Infra-red Based Image Processing

Szilárd Hegedűs
szilard.hegedus@student.unitbv.ro

Submitted for the 2019 Digilent Design Contest Europe

May 3, 2019

Advisor: Dan Nicula

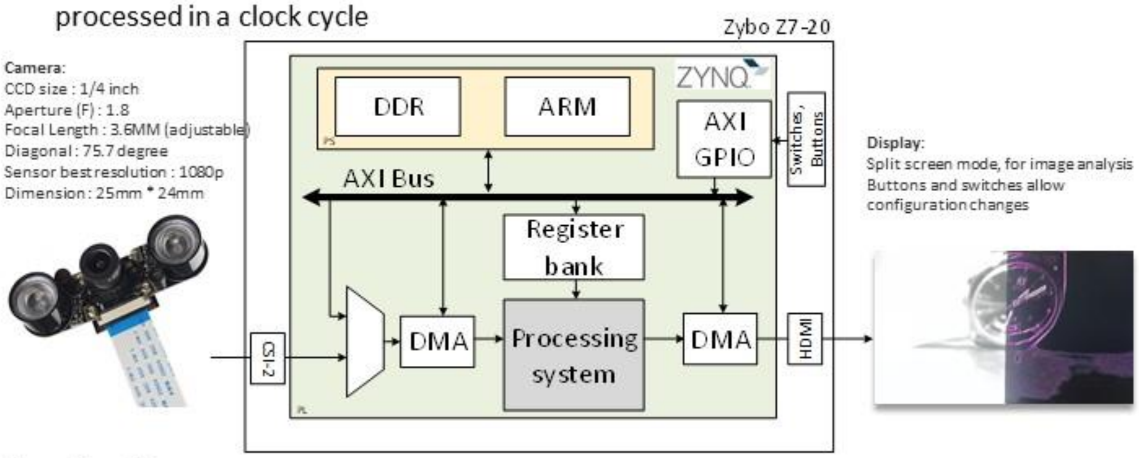
Transilvania University
Brasov, Romania

Infrared Image Processing Unit

Features:
- Configurable algorithm succession
- Configurable used algorithm number
- Resolution: 1080p@30Hz
- Frequency: 100 MHZ
- Double DMA allow multiple pixels to be processed in a clock cycle

Implemented algorithms:
- Dead/Stuck pixel correction
- Median filter
- Image smoothing
- Image sharpening
- Edge detection

Camera:
CCD size : 1/4 inch
Aperture (F) : 1.8
Focal Length : 3.6MM (adjustable)
Diagonal : 75.7 degree
Sensor best resolution : 1080p
Dimension : 25mm * 24mm



Display:
Split screen mode, for image analysis
Buttons and switches allow configuration changes

Applications

- Image effects
- Image enhancement

- Driver behavior detection
- Road detection

Working and comparison of different image processing algorithms

- Blood analysis
- Tumor detection
- Cell mineral analysis

Contents

Introduction

Infra-red image processing is trending in the automotive industry as the cars get more sophisticated, we can face detection and Eyegaze solutions. It is widely used because it does not depend on ambient light it has it own light source that luminates the target and IR waves reflected are captured with the specific camera. Furthermore, sunglasses do not block IR, it is intended for visible light and the eyes can be seen even sunglasses on.

We can observe infra-red imaging in medical applications the veins and the blood flow in them can be tracked using this technology than can't be done with RGB cameras.

The most common and known application is in security cameras for the same reason as in automotive its night vision property, there are come cameras can are hybrid with some filters they are RGB at day time, note that for these the red color isn't shown as red is closer to pink, and IR at night. These can be found as night vision cameras.

Nowadays all cameras support some image processing algorithms that are already integrated in the silicon next to the CMOS sensor. These can be automatic white balance, black balance, exposure control, band filter, 60/50 Hz detection and so on. Some programmable features like mirror/flip, crop, windowing, panning and others.

Abstract

Hardware extendable and software configurable image processing unit written in Verilog, that applies different algorithms to an input frame. In this case five algorithm are presented: dead/stuck pixel correction, image sharpening, image smoothing, median filtering and edge detection. All this in real time, with a resolution of 1080p@30Hz.

Objectives

- Extend the existing Pcam 5C demo
- Add image processing algorithms to the existing chain
- Design a hardware extendible module
- Software programmable module
- Use the Zybo Z7-20 CSI connector with an existing camera on the market

Features-in-Brief

- There are five image processing algorithms in the IP these can be applied in any order and number
- Processing 1 pixel/clock
- Maximum tested frequency 150 MHz on Zybo Z7-20 board
- Resource occupation 35% LUT and BRAM

Project Summary

The project extends the Pcam 5C demo project, by inserting an image processing module with five filters:

- Dead/stuck pixel correction
- Median filter
- Image sharpening
- Image smoothing
- Edge detection

Each of these can be software configured in witch order or how many of these will be applied to the input stream.

The camera is an OV5647 night vision camera intended for the Raspberry Pi, but has the same CSI-2 connector as the Zybo Z7.

Digilent Products Required

Zybo Z7-20 SoC

Tools Required

- Soldering iron
- Wire
- Logic analyzer
- Raspberry Pi compatible camera

Design Status

Implemented, tested. Can be further developed

Background

Why This Project?

Infra-red image processing started to take over the automotive industry, because it is not sensitive to visible light. Application like tracking human behavior in the car is monitored to prevent accidents, these can be falling asleep while driving keeping track if the driver is paying attention to the road and so on.

IR cameras can see the human eye even through sunglasses, and the cameras image is not affected from daylight or if its night, and there is from very little to none ambient light. The cameras have their own light source, LED's mounted next to the lens.

This project shows some image processing algorithms that improve the quality of the image for it to be further processed. Algorithms like face detection and/or Eyegaze are very common application. But for them to work properly good quality image is remanded at the algorithm input.

Why These Algorithms?

Dead/stuck pixels are an issue in any camera system. This means there some pixel values that are always constant, the sensor does not detect properly. These must be replaced; they are considered to be noise.

Median filtering is a common preprocessing method that smooths the image and eliminates noise statistically, this all pixels in neighborhood have high probability to same similar values so if any noise gets in this area, by sorting it this value/noise will get in of the ends of the sorted array. At edges median filtering attenuates them and the sorting technique is best for spiky noise.

Image sharpening accentuates the edges, any low-pass will attenuate edges in exchange to reduce noise. Sharpening is a method to restore a blurred image.

Smoothing filter reduces noise in an image, using a 2D convolution. The kernel has a gaussian distribution that will not blur the image as much as the median filter, but it not that efficient on spiky noise.

Edge detection is used commonly for feature extraction algorithms or sharpening where a proportion of the edges are added to the blurred image to regain its details.

All these five algorithms are based upon kernel and neighborhood processing in the accent of this project is not on the algorithms but the structure of the system where these can be applied in anny succession to an input image.

Reference Material

Yon, J. J., Mottin, E., Biancardini, L., Letellier, L., & Tissot, J. L. (2003). Infrared microbolometer sensors and their application in automotive safety. In *Advanced Microsystems for Automotive Applications 2003* (pp. 137-157). Springer, Berlin, Heidelberg.

Reich, G. (2005). Near-infrared spectroscopy and imaging: basic principles and pharmaceutical applications. *Advanced drug delivery reviews*, *57*(8), 1109-1143.

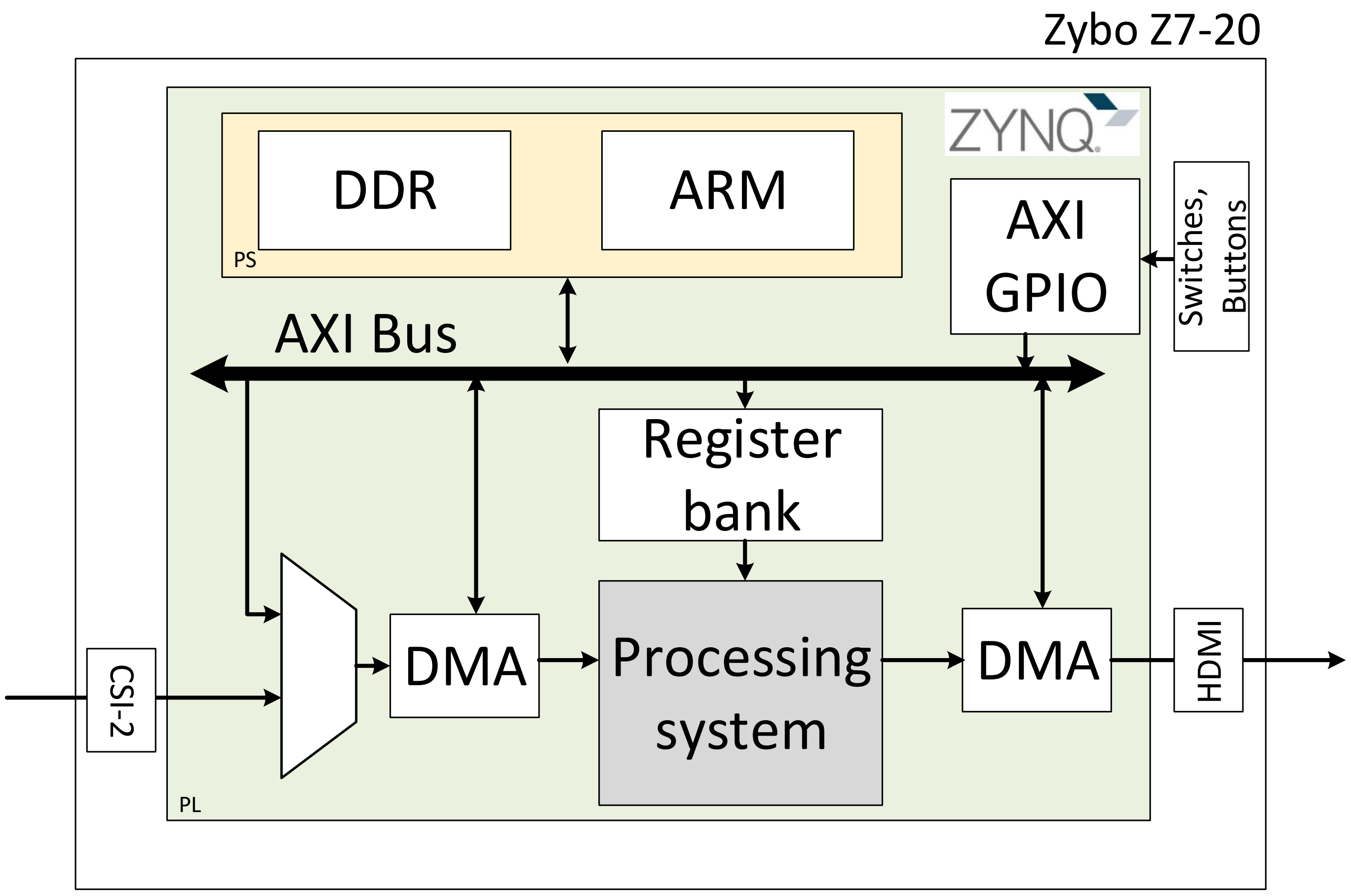
Stein, G. S., Shashua, A., Gdalyahu, Y., & Liyatan, H. (2010). *U.S. Patent No. 7,786,898*. Washington, DC: U.S. Patent and Trademark Office.

Hirota, M., Ohta, Y., & Fukuyama, Y. (2008, May). Low-cost thermo-electric infrared FPAs and their automotive applications. In *Infrared technology and applications XXXIV* (Vol. 6940, p. 694032). International Society for Optics and Photonics.

Ibarra-Castanedo, C., Gonzalez, D., Klein, M., Pilla, M., Vallerand, S., & Maldague, X. (2004). Infrared image processing and data analysis. *Infrared physics & technology*, *46*(1-2), 75-83.

Diakides, M., Bronzino, J. D., & Peterson, D. R. (Eds.). (2012). *Medical infrared imaging: principles and practices*. CRC press.

Design



The design consists of two modules ir_filters, that will process the image from the input, and an axi2frame module than can read an image from the DDR memory and feds that image as input, the later module is used for testing if there is no camera available. This architecture extends the Pcam 5c demo.

Features and Specifications

[Ft 1] 24-bit, 1 pixels RGB (8-bit each layer) grayscale input image only

[Ft 2] 24-bit, 1 pixels RGB output, each byte represents one output pixel

Data	Byte
Data [23:16]	Blue
Data [15:8]	Green
Data [7:0]	RED

Table 1, Data format

Pixel
r7 r6 r5 r4 r3 r2 r1 r0 g7 g6 g5 g4 g3 g2 g1 g0 b7 b6 b5 b4 b3 b2 b1 b0

Table 2, Byte format

[Ft 3] Configurations can be changed only when block is disabled

[Ft 4] Support: Maximum Image width 2048 pixels; minimum image width 4 pixel

[Ft 5] The output image size is equal to the input image size, the result will have 1-pixel width junk on the border for each **PE** it passes through, as shown in Figure 4, Input/output format (for 1 algorithm)

[Ft 6] Configurable input source for each 5 **PE:**

1) Each **PE** can take the input frame from the other **PE**s outputs, or from system input
2) A **PE** cannot take its output as input
3) Each **PE** can be used only once during a frame

[Ft 7] **PE** configuration signals:

PE	Register
Dead/stuck pixel correction	cfg_dpc
Median filter	cfg_med
Low-pass filter	cfg_lpf
Sharpening	cfg_sharp
Edge detector	cfg_edge

Table 3, Configuration naming

[Ft 8] Filter input selection codes

PE output	Selection code
Dead/stuck pixel correction	00001
Median filter	00010
Low-pass filter	00100
Sharpening	01000
Edge detector	10000

Table 4, Selection codes

For each selector the code for the corresponding **PE** input will be the global input.

Exapmle1: cfg_dpc(00001) means that the dead pixel correction module will receive the global input

Example2: cfg_med(00001) means that the median filter module will receive the output of the dead pixel correction **PE**.

Configuration details

Example1: If given the configuration below

Register	Configuration
cfg_dpc	00001
cfg_med	00010
cfg_lpf	00100
cfg_sharp	01000
cfg_edge	10000

Table 5, Configuration Example 1

The data will travel as shown below:

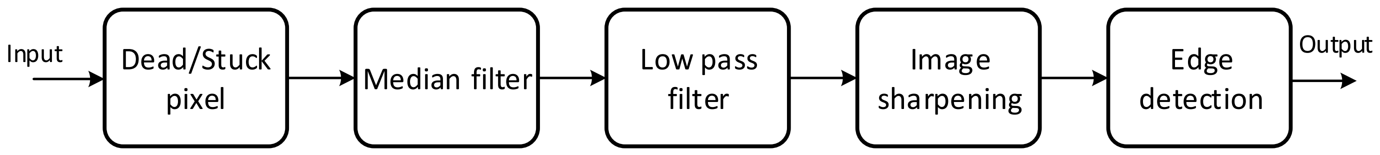


Figure 1, Data flow for the above configuration

Example 2:

Register	Configuration
cfg_dpc	00000
cfg_med	00000
cfg_lpf	01000
cfg_sharp	00001
cfg_edge	10000

Table 6, Configuration Example 2

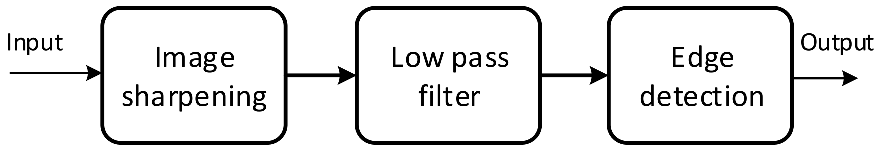


Figure 2, Example 2 wiring

SW2, SW1, SW0	Filters
0 0 0	Transparent
0 0 1	Dead stuck pixel correction
0 1 0	Median filter
0 1 1	Laplace filter
1 0 0	Smoothing filter
1 0 1	Sharpening filter
1 1 0	Smooth + Laplace filter
1 1 1	Smooth + Sharpening + Laplace filter

Table 7, Switch configurations for filters

[Ft 9] Switch configs can be changed only when module is disabled

[Ft 10] SW3 changes between original i9mage and camera input

[Ft 11] For testing and image must be copied to an SD card all color planes in 3 different files. These will be read and written to the DDR memory to the specified address in the C code

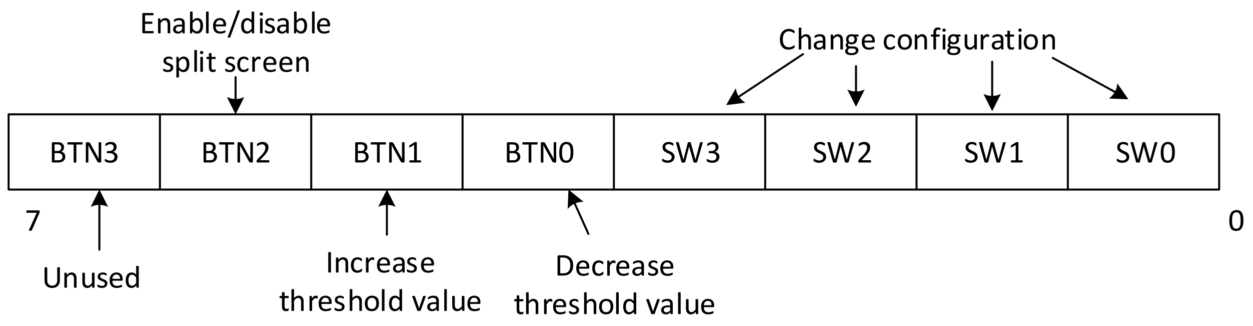


Figure 3, GPIO bits for buttons and switches

Line Buffer features

[Ft 1] Byte/pixel input data (8-bit each pixel)
[Ft 2] **FI** input
[Ft 3] 3x3 image segment output
[Ft 4] Adapts **FI** control signals for the output frame
[Ft 5] Self-reset at start of frame

Filter features

[Ft 1] Byte/pixel input (24-bit each pixel)
[Ft 2] Uses a configuration threshold for the permitted maximum error
[Ft 3] Output has the same size as the input, but with junk data on 1-pixel border for each filter applied
[Ft 4] Input and output both on **FI**

Additional features:

[Ft 1] Sharpening filter has a coefficient of the mask that is added to the input image, the number is an integer on 4 bits
[Ft 2] Dead pixel correction uses a threshold for the permitted maximum error
See *Table 8,* Algorithm description and ***Error! Reference source not found.*** for details

Design Overview

The **Ir_filters** module applies 5 different image processing algorithms in a user defined number and order on an input IR image (a single plane of data).

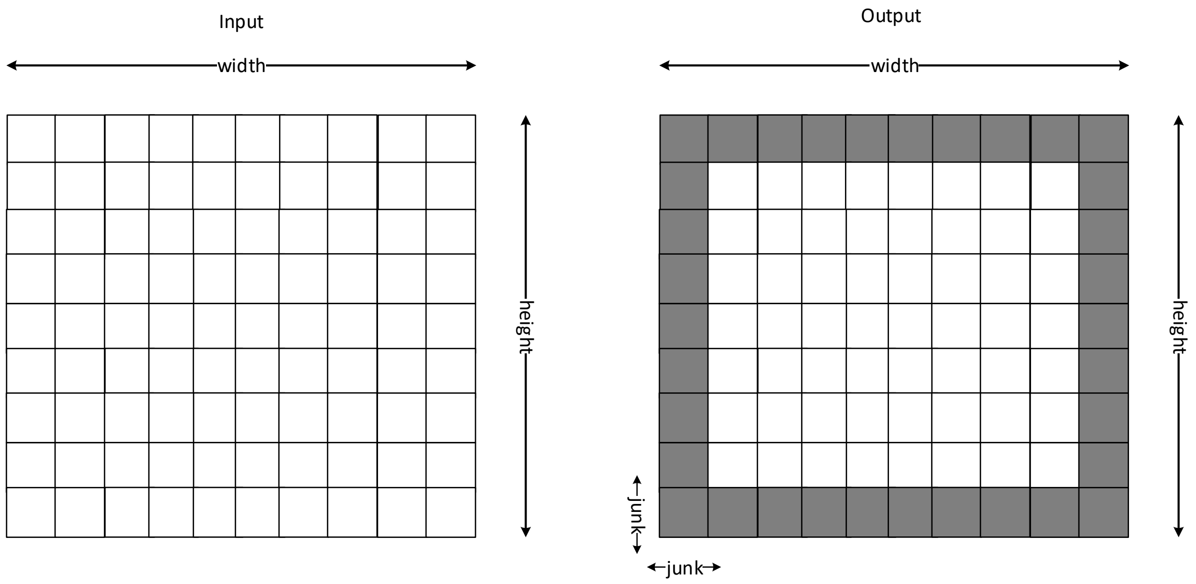


Figure 4, Input/output format (for 1 algorithm)

The processed, output image, has the same size as the input but its borders are junk data (Figure 4, Input/output format (for 1 algorithm)3). The width of the border containing invalid values depends on the number of algorithms that is applied to the input. The size of the junk on the borders can varies between 1 and 5.

