's run() function.
  ac_array<PIX_TYPE, 5, AC_WORDS> window_out;
  bool sof_out, sol_out, eol_out, vld_out;
  window.run(din, 8, 8, window_out, sof_out, eof_out, sol_out, eol_out,
vld_out, read_data);

  // Write output data.
  if (vld_out) {
    uint8 dout;
    // Code to carry out convolution with window_out and write the result of
    // the convolution to dout.
    dout_ch.write(dout);
```

```
  }
} while (!eof_out);
```

The pattern of reading inputs and writing outputs will be the same as that shown in the example that uses for loops, it's just more difficult to visualize as we don't have loops with clear iterators and exit conditions that depend on the image/window dimensions. In the above example, we rely entirely on flags generated by the window class for input/output gating as well as loop control:

- Input gating is done using read_data. read_data must still be declared external to the loop and initialized to true before the design enters the loop.
- Output gating is done using vld_out.
- The loop exits when eof_out is set to true, meaning the we've reached the end of the output frame and the design is done flushing outputs. Since eof_out is used as the loop exit condition, it must be declared outside the do-while loop.

Window States for Line Flushing

Regardless of the type of loop(s) used, the window always keeps track of its position and can determine which region it is in through the use of two state variables window_hstate and window_vstate. The former is used to determine the horizontal position of the window, while the latter is used to determine the vertical position. These are declared as data members of the window class and are of type ac_window_v2_lflush_state, an enumeration declared in the header file ac_window_v2_lflush_enums.h:

```
enum ac_window_v2_lflush_state {
  AC_WINDOW_RAMP_LF,
  AC_WINDOW_BETWEEN_LF,
  AC_WINDOW_ER_LF,
};
```

The state each enumerator represents is is given below:

- AC_WINDOW_RAMP_LF – Rampup state. This represents the horizontal ramp-up for window_hstate and the vertical rampup for window_vstate.
- AC_WINDOW_ER_LF – Extension region. This represents the RER for window_hstate and roughly represents the BER for window_vstate.
- AC_WINDOW_BETWEEN_LF – This represents the part of the image between the ramp-up and the extension region.

window_hstate and window_vstate cycle through these values as the design processes the inputs and outputs. The transition between these states for both window_hstate and window_vstate is represented diagrammatically in Illustration 57. The example used in the illustration and the associated parameters are the same as those discussed in Visualizing Extension Regions with for Loops.

Note: The region where window_vstate equals AC_WINDOW_ER_LF does not precisely match up with the BER because the transition from AC_WINDOW_BETWEEN_LF to AC_WINDOW_ER_LF happens after the

last input pixel is read. This is done to simplify the FSM code. The overall algorithm still functions as expected.

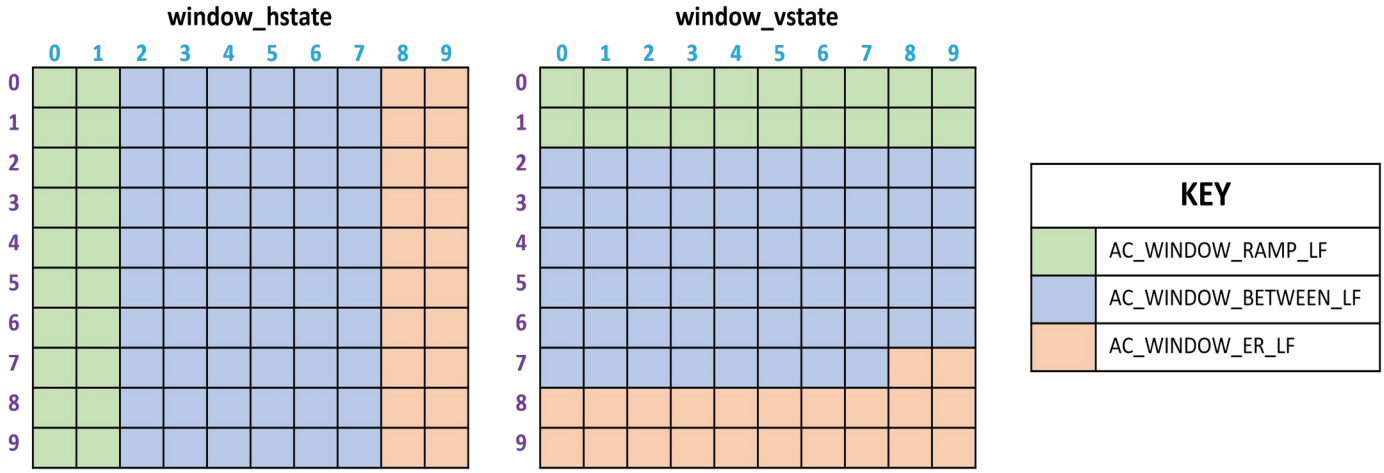

Illustration 57: window_hstate and window_vstate Values in Different Image Regions

4.12.4. Line Repetition

Image scaling and interpolation require the window to iterate over the same set of lines when the height of the resized output image is greater than the height of the original image. To accommodate this requirement, the window must not slide vertically while it goes over the repeated lines. To stop the window from sliding vertically in such a case, the design does the following:

- The write enable (“we”) input to the linebuffers is set to false. The linebuffers do not accept new inputs and only output previously stored data.