The description of these algorithms is described in Table 8, Algorithm description:

	Description
Stuck/dead pixel correction	Correct dead or stucked pixels in an image, caused by the sensor, by verifying if the center pixel is more different from its neighbors. If the difference is higher than a specified threshold, it is replaced by with the average of its neighbors. $\begin{array}{\|c\|c\|c\|}\hline \boldsymbol{N_1} & \boldsymbol{N_2} & \boldsymbol{N_3} \\ \hline \boldsymbol{N_4} & \boldsymbol{P_{in}} & \boldsymbol{N_5} \\ \hline \boldsymbol{N_6} & \boldsymbol{N_7} & \boldsymbol{N_8} \\ \hline \end{array}$ $P_{out}(x,y) = \begin{cases} P_{in}(x,y), \lvert P_{in}(x,y) - N_i \rvert < threshold, i = \overline{1,8} \\ \frac{1}{8}\sum_{i=1}^{8} N_i\,, otherwise \end{cases}$
Median filter	Applies median filter to input image, replacing the center pixel from a 3x3 kernel with the median value of the 9 pixels.

Low-pass filter (smoothing filter)	Applies a mean filter to the input image, using the kernel: $$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$
Image Sharpening	Sharpens the input image using a Laplacian filter. The image is passed through convolved with M, the result is subtracted from the input, the result is called mask. The mask is added to the original image. $$P_{out}(x,y) = P_{in}(x,y) + coef * (P_{in}(x,y) - P_M(x,y))$$ $$P_M(x,y) = \frac{1}{8}\sum_{i=-1}^{1}\sum_{j=-1}^{1} P_{in}(x+i, y+j) * M(x+i, y+j)$$ $$M = \begin{bmatrix}0 & -1 & 0\\ -1 & 4 & -1\\ 0 & -1 & 0\end{bmatrix}$$
Edge detection	Detects edges by using a Laplacian filter, using the kernel below: $$\begin{bmatrix}0 & -1 & 0\\ -1 & 4 & -1\\ 0 & -1 & 0\end{bmatrix}$$

Table 8, Algorithm description

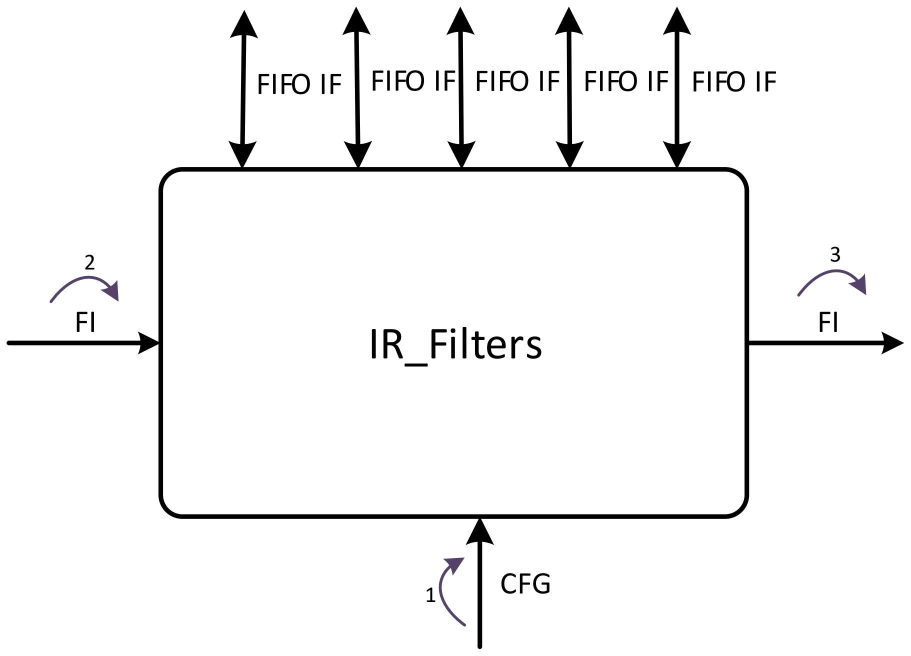


Figure 5, Ir_filters symbol

Flow description

1) The module is configured then enabled
2) Input image is provided on input **FI_i**
3) Output is calculated according to the configured algorithm order and sent to **FI_o**
4) FI protocol violated at output interface if disabled before end of frame

Interfaces

The diagram presented by Figure 6, ir_filters architecture for the interfaces:

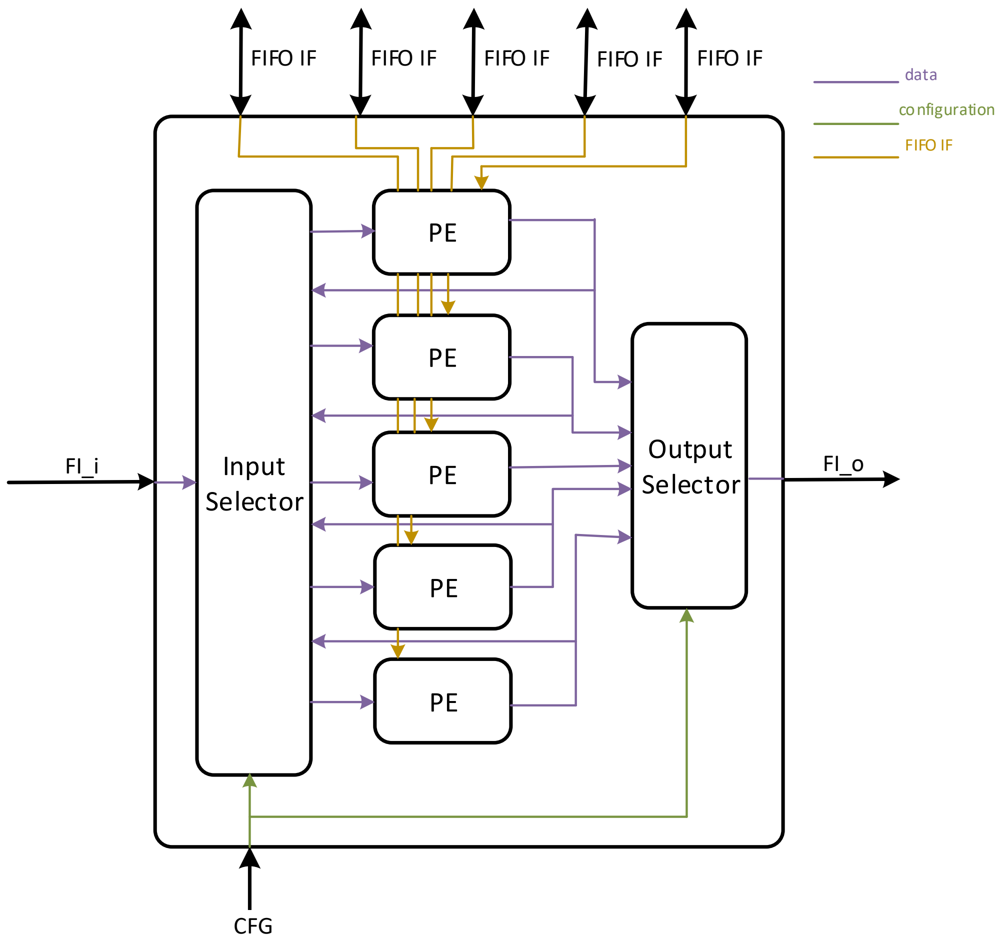


Figure 6, ir_filters architecture

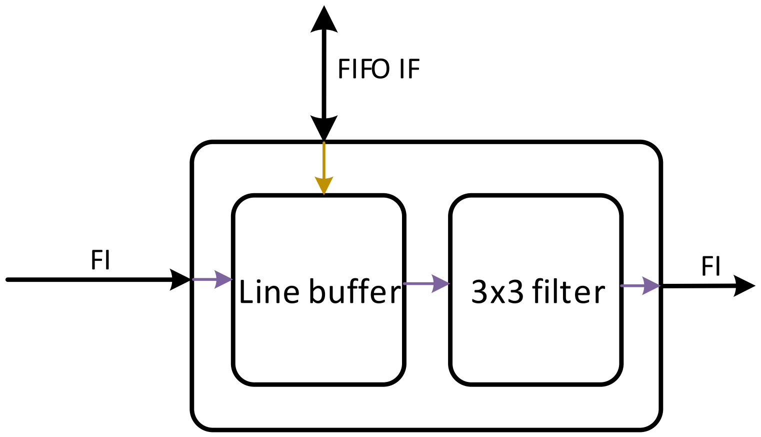


Figure 7, Processing Element architecture

System IF

Signal name	Functionality	I/O	Width [bits]
Clk	System Clock	I	1
rst_n	Asynchronous system reset active low	I	1

Configuration interface

Signal name	Functionality	I/O	Width [bits]
cfg_blk_en	Block enable	I	1
cfg_dpc	Dead pixel correction (one-hot)	I	5
cfg_med	Median filter	I	5
cfg_lpf	Low-pass filter	I	5
cfg_sharp	Sharpening filter	I	5
cfg_edge	Edge detection	I	5
cfg_dpc_thr	Dead pixel correction threshold	I	8
cfg_sharp_coef	Sharpening module coefficient (integer)	I	3

Input Frame interface

Signal name	Functionality	I/O	Width [bits]
frm_i_rdy	Module is ready to receive the data	O	1
frm_i_val	Data valid	I	1
frm_i_data	Input Data (4 pixels per cycle)	I	32
frm_i_sof	Start of Frame	I	1
frm_i_eof	End of Frame	I	1
frm_i_sol	Start of Line	I	1
frm_i_eol	End of Line	I	1

Output Frame interface

Signal name	Functionality	I/O	Width [bits]
frm_o_rdy	The target is ready to receive the data	I	1
frm_o_val	Data valid	O	1
frm_o_data	Output Data (4 pixels per cycle)	O	32
frm_o_sof	Start of Frame	O	1
frm_o_eof	End of Frame	O	1
frm_o_sol	Start of Line	O	1
frm_o_eol	End of Line	O	1

FIFO interface

Signal name	Functionality	I/O	Width [10* bits]
lb_fifo_push	Push data	O	1
lb_fifo_pop	Pop data	O	1
lb_fifo_pushdata	Input data	O	32
lb_fifo_empty	Fifo empty	I	1
lb_fifo_full	Fifo full	I	1
lb_fifo_popdata	Output data	I	32

Detailed Design Description

The **ir_filters** module processes an 8-bit greyscale/infrared image, using 5 different processing units:

- Dead/Stuck pixel correction
- Median filtering
- Mean filtering
- Image sharpening
- Edge detection

The order in which the algorithms are applied is configurable and not all must be applied, but each can be applied only once and one processing units' output shouldn't be feedback as input.

Internal design and flow

The **ir_filters** has 5 processing units for each algorithm, all these modules get its input from a line buffer, providing 3x1 or a 3x3 kernel. The data flow is managed by an arbiter, that also generates the interrupt.

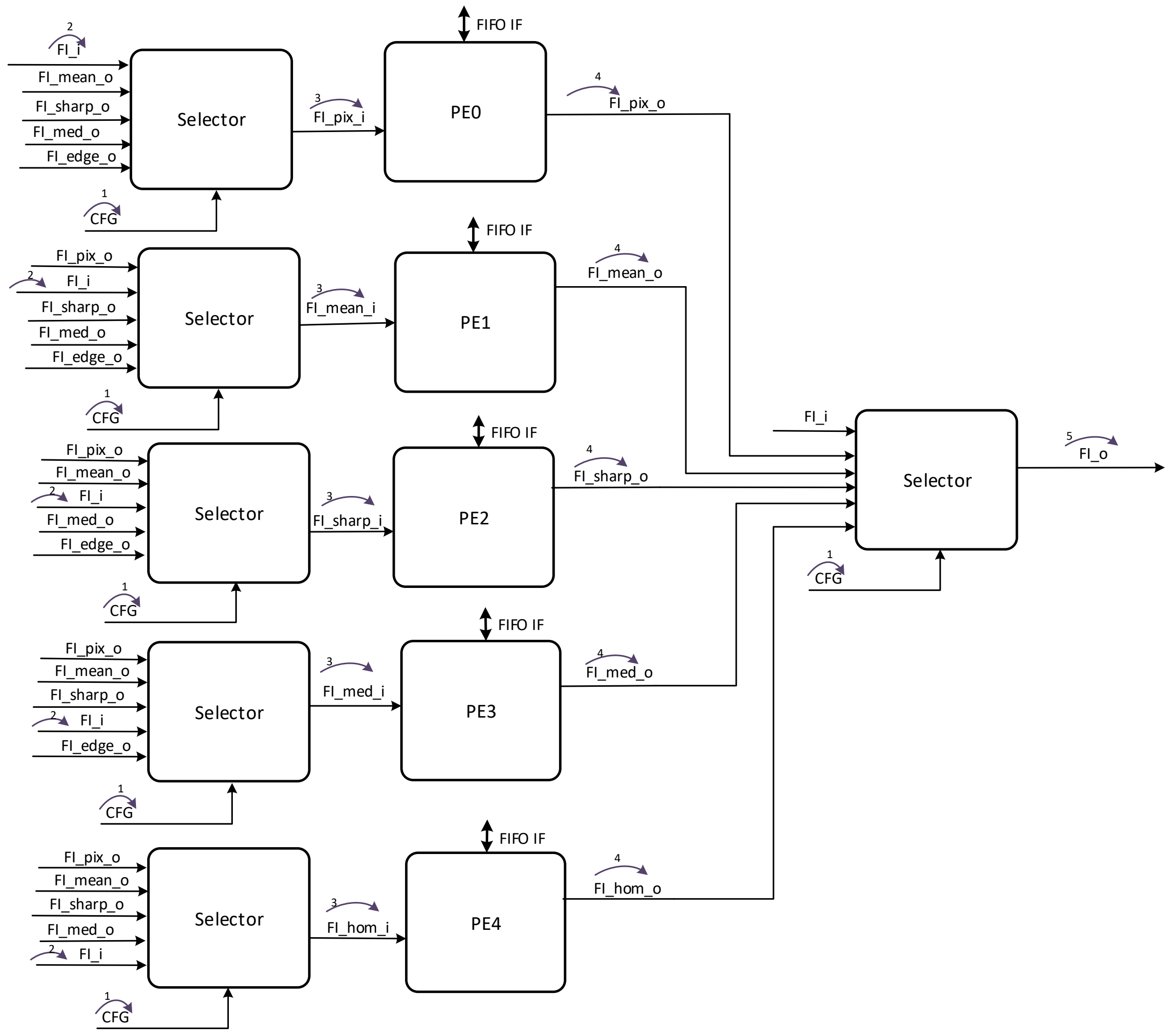


Figure 8, ir_filters internal design and flow

Flow description

1) Module is configured than enabled
2) Image is provided on the input FI
3) Input is provided to the processing units depending on the configured order
4) Every processing unit output will be sent to the next processing unit input
5) Final output is sent on output **FI**

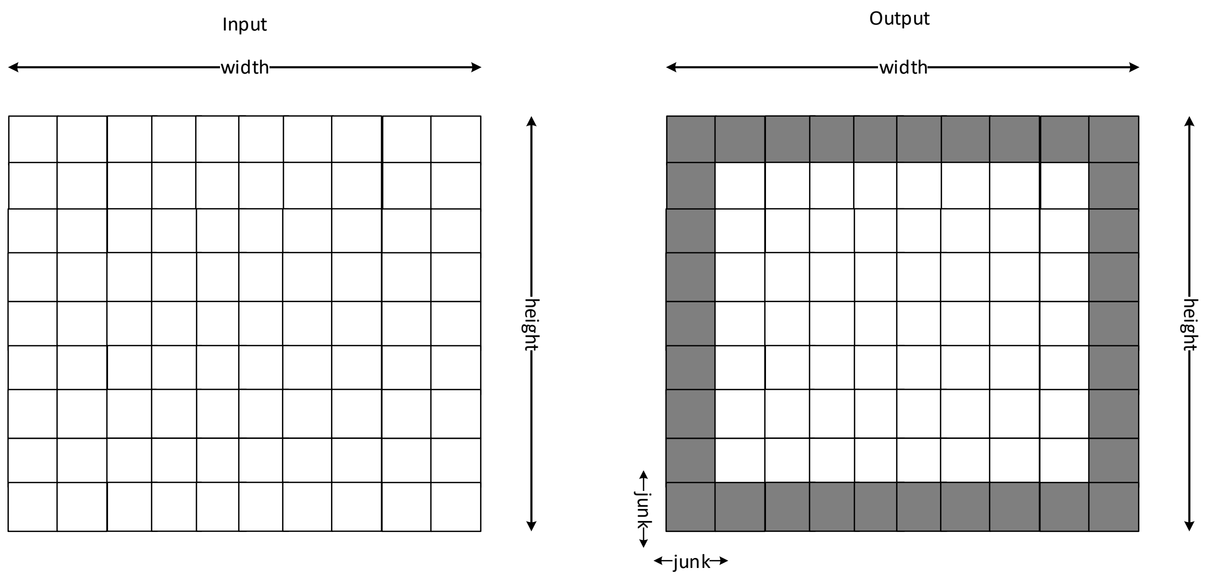


Figure 9, Input/Output format (for 1 algorithm)

The processed, output image, has the same size as the input but its borders are junk data (Figure 9, Input/Output format (for 1 algorithm)). The width of the border containing invalid values depends on the number of algorithms that is applied to the input. The size of the junk on the borders can varies between 1 and 5.

IR_FILTERS Module Description

ir_filters Module

This module is the top module that contains all the blocks described above.

ir_filters module interfaces

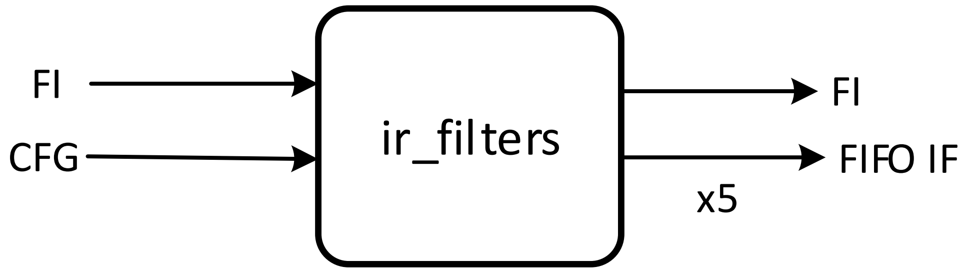


Figure 10 Top module interfaces

IR_FILTERS internal design

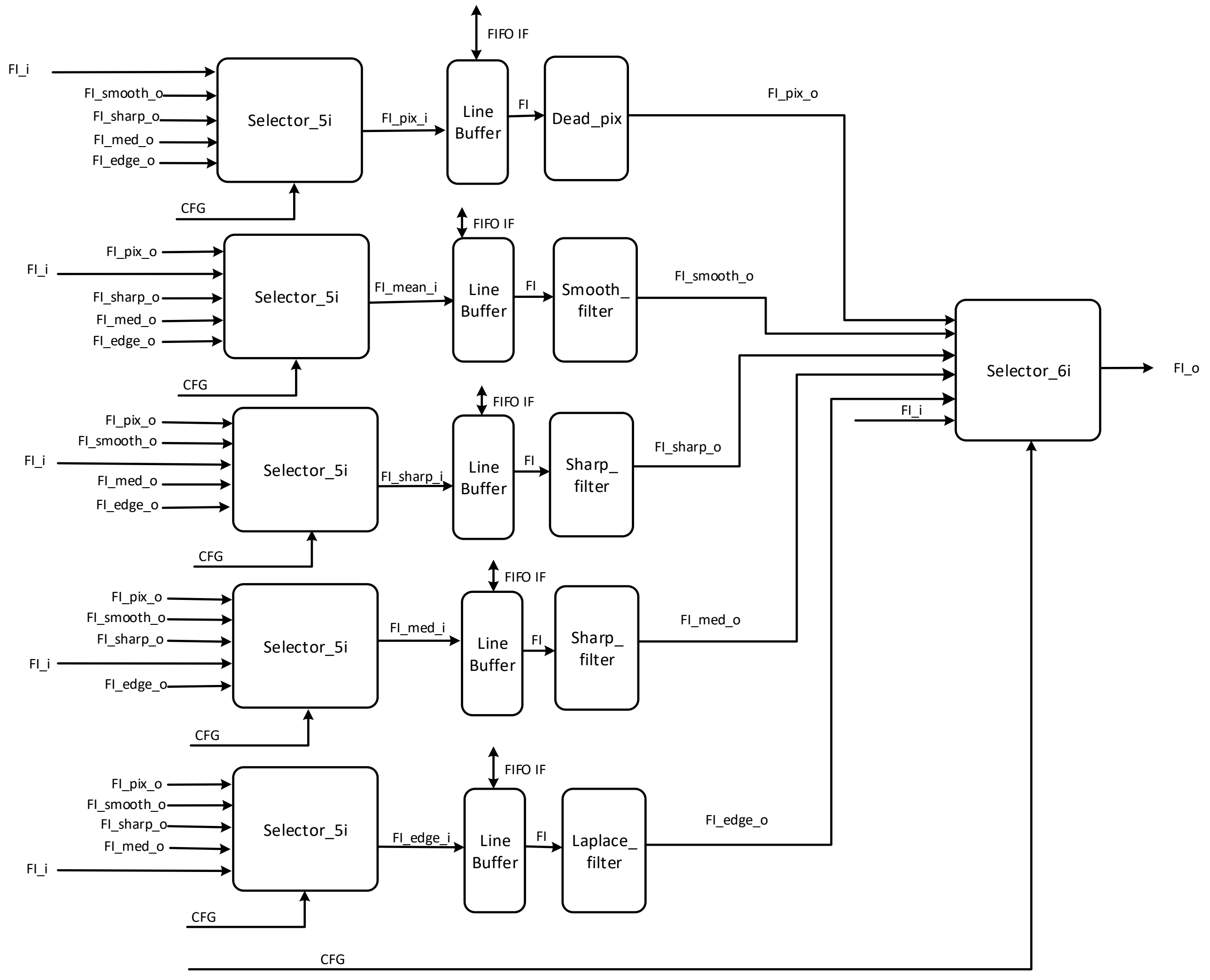


Figure 11 ir_filters internal design

Line Buffer Module

Line buffer module gets an input image on the **FI** and outputs all 3x3 or 3x1 kernels for further processing.

P00	P01	P02
P10	P11	P12
P20	P21	P22

Table 9, line_buffer output data

Line buffer interfaces

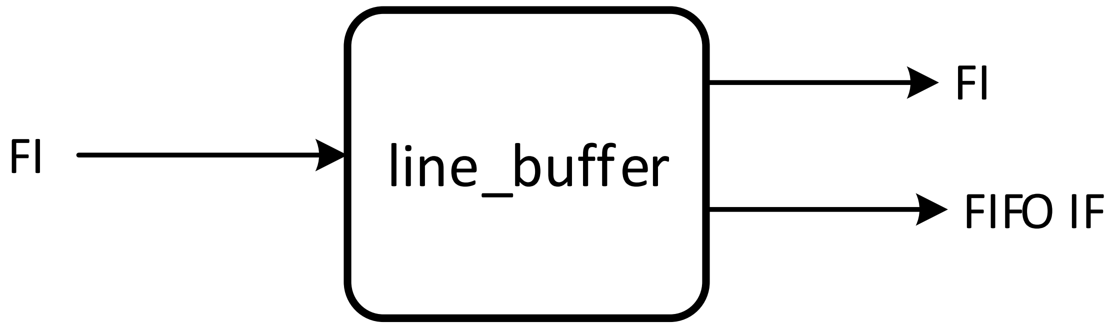


Figure 12 Line buffer interfaces

Line buffer internal design

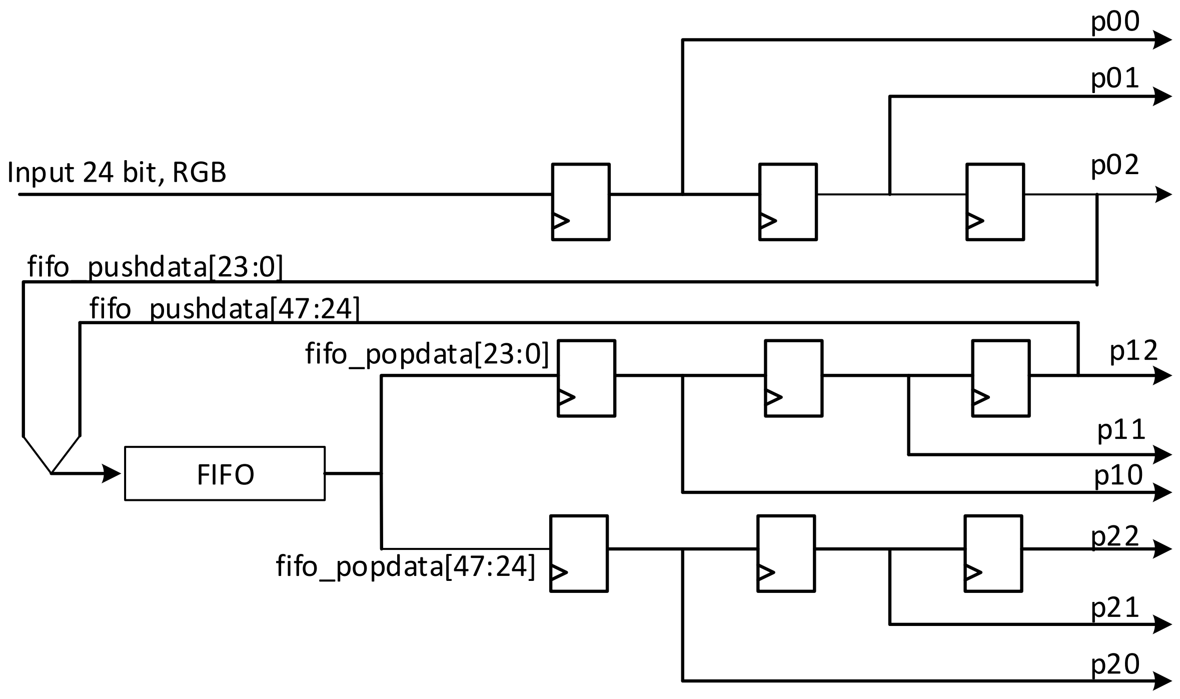


Figure 13, Line buffer internal design

The full image is fed into the module, a FIFO is used with double the width than the input pixels width. The data is delayed than fed as the lower half of the FIFO input data. At the output that data is further delayed getting the second row of the 3x3 matrix and now this data is fed as the upper half of the FIFO input and at the output it will represent the third row of the 3x3 window.

The delay between the input and the output of the FIFO is one-line width, the depth of the FIFO is 2048, the cameras image is 1920 pixels wide. The height if the image negligible, this module work for any width greater than three.

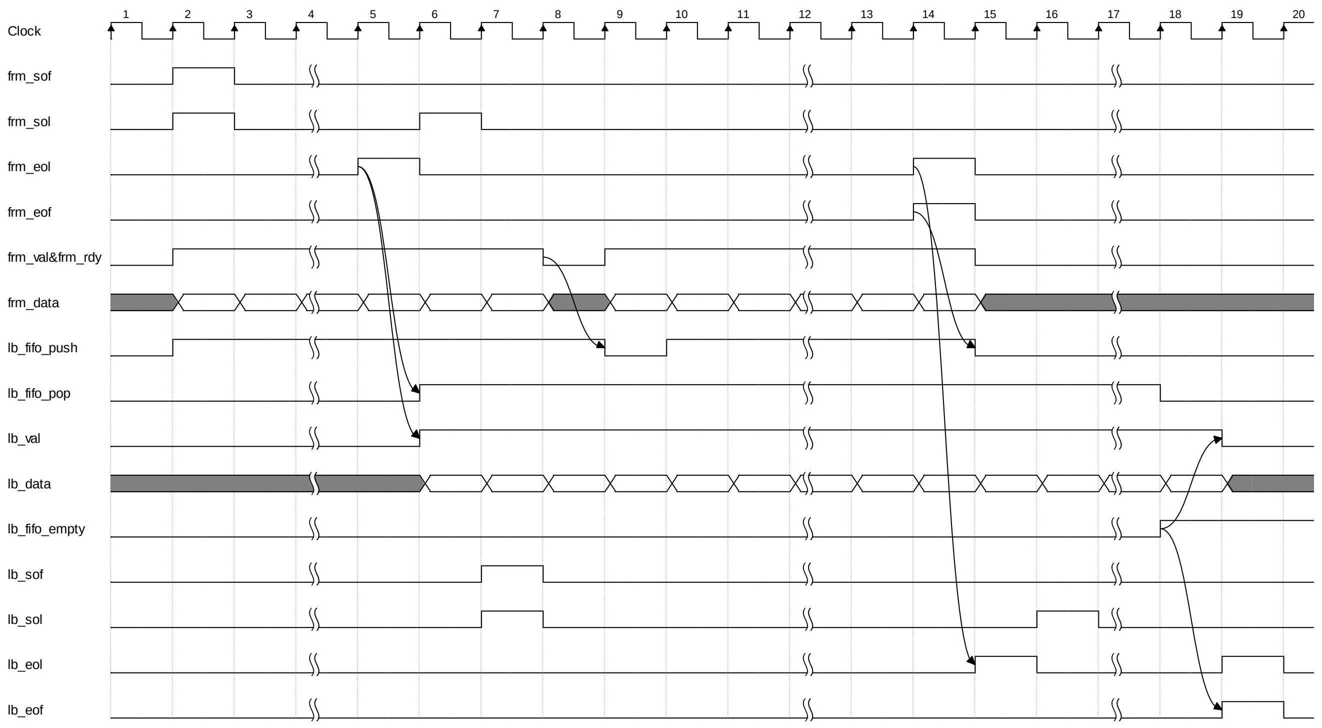


Figure 14, line_buffer timing diagram

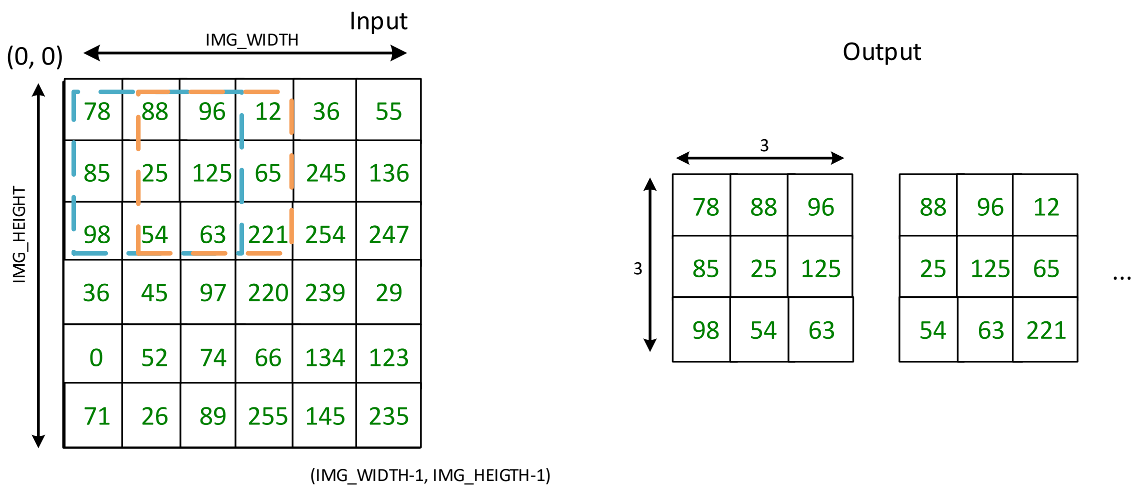


Figure 15, Sliding window working principle

The windows move in raster order, as shown in *Figure* 15, Sliding window , the orange window represents the first window and the purple the second one.

Smoothing filter module

Applies mean filter on 3x3 sequence provided from a line buffer.

Smoothing filter internal design

See Table 9, line_buffer output data, for pixel inputs.

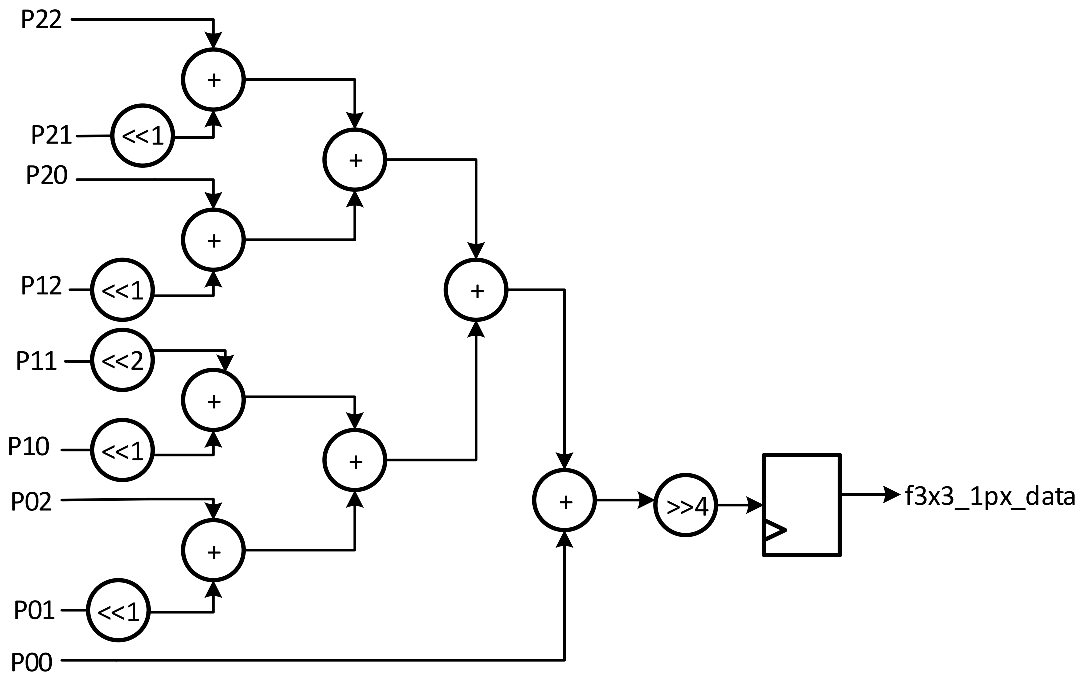


Figure 16, smooth_filter3x3 internal design

The module convolves 9 input pixels with the mask $\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix}$.

This architecture calculates one pixel in a cycle, it must be instantiated four times.

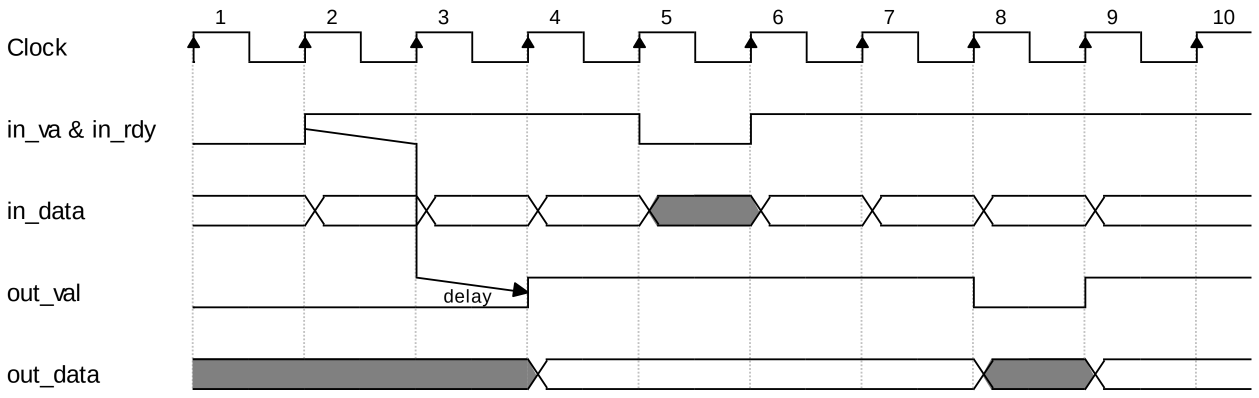
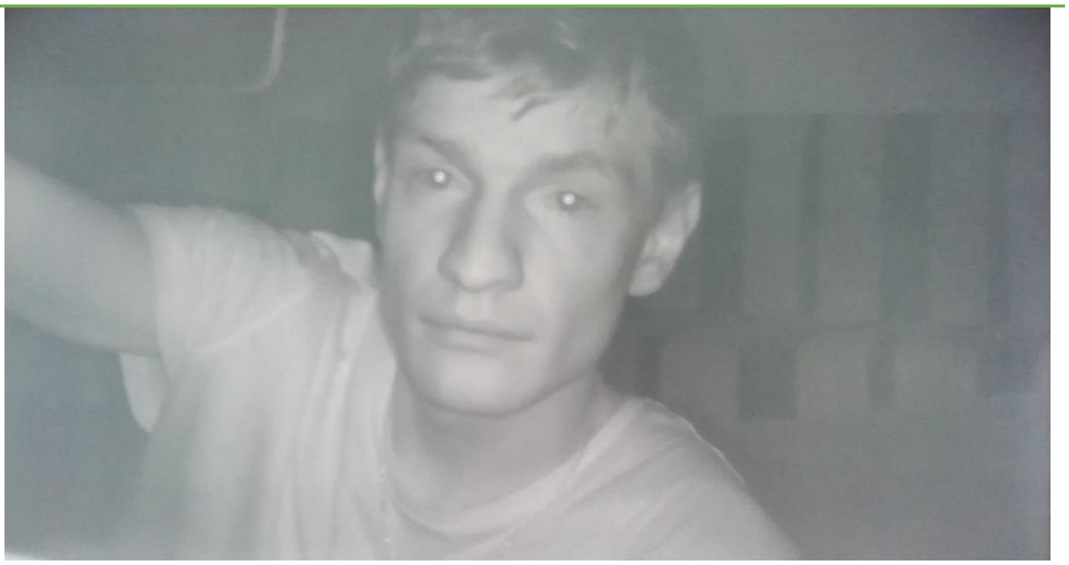
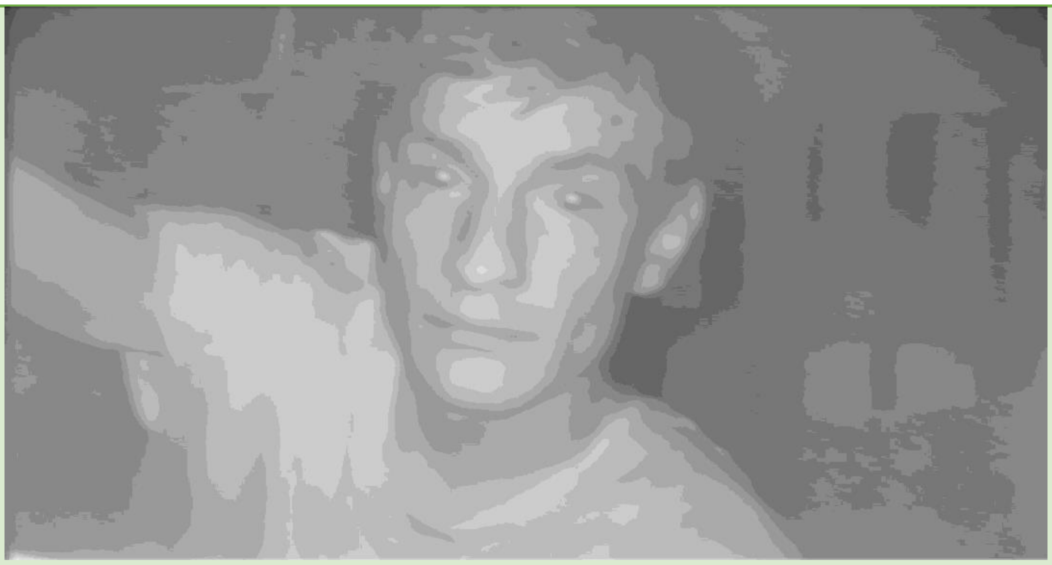


Figure 17, smooth_filter timing diagram

Input	
Output: **Blurred** **Noise reduced**	

Dead pixel correction module

The module corrects dead or stuck pixels by subtracting the center pixel from all its neighbors and if the absolute value exceeds the given threshold, the pixel will be replacing with de average of its eight neighbors.