- The repeat_line input to the flag generator and shifter blocks is set to true. Like the linebuffers, the vertical flag shifters, i.e. the sof_regs and eof_regs data members in ac_flag_shift_2d, do not accept new inputs.

Support for line repetition calls for unique hardware requirements that are not expected when performing standard 2D convolution. Setting the AC_REPEAT template parameter to true will ensure that the appropriate hardware is synthesized. The design must also know when to repeat the same set of lines and when to process new lines at run-time. If the repeat_line input to the run() function is set to true, the design will perform line repetition. If it is set to false, the design will process new lines, as per its default behavior.

If AC_REPEAT = true, it is assumed that the design is resizing the input image. In such a case, the output EOF might not be a reliable indicator of when to call reset_flags() and the design does not implicitly call it as a result. The user must explicitly call reset_flags() right before they send a frame in for processing. The window_vstate value is set to AC_WINDOW_RAMP_LF in the reset_flags() method, as the FSM cannot reliably change the vertical window state to the rampup state at the start of the next input frame.

Setting repeat_line to true without setting AC_REPEAT to true first is invalid, as it would imply that the user expects the design to perform line repetition despite not having synthesized the appropriate hardware. A software assert (AC_ASSERT) is provided to guard against this condition during C simulation.

4.12.5. Limitations

- Only 2D windows are supported.
- Boundary padding of some form is always enabled. A static assertion is triggered if the user tries to instantiate the window with AC_NO_PADDING as the padding mode.
- If using 2D windows, the pixel type must be compatible with the ac_packed_vector container class. Currently, only the ac_int, ac_fixed and RGB_pv (in ac_pixels.h) datatypes as well as the ac_packed_vector type itself have this compatibility. Support for floating point datatypes might be provided with a future release.
- Windows less than three rows tall are not supported if boundary padding is used, i.e. AC_WIN_HEIGHT must not be less than 3 in such a case.
- The design currently only supports a stride of 1. Strides greater than 1 might be supported in a future release.

Chapter 5: AC Window 3.0 Blocks

5.1. Version History

The following table compares AC Window 3.0 to previous AC Window versions and highlights what's new:

Feature(s)	Original	v2.0	v3.0
Common Features: • In-built buffering, shifting, boundary padding and output flag generation. • Standard Windowing • Line Flushing • 1D and 2D windowing • Can be pipelined with an II of 1.	☑	☑	☑†
Multiple inputs/outputs processed per clock cycle	☒	☑	☑
Singleport write-mask memories supported	☒	☑	☑
Modular design allows reuse of sub-blocks	☒	☑	☑
Flushing Support provided	☒	☑	☑‡
Strides > 1 supported	☒	☒	☑
Guaranteed linebuffer partitioning	☒	☒	☑
Guaranteed register-mapping for all non-linebuffer arrays	☒	☒	☑

5.2. Features

- **Highly Configurable:** The user can configure a wide array of parameters such as the window size, stride, boundary padding, linebuffer architecture and image dimensions.
- **HLS Optimized**: Efficiently coded in C++ using high-level synthesis (HLS) best practices.
- **Robust Testbench**: Comprehensive testbenches validate the output against golden reference values.
- **Optimal QofR**: Multiple inputs and outputs can be processed per clock cycle, with optimized latency, area and performance.
- **Scalable Array Mapping**: Linebuffer arrays will always be partitioned into memory banks as needed, no matter the size or number of arrays. All other arrays will always be mapped to registers.
- **Modular Design:** Sub-blocks can be re-used between designs as needed.