Dead pixel module internal design

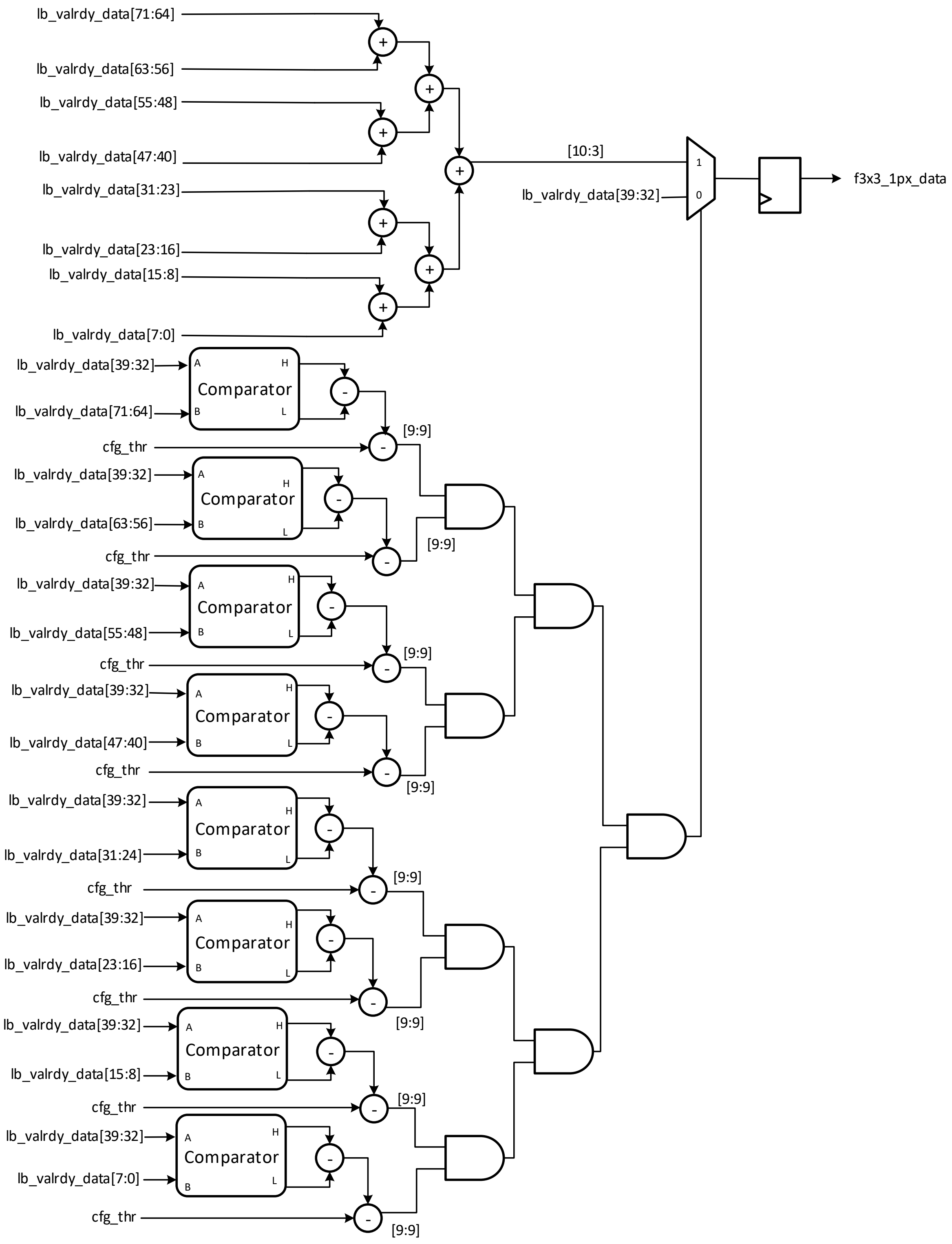


Figure 18, dead_pix_3x3 correction internal design

The output data is changed only if the center pixel of the 3x3 mask is different from its neighbors. That is checked by firstly calculating max(center, neighbor) – min(center, neighbor) and from this result the threshold is subtracted. All eight sign bits are checked if all subtractions are negative that means |center - neighbor| is less that the configured threshold and the output will be the average of the pixels around the center one.

$$P_{out}(x,y) = \begin{cases} P_{in}(x,y), |P_{in}(x,y) - N_i| < threshold, i = \overline{1,8} \\ \frac{1}{8}\sum_{i=1}^{8} N_i \text{, } otherwise \end{cases}$$

N_1	N_2	N_3
N_4	P_{in}	N_5
N_6	N_7	N_8

$$|P_{in}(x,y) - N_i| < threshold$$
$$\max(P_{in}(x,y), N_i) - \min(P_{in}(x,y), N_i) < threshold$$
$$\max(P_{in}(x,y), N_i) - \min(P_{in}(x,y), N_i) - threshold < 0$$

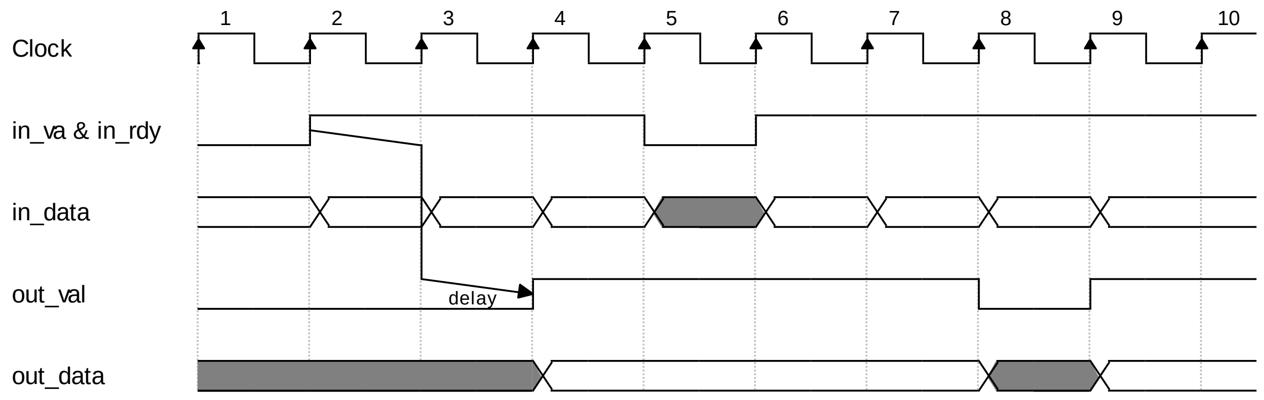
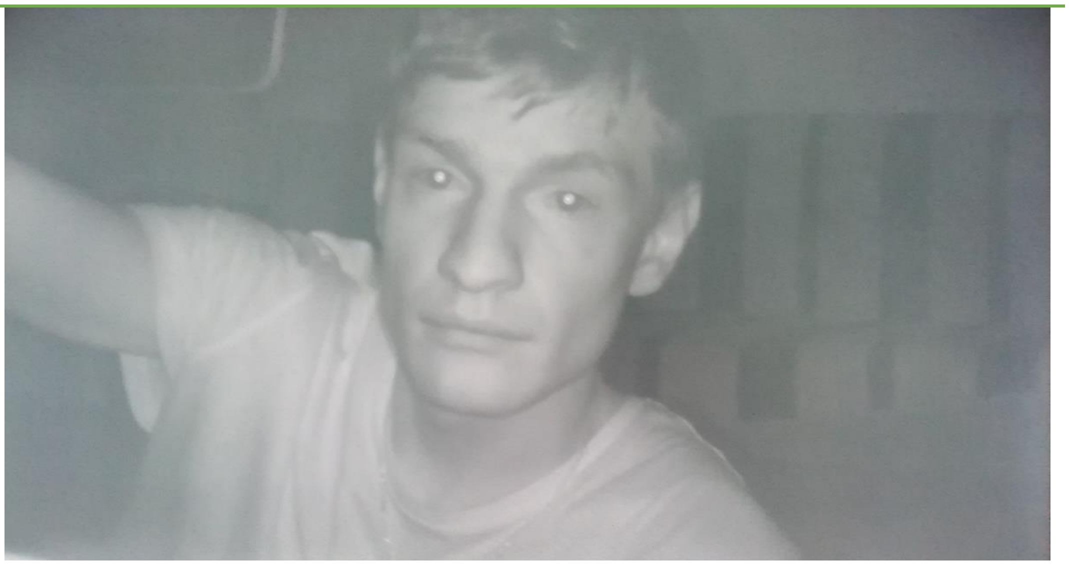
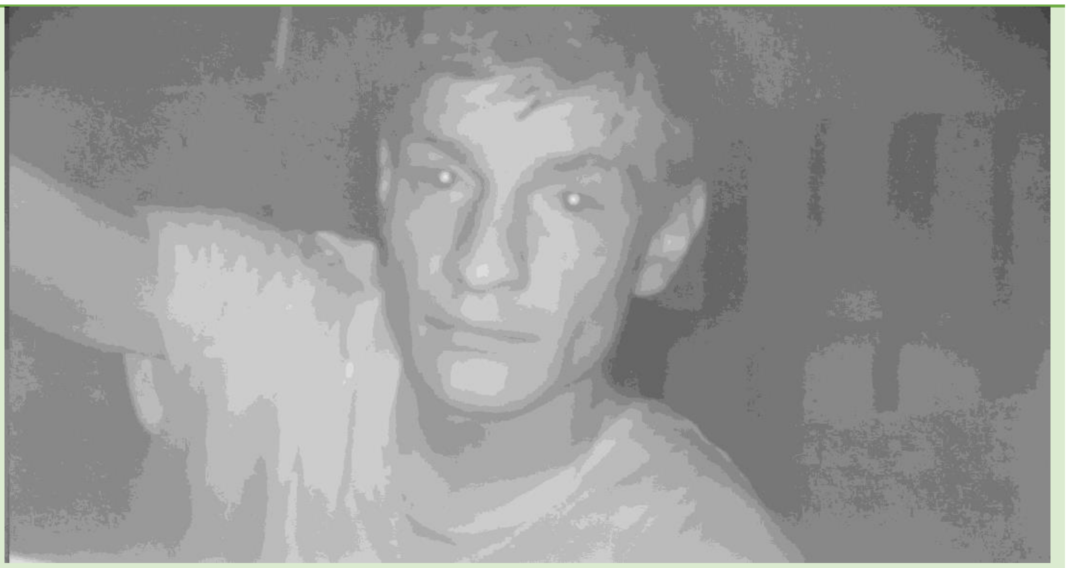


Figure 19, Dead pixel correction timing diagram

This module works only for 3x3 images because the center is always compared. To achieve 4 pixels in a cycle the module must be instantiated multiple times.

Input	
Output: **No blur** **Reduced noise** **Needs threshold**	

Laplace filter

Applies 3x3 convolution and the result image is a map of the edges in the image.

Laplace filter internal design

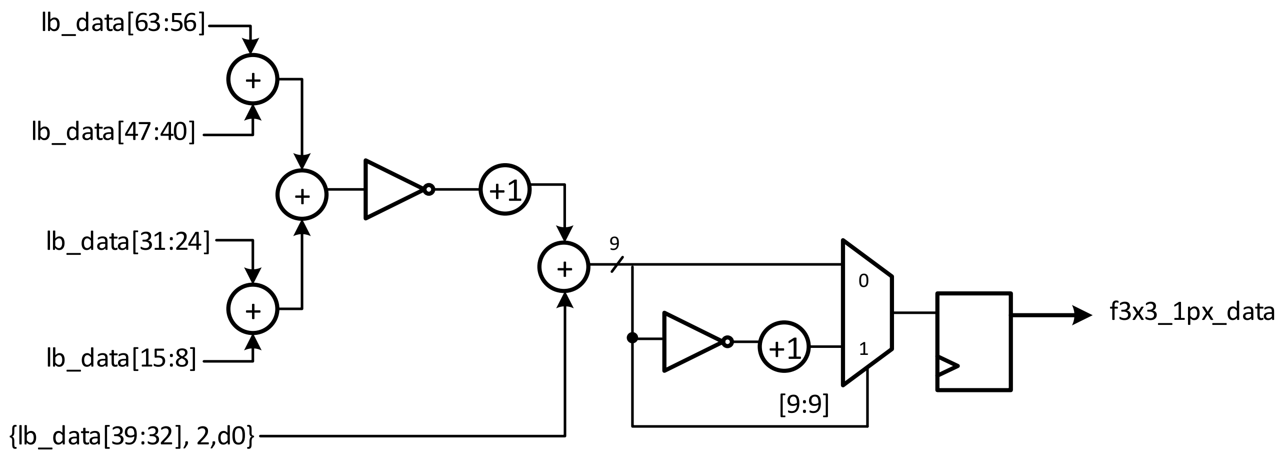


Figure 20, laplace_filter3x3 internal design

The output is computed using the formula below:

$$P_{out} = |-(P_{01} + P_{11} + P_{12} + P_{21}) + 4P_{11}|$$

The convolution is computed and then the sign bit is wired to the mux selection to set the output to the absolute value of the convolutions result.

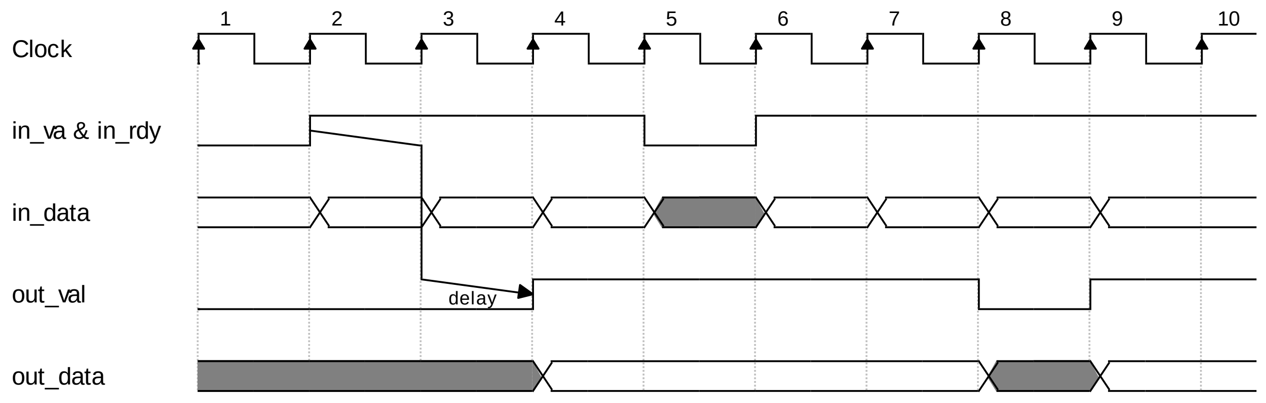
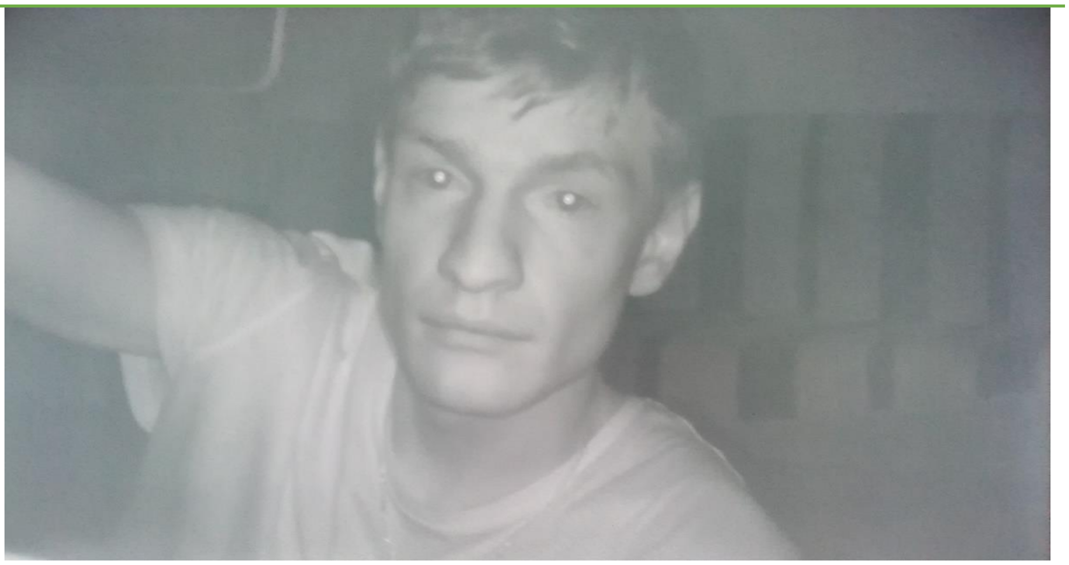
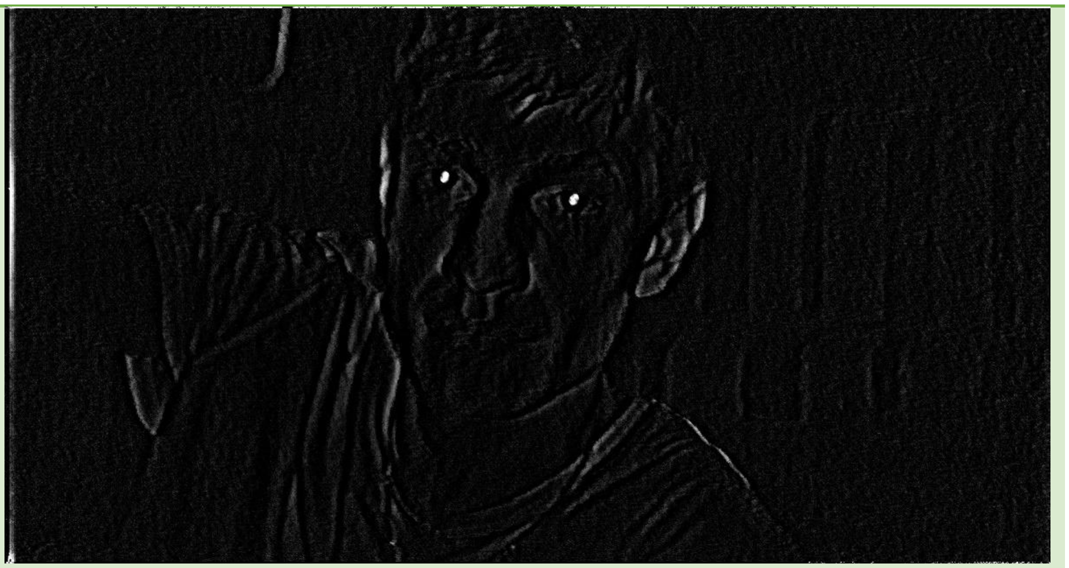


Figure 21 Laplace Filter timing diagram

To obtain the image for three channels the line buffer input output is divided into separate channels and feed intro three instances, the outputs are then concatenated to form an RGB image.

Input	
Output: **Edges only**	

Sharpening filter

The **sharp_filter** module applies a sharpening filter to an input grayscale image, calculated as the convolutions of the pixels in a 3x3 mask.

Sharpening filter internal design

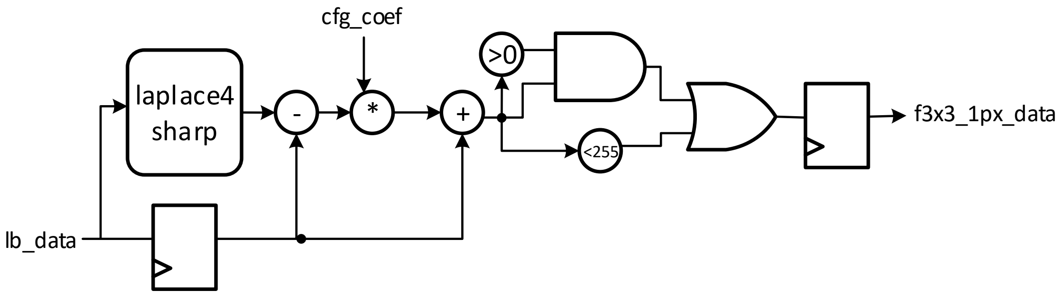


Figure 22 sharp_filter3x3 internal design

The module uses the mean filters output and uses the 4 instances of the circuit shown above. What calculates the output as shown in the formula, where the blurred pixel is the mean filter output.

$$P_{out}(x,y) = P_{in}(x,y) + coef * (P_{in}(x,y) - \ P_M(x,y))$$

$$P_M(x,y) = \frac{1}{8}\sum_{i=-1}^{1}\sum_{j=-1}^{1} P_{in}(x+i, y+j) * M(x+i, y+j)\,;\ M = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ -0 & -1 & 0 \end{bmatrix}$$

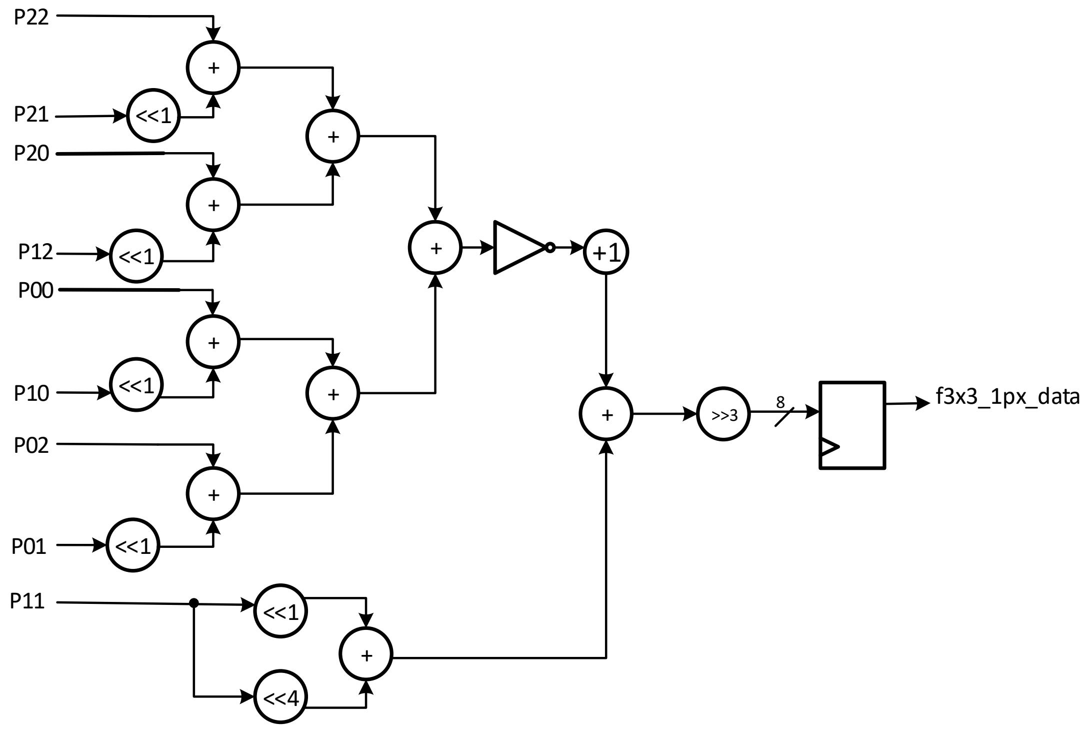


Figure 23, laplace4sharp internal design

Calculates the 3x3 convolution with kernel M.

First the borders are added together and then converted to twos complement. The center pixel is multiplied by 20 the two results are added together and then divided by 8. The module has an initial latency of 3 clock cycles.

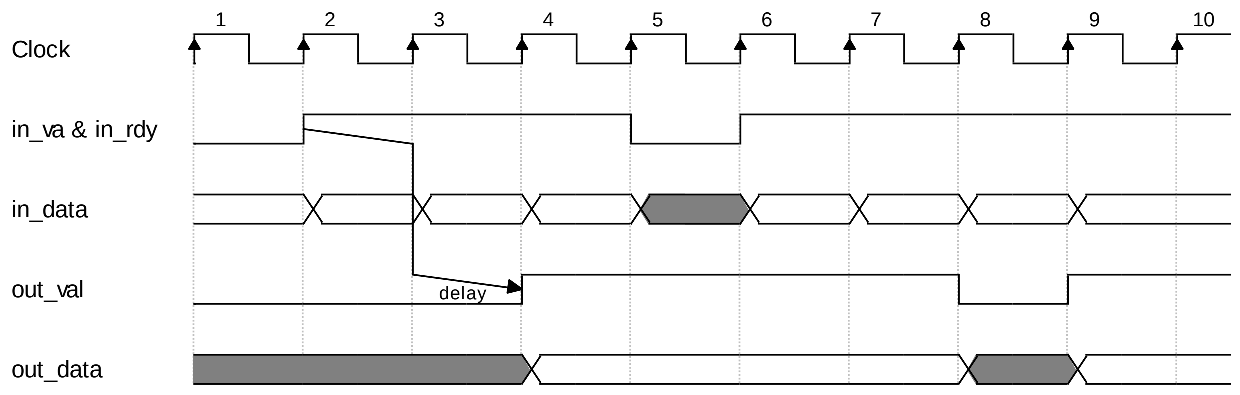
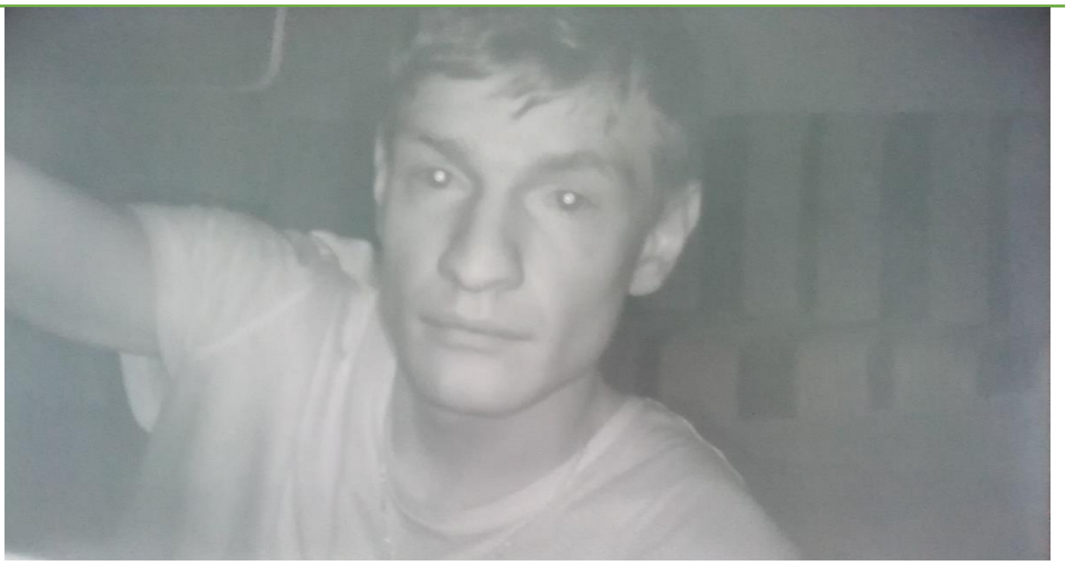
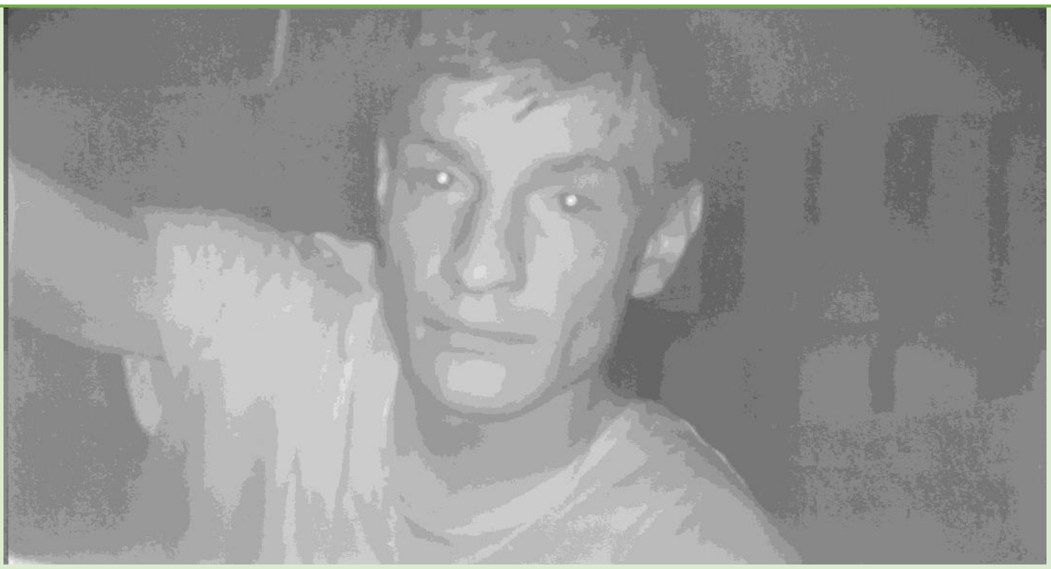


Figure 24, Sharpening filter timing diagram

Input



Output: **More details, the curves of the face are more visible**	

Median filter module

Median filter interfaces

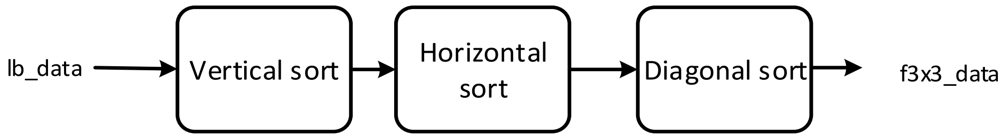


Figure 25, median_filter block diagram

Median filter internal design

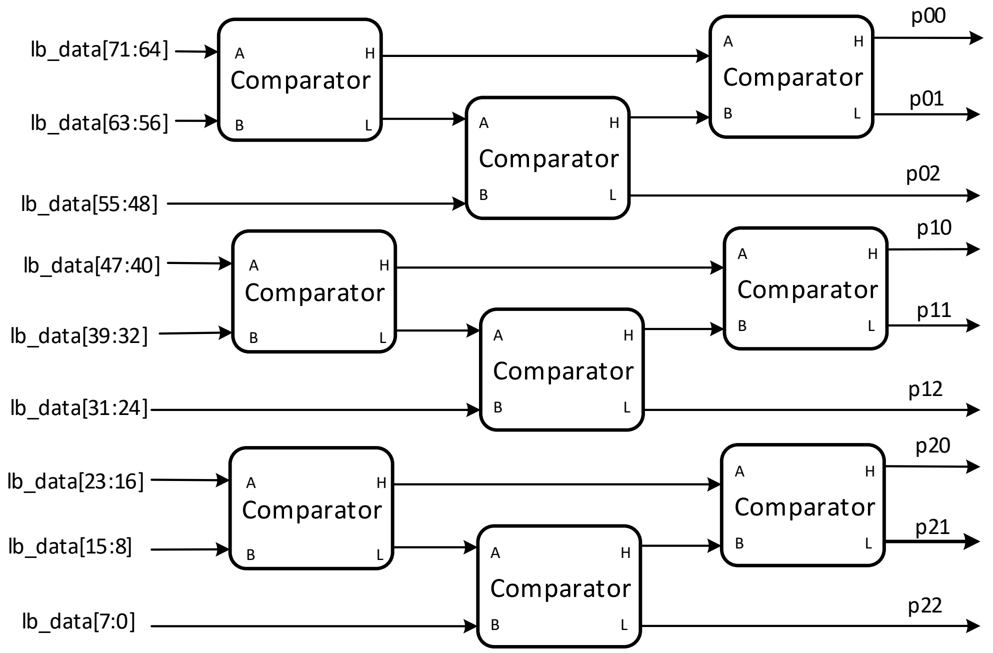


Figure 26 median_filter, vertical sorting

The filter sorts each column in the 3x3 segment, the output of this stage is a 3x3 matrix that will be further sorted. The comparators have two outputs for the high and low value of the comparison.

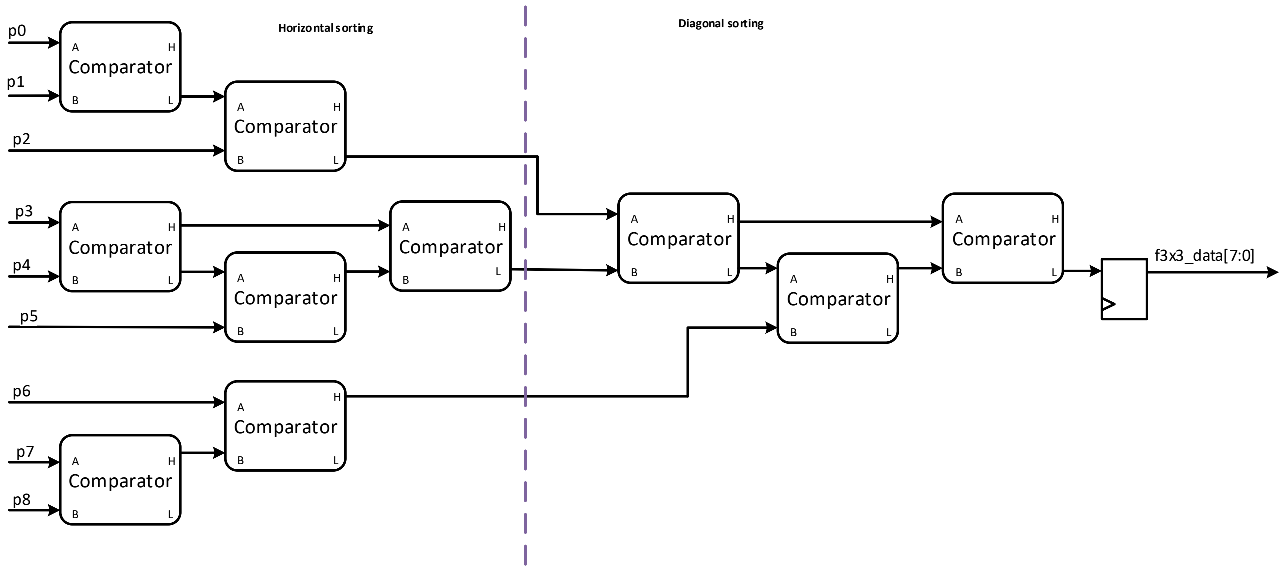


Figure 27 median_filter, horizontal, diagonal sorting and output generation

The vertically ordered 3x3 matrix is given as input, the first comparison stage orders the data horizontally and the last sorts the diagonal, the middle value of matrix is the median.

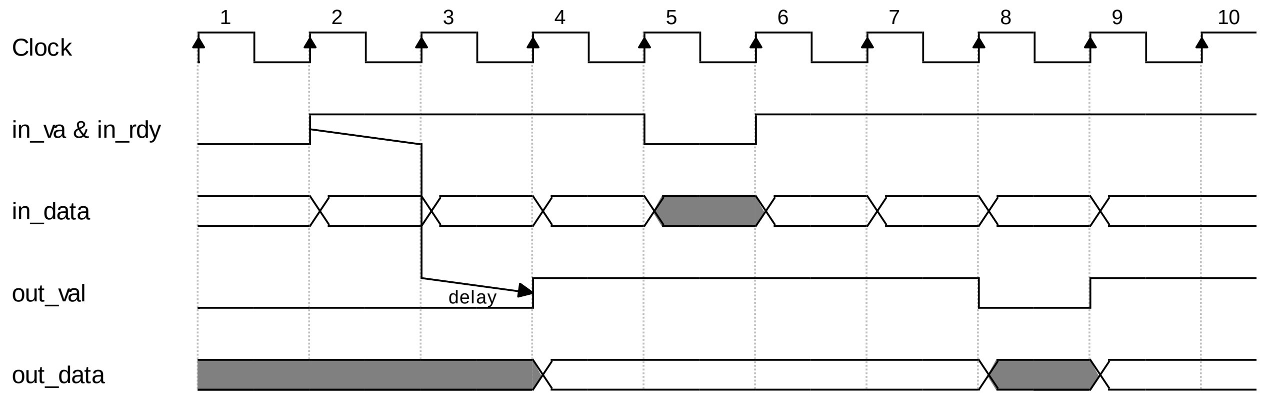
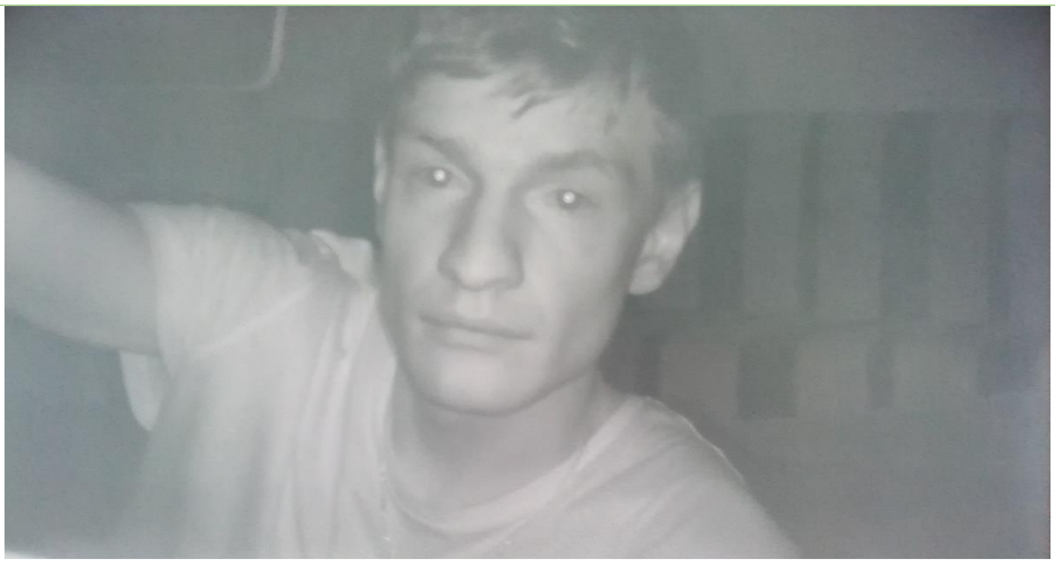
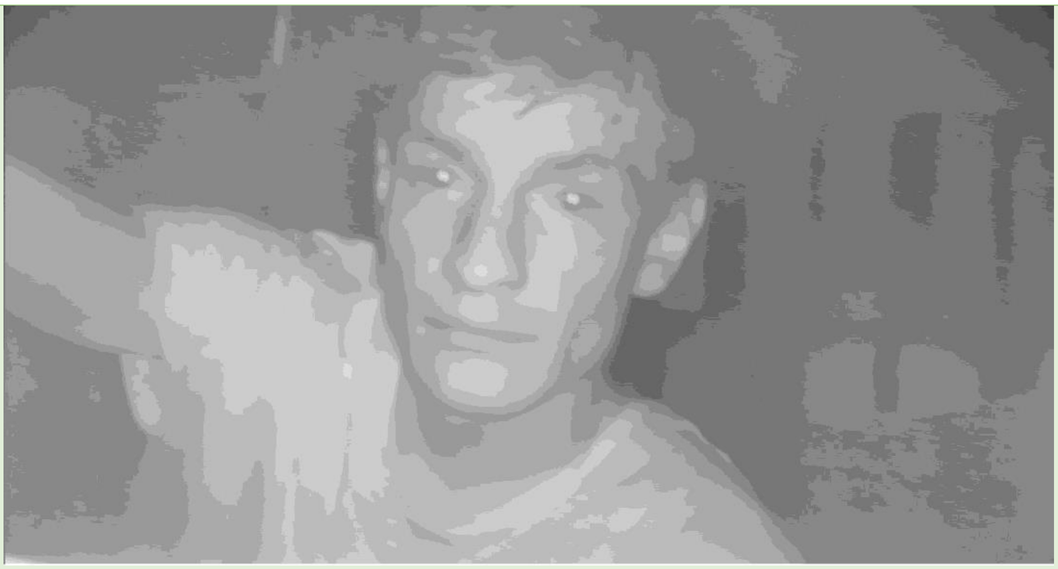


Figure 28 median_filter timing diagram

Input	

Output: **Blurred**	

Selector module

Controls the flow of how each processing units gets its input data.

Selector module interfaces

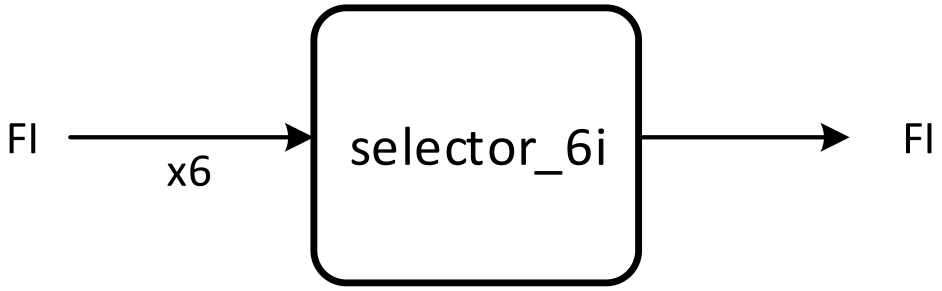


Figure 29 Selector module interfaces

Selector module internal design

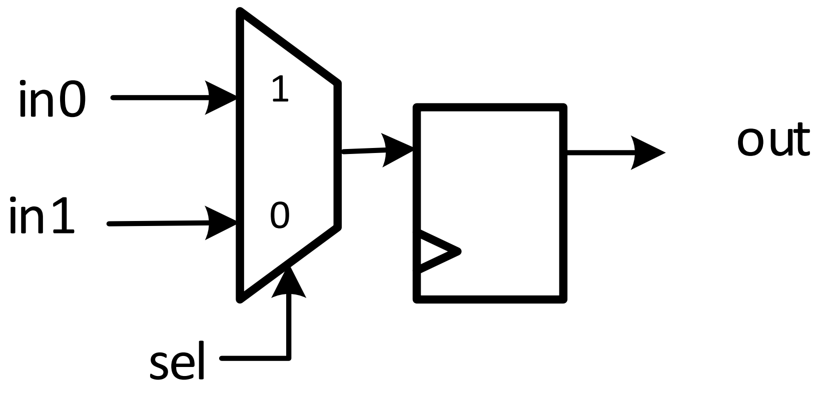


Figure 30, selector_2i module internal design

Each selector will receive a one hot code, according to that input the output will be frame input for what the select has a 1 on the corresponding bit. Ready signal will be output in the same manner.