† Line Flushing will be added with a future release.
‡ Flushing Support will be added with a future release.

5.3. Introduction

Image windowing is commonly used to extract a portion of the image (the window) for Convolutional Neural Networks and Image Resizers. Since this serves as a building block for more complex applications, there are a wide variety of configurations which any window functions need to accommodate. The AC Window 3.0 libraries serve to meet those requirements so that the user can focus on other aspects of their design. This document presents the corresponding catapult toolkit which provides usage examples of the AC Window 3.0 libraries.

5.4. Windowing Types

As of Catapult 2025.4, only standard windowing is supported in AC Window 3.0, but the aim is to introduce line flushing and flushing support with future Catapult versions. The following table compares the windowing types supported and highlights the trade-offs involved.

	Standard Windowing	Line Flushing	Flushing Support
Between-line gaps	☒	☑	☒
Suitable looping styles	do-while	do-while, for	do-while
Simultaneously processing different frames	☒	☒	☑
Optional Boundary Padding	☑	☒	☒
1D Windows Supported	☑	☒	☒

5.5. Block Diagram

The block diagram for the standard windowing design is shown in Illustration 58.

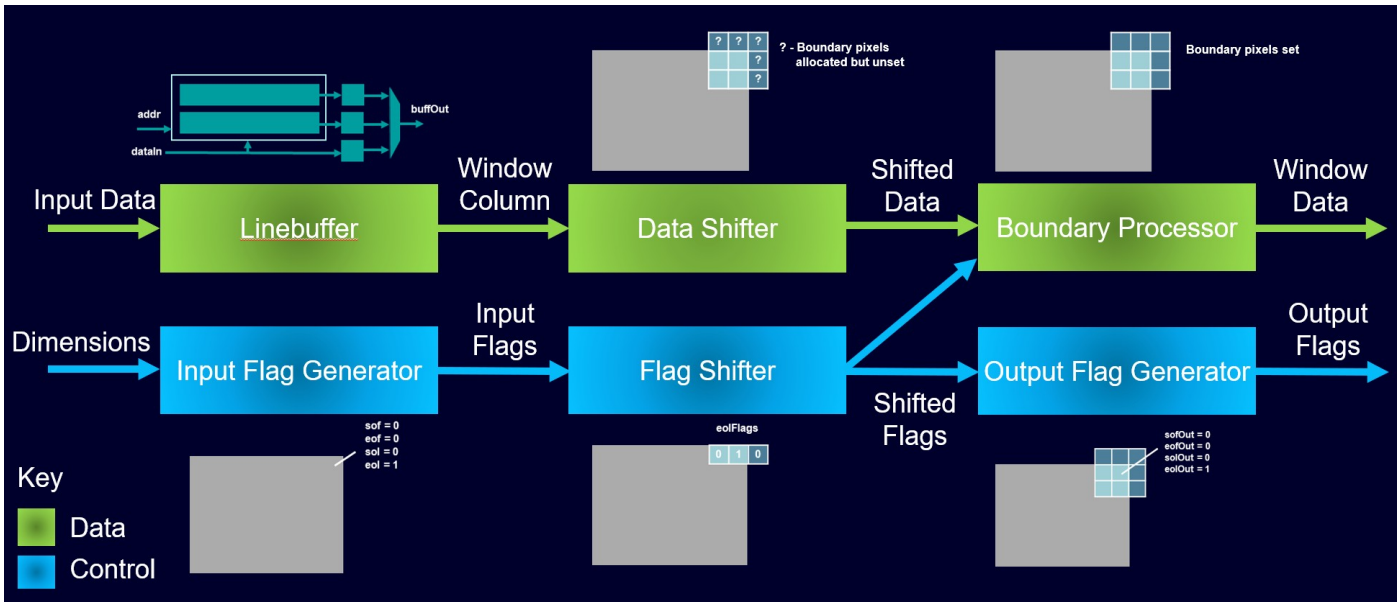

Illustration 58: Block Diagram

The sub-blocks can broadly be divided into the data and control sub-blocks. The data sub-blocks are primarily concerned with buffering the image, shifting the window and padding boundary values at the appropriate places to produce the final image window. The control sub-blocks generate the input flags (sof, eof, sol and eol), shift them such that they are aligned with the shifted window data and can be used to discern the boundary regions for boundary padding. They also produce flags corresponding to the final output window.

5.6. Configurable Parameters

The AC Window 3.0 designs are implemented using class templates. The table below describes the template parameters that are used to configure the model for a specific hardware implementation.