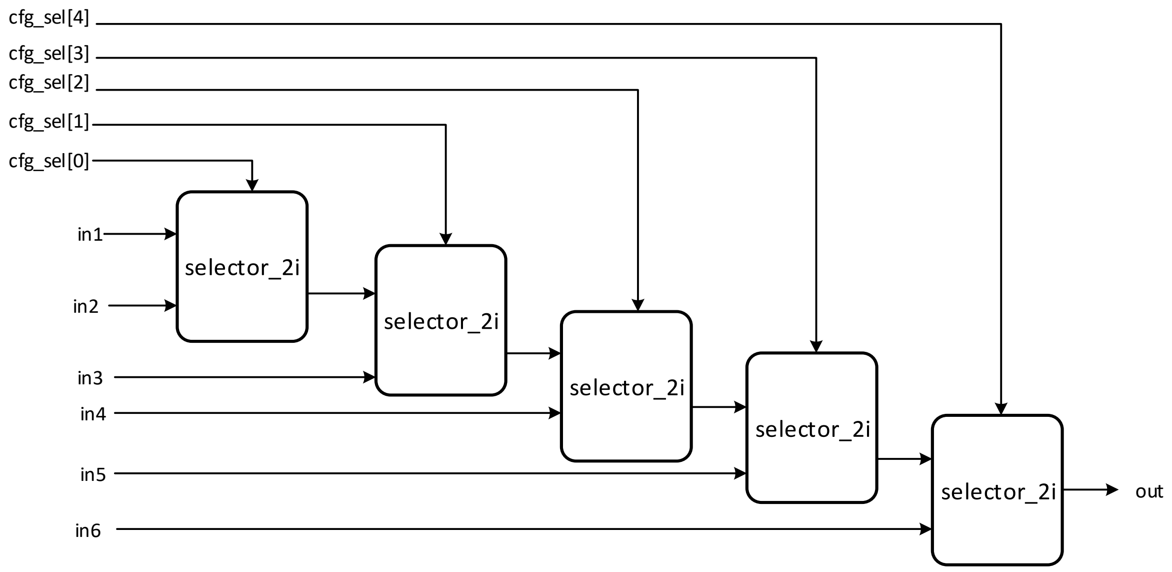


Figure 31, selector_6i architecture

AXI2FRAME

The AXI2FRAME reads 3 maps through AXI, stores them in FIFO memories, then these maps are recombined to a single channel data and interrupts are generated to signal any AXI error, read done.

Internal design and flow

The AXI2FRAME has 3 different submodules to read 3 maps on **AXI** and store them in FIFOs, to read the FIFOs and generate output data on **FI** and to generate interrupts. See the diagram below for details.

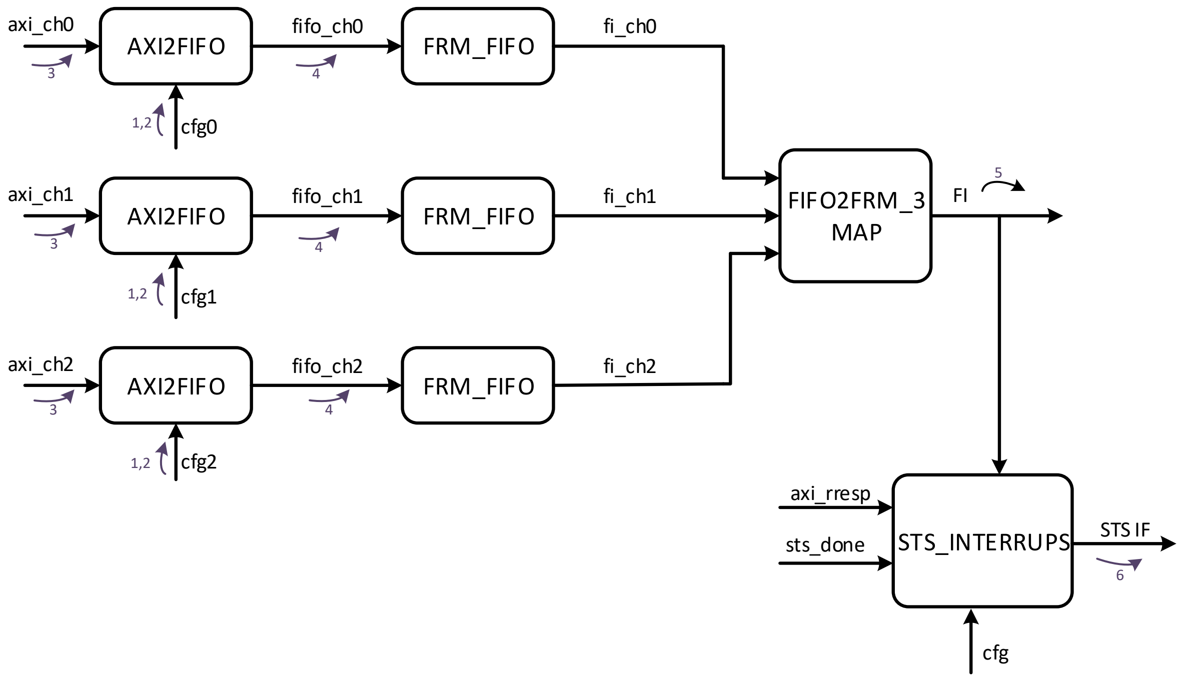


Figure 32, axi2frm internal design

Flow description

1) All AXI2FRM modules are configured
2) Module is enabled
3) The configured number of Image maps are read from the DDR
4) Every channel data is stored in the corresponding FIFO
5) The FIFOs are read and data is sent out on FI
6) When the whole image was read, interrupt signal is generated

See in the diagram below for the interfaces:

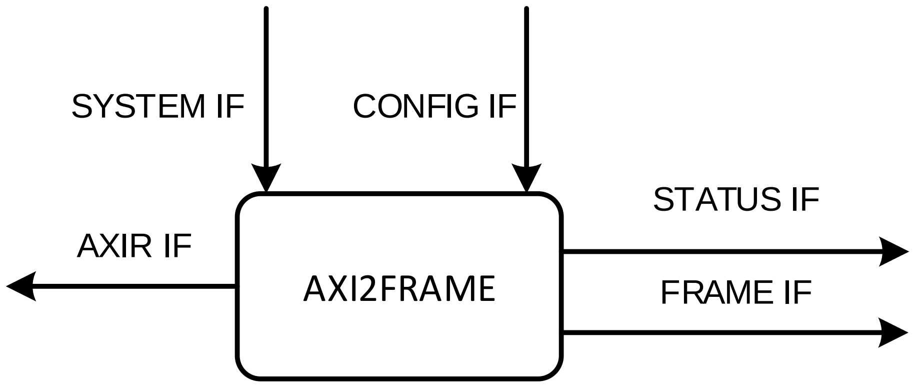


Figure 33, axi2frm interfaces

System IF

Signal name	Functionality	I/O	Width [bits]
clk	Clock	I	1
rst_n	Asynchronous system reset active low	I	1

Configuration interface

Signal name	Functionality	I/O	Width [bits]
cfg_blk_en	Module enable	I	1
cfg_img_width	Image width	I	11
cfg_img_height	Image height	I	11
cfg_stride	The address distance between the first address of successive “horizontal” reads		11
cfg_map0_ba	Channel 0 base address	I	32
cfg_map1_ba	Channel 1 base address	I	32
cfg_map2_ba	Channel 2 base address	I	32
cfg_map0_en	Enable map0 read	I	1
cfg_map1_en	Enable map1 read	I	1
cfg_map2_en	Enable map2 read	I	1
cfg_max_burst_length	Maximum burst length	I	8
cfg_reverse_byte	1 if input in Big Endian, 0 if data is in Small Endian	I	1
cfg_int_ack	Interrupt acknowledge	I	1

FRAME interface

Signal name	Functionality	I/O	Width [bits]
frm_rdy	The target is ready to receive the data	I	1
frm_val	Data valid	O	1
frm_data	Data	O	24
frm_sof	Start of Frame	O	1
frm_eof	End of Frame	O	1
frm_sol	Start of Line	O	1
frm_eol	End of Line	O	1

Output status interface

Signal name	Functionality	I/O	Width [bits]
sts_axi_error	AXI transaction error	O	1
sts_idle	Module idle	O	1
sts_read_done	Read done	O	1
sts_frm_int	Frame interrupt	O	1

AXI READ interface

Address channel

Signal name	Functionality	I/O	Width [bits]
araddr	Address	O	32

arlen	Burst length	O	8
arsize	Burst size	O	3
arburst	Burst type	O	2
arvalid	Read address valid	O	1
arready	Ready to receive address	I	1

Read data channel

Signal name	Functionality	I/O	Width [bits]
rdata	Data	I	AXI_BUS_SIZE
rlast	Read last	I	1
rvalid	Read valid	I	1
rready	Ready to receive read data	O	1
rresp	AXI response	I	2

Input parameters

i. AXI_BUS_SIZE – AXI bus size (only 64)

AXI IF features

[Ft 1] The module uses three different AXI read channels

[Ft 2] Independent base address configuration for all 3 AXI channels

[Ft 3] Uses AXI4 reduced interface

[Ft 4] ARBUSRT is stuck at 2'd1 – only incremental supported

[Ft 5] ARSIZE must be 3 (64-bit data width)

Functionality Features

[Ft 12] The AXI2FRAME reads the maps from the DDR that are enabled, and sets the output as shown at [Ft 5]. The maps that are not enable will be 0 in the output data.

[Ft 13] When done reading the image status idle signal will be activated and done pulse will be generated indicating the moment when the last valid data was transferred on **FI**.

[Ft 14] Input data format for AXI on 64 bits:

Data	Bytes	Bytes
Data [63:56]	BYTE7	BYTE0
Data [55:48]	BYTE6	BYTE1
Data [47:40]	BYTE5	BYTE2
Data [39:32]	BYTE4	BYTE3
Data [31:24]	BYTE3	BYTE4
Data [23:16]	BYTE2	BYTE5
Data [15:8]	BYTE1	BYTE6
Data [7:0]	BYTE0	BYTE7

[Ft 15] Output data format:

Data	Channel 2	Channel 1	Channel 0
Data [23:0]	Map1	Map1	Map0

Limitations

[Lim 1] No pending request

[Lim 2] Max image size 2047x2047

[Lim 3] Min image size is 8x8 pixels, when AXI Data Width is 64

Behavior at Enable/Disable System Enable

- If enable ''**cfg_blk_en**" is 0 the memory will not be read and all signals in the FI will be low.
- When enable "**cfg_blk_en** " is 1 it will respect the functionality described at 5.2
- The module must be enabled only after the proper configuration
- The module configuration should not be changed while enable
- The module configurations should be changed during the interrupt phase
- The module will start and will reset at posedge enable
- When module is done reading the image an interrupt will be sent
- Module cannot be disabled while status idle is not active

Module Description

AXI2FIFO Module

This module reads a map through AXI read interface and pushes data into a FIFO memory.

AXI2FIFO module interfaces

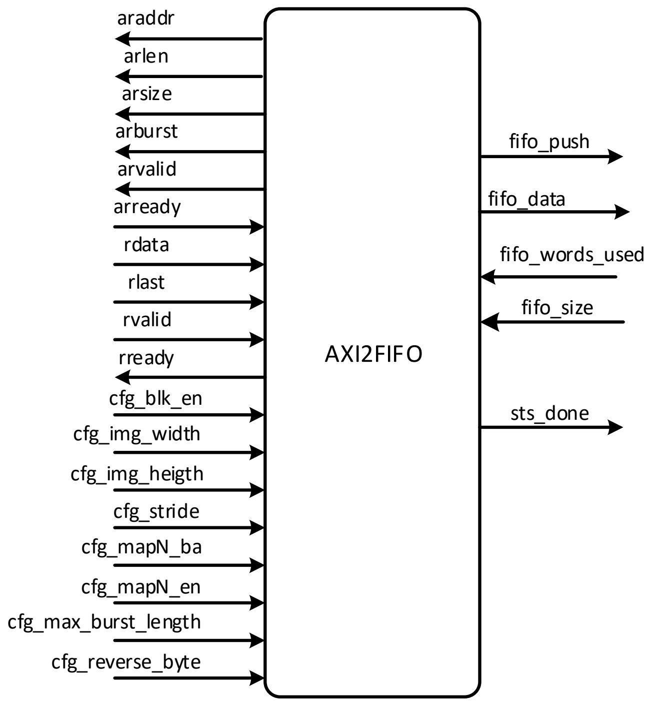


Figure 34, axi2fifo module interface

Module Parameters

Parameter	Description
AXI_BUS_SIZE	AXI bus size 64

AXI2FIFO module internal design

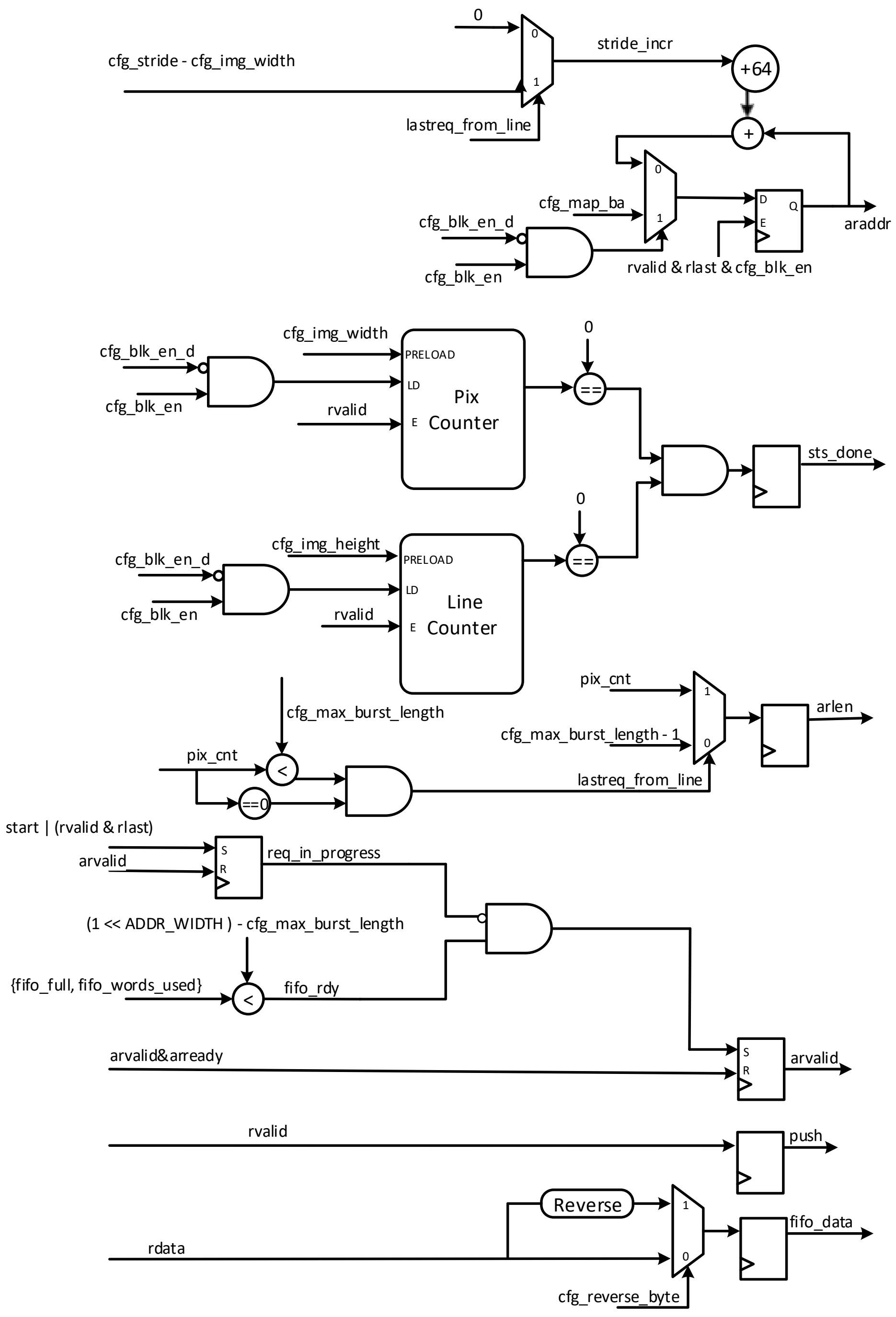


Figure 35, axi2fifo internal design

Addresses are generated by adding a constant increment to the base address, the constant is the (burst length * burst size). The increment is calculated as burst length multiplied by the bus size.

Two counters keep track of the pixel read from the memory, after each valid data a burst length will be decremented from the pixel counter, when both line and pixel counter are 0 the done flag will be activated.

The module has no pend request, the address is valid while there is data to be read and the to read, after the last data has arrived the address will increment.

Data will be pushed into the FIFO if it is ready (there is enough space in the FIFO for a new burst) and the data is valid.

When the data is in big endian the output will be converted to small endian, otherwise the output is same as it is read from the DDR.

Module is reset on system reset or positive edge of enable.

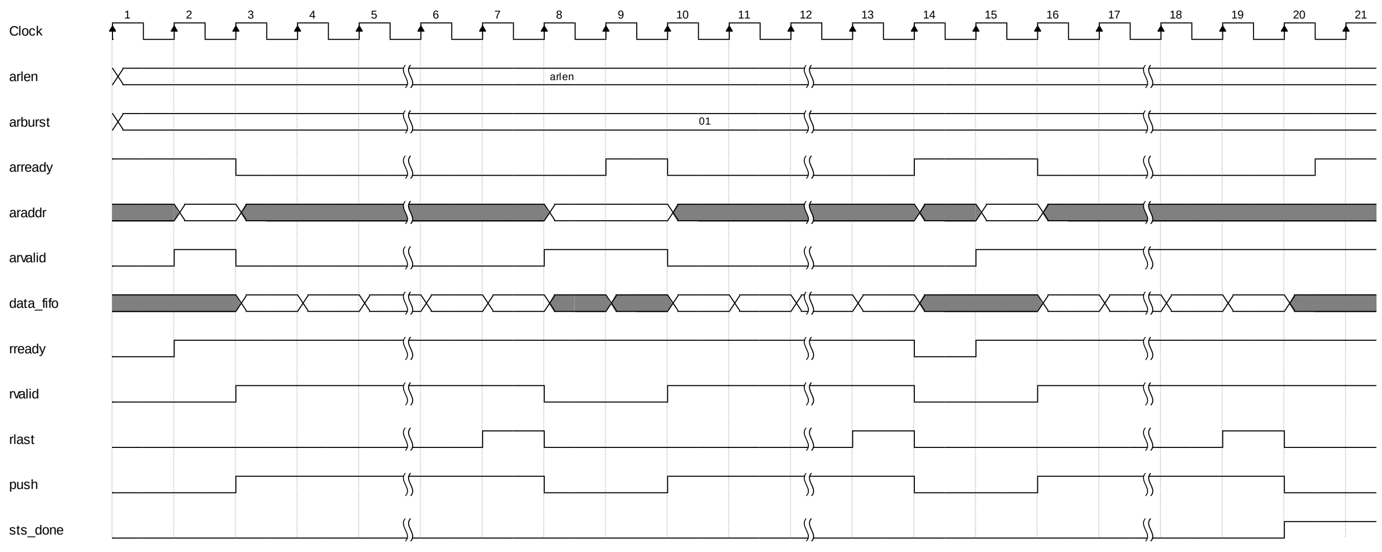


Figure 36, axi2fifo timing diagram

FIFO2FRM_3MAP Module

FIFO2FRM_3MAP module interface

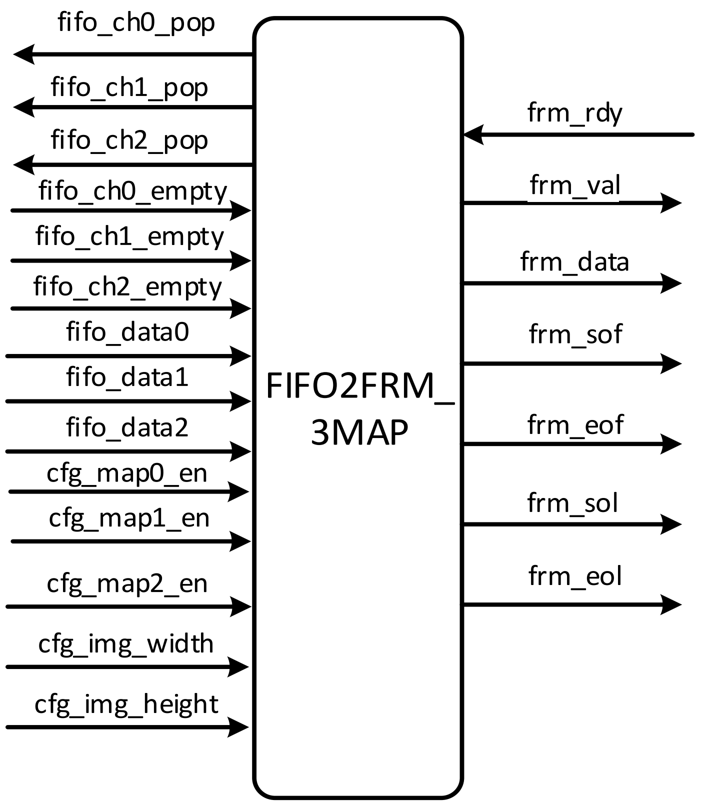


Figure 37,fifo2frm_3map module interface

Module parameters

Parameter	Description
FIFO_DATA_WIDTH	Input data width

FIFO2FRM_3MAP module internal design

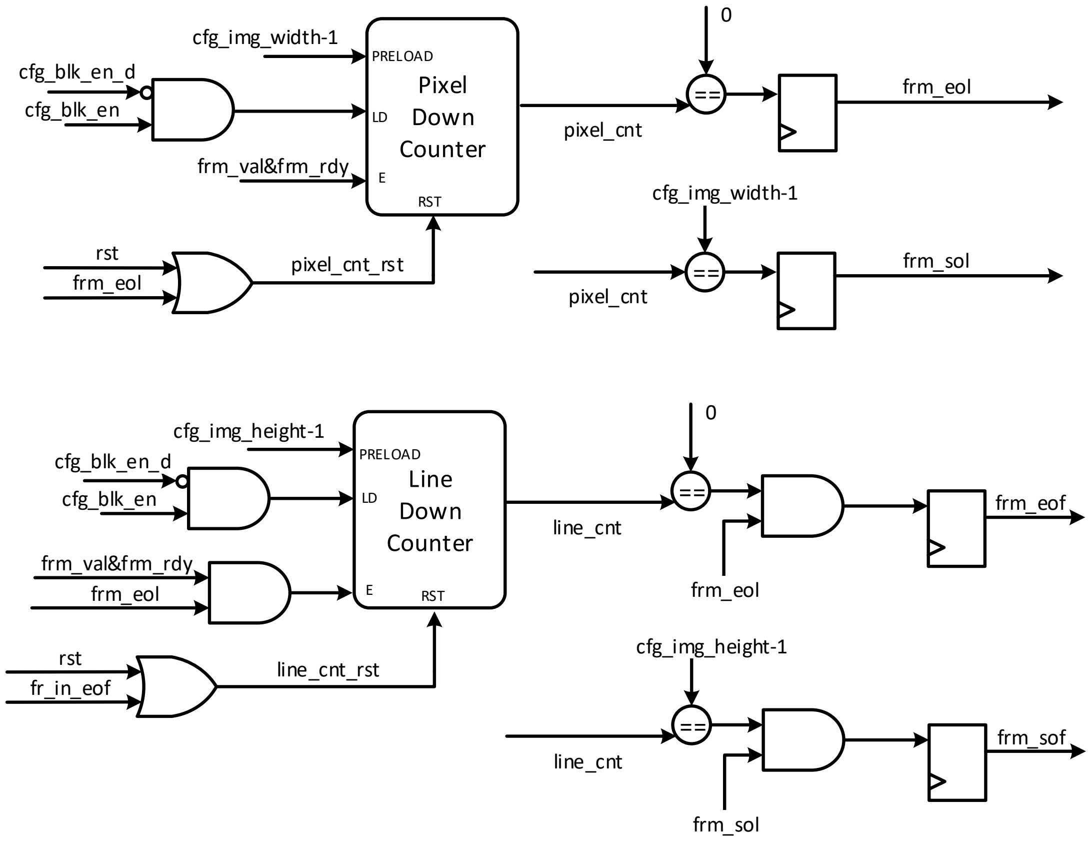


Figure 38, fifo2frm_3map internal design, control signals

Start of frame is generated and the posedge enable and reset after the first valid data is sent. Start of line is set at start or after end of line. End of line is set at the last pixel from the current line, and end of frame is set at the last pixel from the frame.

The pixel and line counters will reset at system reset or when end of line respective end of frame is active. Pixel counter in counting the number of pixels in a line, Line counter tracks the number of lines in the frame.

A “**fifo_chN_pop”** is sent if the current channel is enabled and is not empty, and all bytes are extracted from the burst.

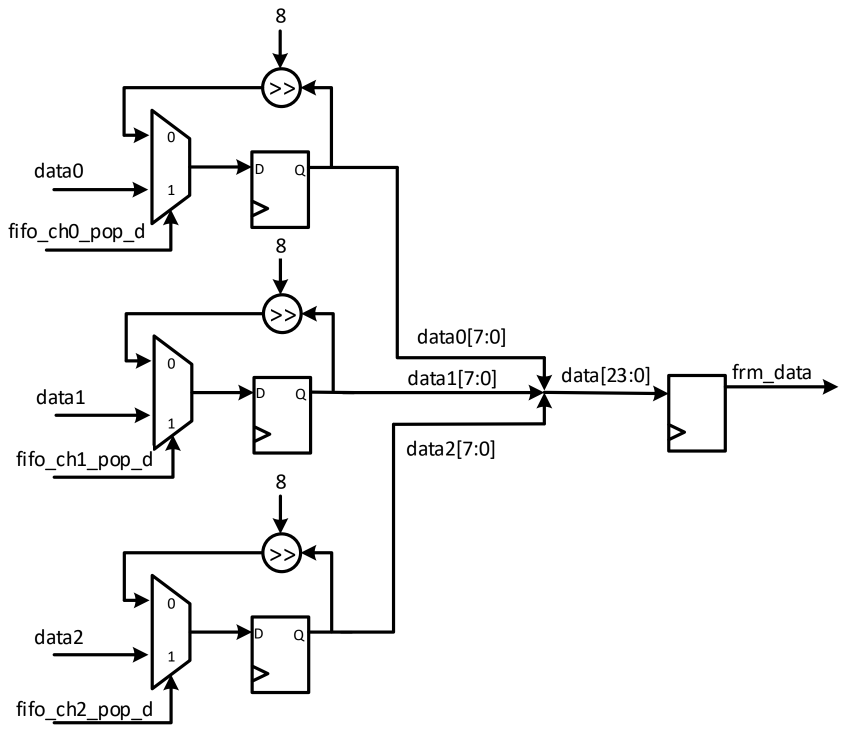


Figure 39, fifo2frm_3map internal design, data

A pixel represents 1 byte of the input data, to generate a 1 channel 24-bit data output the data from all 3 channels are shifted right by 1 byte as many times as many bytes it contains extracting the last 8 bits of the data and concatenating them.

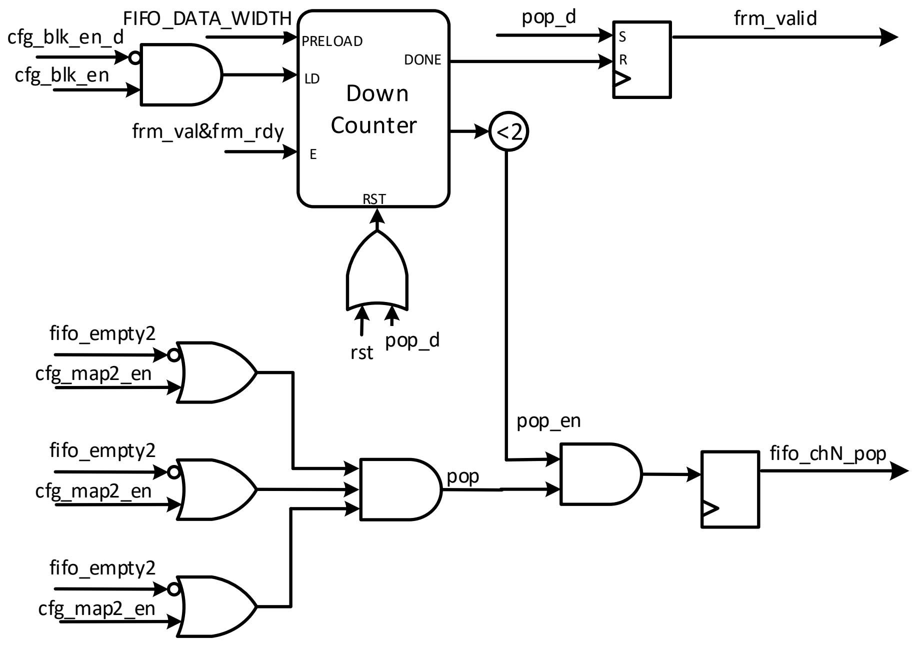


Figure 40, fifo2frm_3map internal design, pop and valid

Pop will be sent when the enabled FIFOs are not empty, and all data is extracted from a burst.

The data on the output is considered valid if the time between the two pop signals is exactly the number of bytes the data contains. If the counter finishes counting the number of shifting operations and if it exceeds the number of bytes in the input data valid will be deactivated.

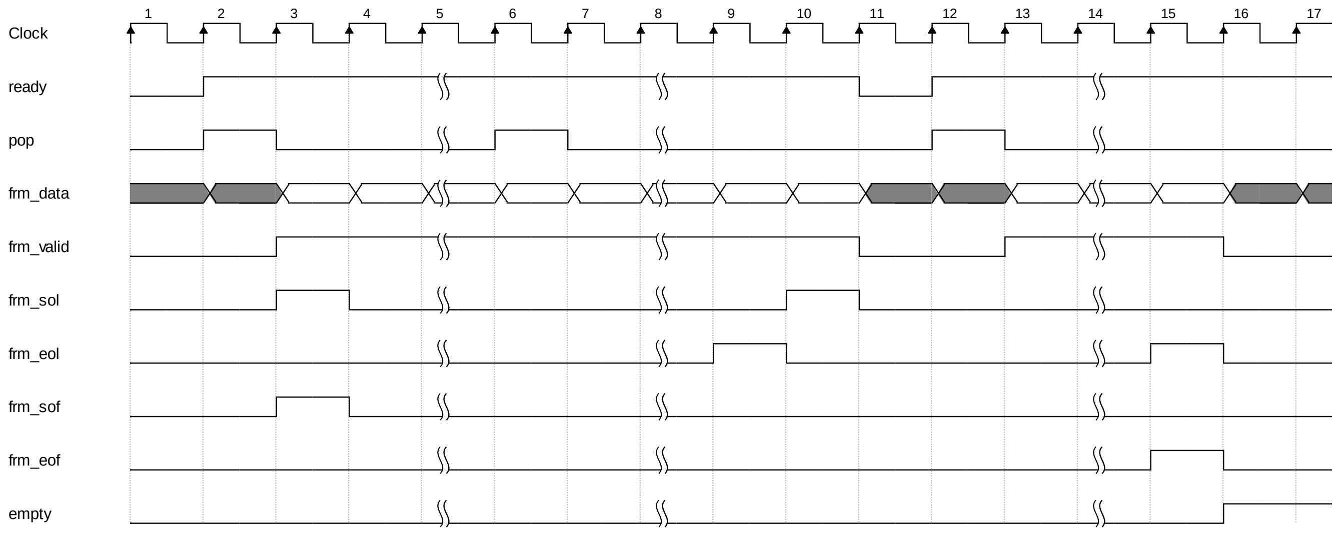


Figure 41, fifo2frm_3map timing diagram

STS_INTERRUPTS module

STS_INTERRUPTS module interfaces

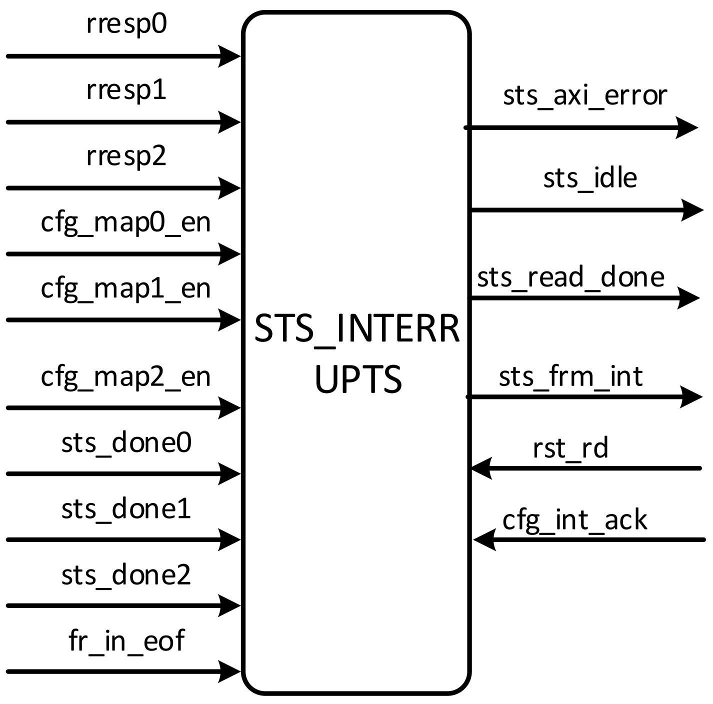


Figure 2.4, STS_INTERRUPTS module interfaces

Status module internal design

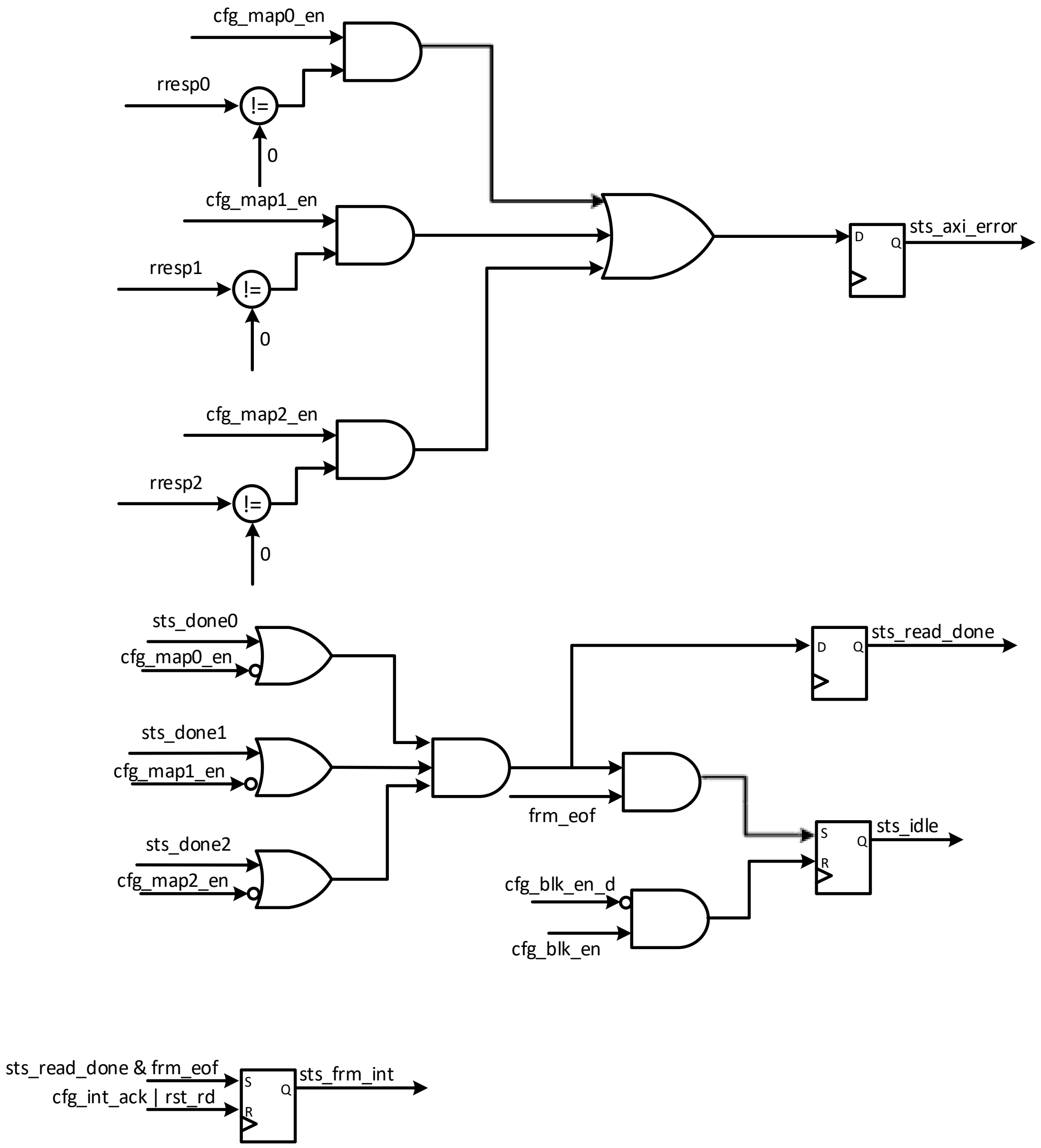


Figure 42, sts_interrupts internal design

The module indicates an error when the AXI response for one of the channels is not 0("OKAY").

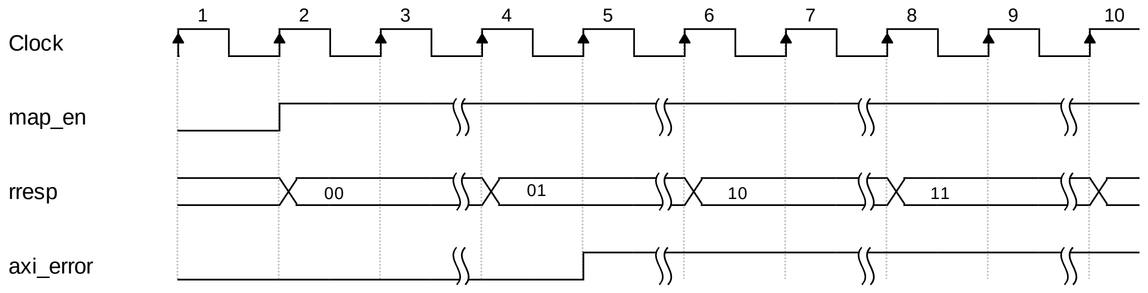


Figure 43, sts-interrupt module diagram for errors

The idle signal and read done is activated if the AXI2FIFO module for one of the channels is done reading and the FRAME module sent the last pixel (eof signal is active). Idle will be deactivate on reset or positive edge enable.

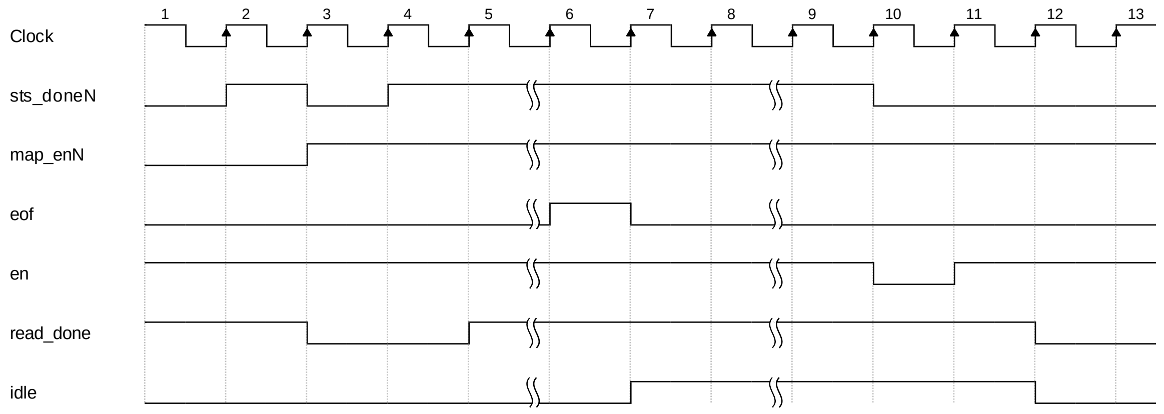


Figure 44, sts_interrupts idle signal timing diagram

Interrupt is generated after end of frame signal is received and will stay active until an acknowledge will arrive or it is reset automatically by the next module using the read reset signal. During the interrupt the module configurations can be changed.