Parameter	Description
Pix_type	Input pixel type
MaxHeight	Maximum image height
MaxWidth	Maximum image width
WinHeight	Window height
WinWidth	Window width
WinMode	Specifies various window properties. A detailed description is given here.
BusWords	Number of pixels processed simultaneously
VertStride	Window stride in the vertical direction.
HorizStride	Window stride in the horizontal direction.

Note that if MaxWidth is not a power of two, the size of the linebuffer arrays won't be a power of two either. In such a case, it's recommended to use the following Catapult directives to optimize array indexing and makes sure that the design schedules when pipelined with an II of 1:

```
directive set -ARRAY_INDEX_OPTIMIZATION true
directive set -ASSUME_ARRAY_INDEX_IN_RANGE true
```

5.6.1. WinMode

The WinMode template parameter defines various properties of the window. To set it, the user must perform a bitwise OR operation to combine various enumerators defined in ac_window_def_v3.h, with each enumerator representing a property.

The table below categorizes and describes some of the properties.

Category	Enumerator	Description
Linebuffer Architecture	AC_SPWRMASK	Singleport memory with write masking.
	AC_SPCIR	Banked singleport memories with individual banks accessed in a circular buffer fashion.
	AC_SPSHIFT	Banked singleport memories with individual banks accessed in a shift register fashion.
	AC_1R1W	1R1W memories.
Boundary Padding	AC_CONSTANT	Constant value padding. Default value is zero. Non-default values can be set through the window ctor.
	AC_REPLICATE	Closest edge pixel replicated across image boundary.
	AC_REFLECT	Pixels reflected across boundary, with the inclusion of edge pixels.
	AC_REFLECT101	Pixels mirrored across boundary, without the inclusion of edge pixels.
	AC_NO_PADDING	No boundary padding. Only available with standard windowing.
Windowing Types	AC_STANDARD_WINDOWING	Standard windowing. Window does not accept input during flushing period. Default windowing type.
	AC_LINE_FLUSHING	Line Flushing. Window inputs and outputs have gaps in between each line. Scheduled for a future release.
	AC_FLUSHING_SUPPORT	Flushing support. Window may accept inputs during flushing period to reduce rampup for the next frame. Scheduled for a future release.

Properties within each category are mutually exclusive. For instance, you cannot use AC_SPCIR and AC_1R1W linebuffers as you can only pick one linebuffer architecture out of the four provided. Static assertions are provided to check for this.

In addition to these, the following miscellaneous window properties are also available:

- HIGH_SPEED_REPLICATE: AC_REPLICATE boundary padding requires a shorter critical path when mapped to a combinational CCORE, making it useful for high-frequency applications.

- AC_REPEAT: The window can iterate over a set of lines for applications like image resizing.
- AC_REVERSE_PADDING: Reverses the padding format for even size kernels, which require asymmetric padding. For instance, a 4x4 window will have 2 pixels padded on the top and left and 1 pixel padded on the bottom and right. The default is 1 pixel on the top and left and 2 pixels on the bottom and right. This reversed padding format is compatible with the TensorFlow library padding. This mode cannot be used with AC_REFLECT101 padding.
- AC_EXTERNAL_FLAG_INPUTS: The window switches to accepting input flags rather than image dimensions. This is useful for applications where the flags are pre-computed. If this is specified, the runFlags(...) method of the window class must be used.

5.6.2. Assertions

The following assertions are built into the design:

Condition	Assert Type	Message
MaxHeight > 0	Static	MaxHeight must be positive.
MaxWidth > 0	Static	MaxWidth must be positive.
WinHeight > 0	Static	WinHeight must be positive
WinWidth > 0	Static	WinWidth must be positive
BusWords > 0	Static	BusWords must be positive.
MaxWidth%BusWords == 0	Static	MaxWidth must be divisible by BusWords.
VertStride > 0	Static	Vertical stride must be positive.
HorizStride > 0	Static	Horizontal stride must be positive.
HorizStride <= BusWords	Static	Horizontal stride must not exceed BusWords.
BusWords % HorizStride == 0	Static	BusWords must be divisible by horizontal stride.
width <= MaxWidth	Dynamic	Image width must not exceed the maximum limit.
width%BusWords == 0	Dynamic	Image width must be divisible by BusWords.
height <= MaxHeight	Dynamic	Image height must not exceed the maximum limit.