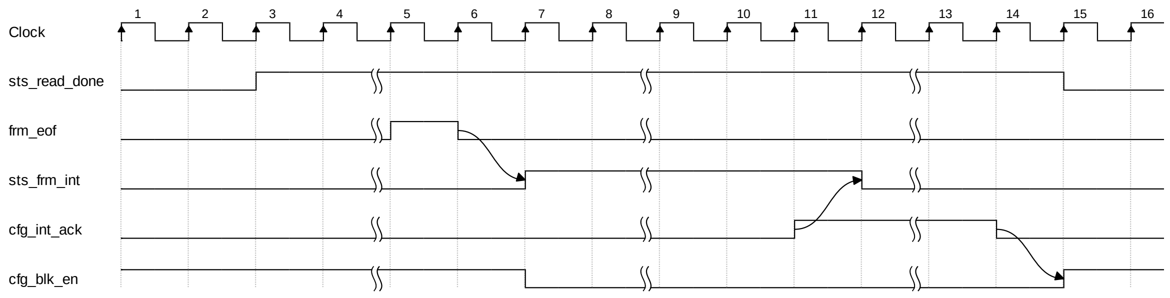


Figure 45, interrupt behavior

Read reset must be a pulse that will reset the all modules it generates a posedge enable, so the module can reset automatically. If this pulse doesn't exist, the module will send only 1 frame at the preset configuration.

Software

Interrupt Handlers

In the software part the majority of the Pcam 5c demo software is kept. Additional features are added. A second VDMA instance to create a split screen, on one half of the monitor to see the original unprocessed image and on the other half the processed one.

The interrupt for this handler is generated by the ir_filters module, at each end of the processed frame.

To achieve this effect the DMAs are two circular buffers, in Vivado the frame number was increased to have a bigger gap between the read and write pointer. In that gap the with memcpy instruction the original image is copied and overwrites the half of the processed one. The copy action must be complete before the read pointer gets to that memory zone. To make sure the read pointer will not be faster than the copy action both read and write pointers where parked at the current frame until the overwriting is completed. At the same time the interrupt is disabled.

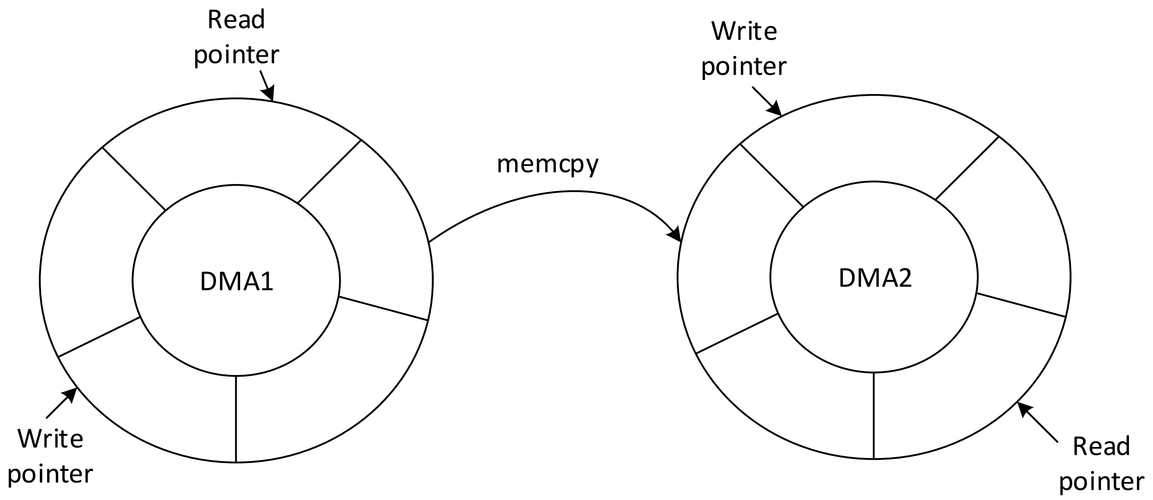


Figure 46, DMA memcpy working principle

The DMA1 from Figure 46, DMA memcpy working principle, writes the data from the camera and the read data is send to the image processing module. The other one, DMA2, stores the processed image and the read frame is sent to the display. The copy action must take place before DMA2's read pointer to get to the frame where the data is being copied.

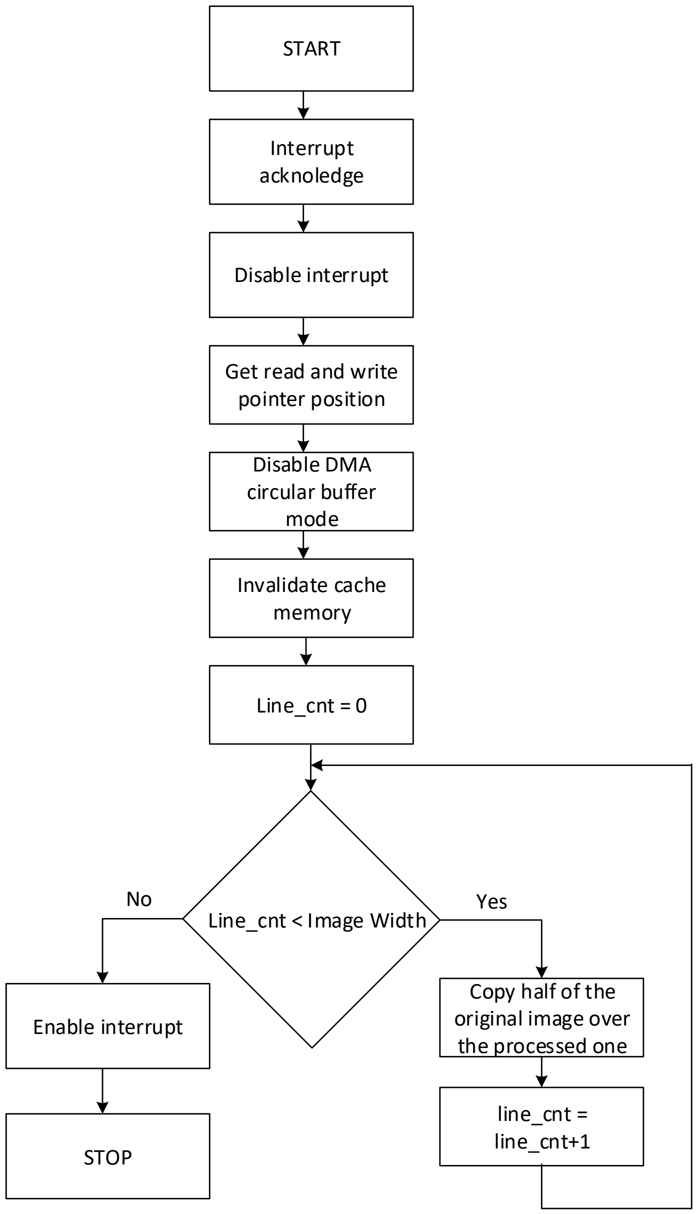


Figure 47, DMA interrupt handler algorithm

Besides the DMA interrupt handler there is a GPIO interrupt handler, whose purpose is to red the push buttons and the switches from the Zybo board and change the configuration of the module. There are some defined configurations in software that have some filter orders and numbers defined like having only one filter active, having a succession of two filters or more, these can be selected using the switches. Some filters have coefficients, the value of these can be changed using the buttons. SW3 will select between

Camera configuration

Int this project there is another camera used than in the Pcam5C demo an OV5647. The configuration part and the camera object instantiation were rewritten, and the a I2C driver was removed from the original project, a new one was added.
The cameras configuration was extracted from a Raspberry Pi with the help of a logic analyzer.

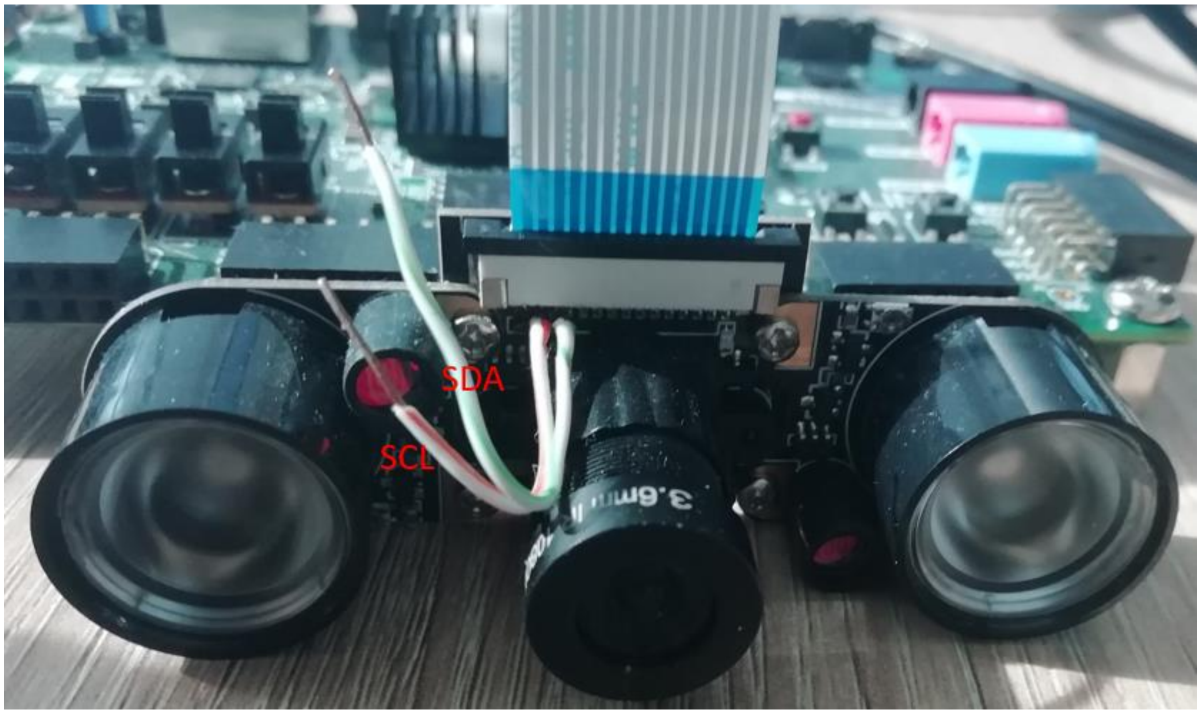


Figure 48, I2C pins on the camera

In the Raspberry Pi after enabling the camera using the *raspivid* command in the terminal the analyzer picked up a configuration, which the Seleae software decoded.

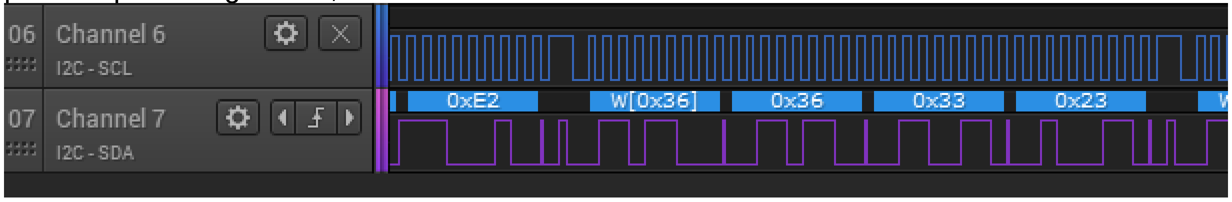


Figure 49, I2C decoded configuration

The extracted configuration was parsed and copied in the C code; the same segment was further sent from the Zybo board to start the camera.

Filter configuration

All configuration is made by writing the register bank via the APB interface. In Vivado the APB bridge connects the processing system to the bank.

```
void filter_cfg()
{
        Xil_Out32(APB_BASE_ADDR + CFG_IMG_WIDTH_ADDR, IMG_W);
        Xil_Out32(APB_BASE_ADDR + CFG_IMG_HEIGHT_ADDR, IMG_H);
        Xil_Out32(APB_BASE_ADDR + CFG_PIX_CORR_SEL_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_SHARP_SEL_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_SMOOTH_SEL_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_MEDIAN_SEL_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_LAPLACE_SEL_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_OUTPUT_SEL_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_PIX_CORR_THR_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_SHARP_COEF_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_TEST_MODE_EN_ADDR, 0);
}
```

The configuration presented above is the selection of each selector module. Now it is configured so that the input stream will go to the output without any processing.

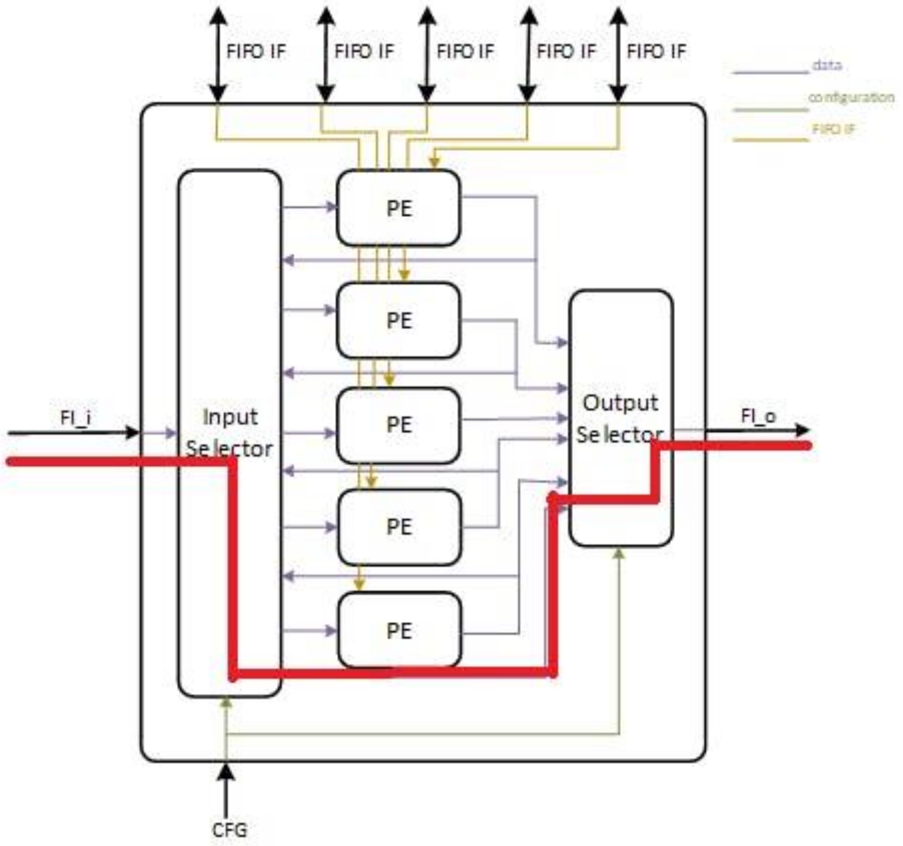


Figure 50, Configuration for transparent

```
void filter_cfg()
{
        Xil_Out32(APB_BASE_ADDR + CFG_IMG_WIDTH_ADDR, IMG_W);
        Xil_Out32(APB_BASE_ADDR + CFG_IMG_HEIGHT_ADDR, IMG_H);
        Xil_Out32(APB_BASE_ADDR + CFG_PIX_CORR_SEL_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_SHARP_SEL_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_SMOOTH_SEL_ADDR, SMOOTH_IN_CODE);
        Xil_Out32(APB_BASE_ADDR + CFG_MEDIAN_SEL_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_LAPLACE_SEL_ADDR, SMOOTH_IN_CODE);
        Xil_Out32(APB_BASE_ADDR + CFG_OUTPUT_SEL_ADDR, LAPLACE_IN_CODE);
        Xil_Out32(APB_BASE_ADDR + CFG_PIX_CORR_THR_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_SHARP_COEF_ADDR, 0);
        Xil_Out32(APB_BASE_ADDR + CFG_TEST_MODE_EN_ADDR, 0);
}
```

Putting the its own input to a processing element will be treated for it to gain the global input, to avoid eventual configuration errors, normally it would connect the input to the output and the module will not work.

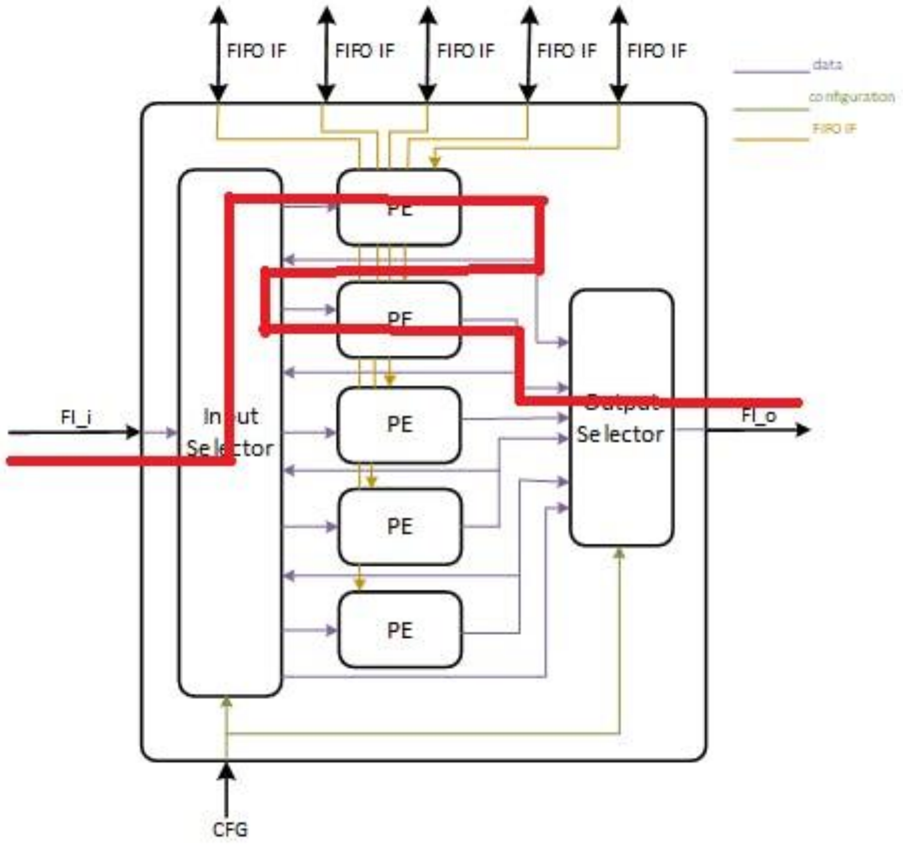


Figure 51, configuration for two filter succession

Discussion

Problems Encountered

Camera configuration, the I2c driver in the Pcam 5c demo was not working. At first it failed at the self-test when reading the registers containing the camera model number. It always read the same value even if no camera was connected. A new I2C driver was written.

Adding the module in the design flow. Before adding the second VDMA the processing module was connected to the GammaCorrection module. In this configuration the system wasn't functional. A bug was detected using chip scope where the GammaCorrection module does not support valid before ready handshake and the filtering module did not provide a ready signal and the system blocked. After resolving this bug when configuring the whole pipeline in software all components were reset and the filters not, this resulted in a shifted image on the display. To resolve this a self-reset was added to the line buffers, sot it will all reset, clear the FIFO and the internal registers at each start of frame signal.

Compatibility between AXI Stream and Frame interface. All modules in the Pcam 5c demo work on AXI Stream interface, but frame interface is needed to generate interrupts on end of frame. An interface converted was written to make create additional start of line and end of frame signal to the existing AXI Stream signals. On the other side there is no need for separate module, it only requires correct wiring.

Understanding the existing Pcam 5c demo to be able to modify it, both hardware and software side, integrating modules/IPs, rewriting the C source codes to match the current configuration settings, modify the drivers for the current setup.

Engineering Resources Used

- Vivado 2016.4 Design Suite
- Modelsim
- Seleae logic
- Notepad++

Marketability

The architecture of the project offers a flexibility, the algorithms used are simple, they are implemented just showcase the design. At the current state the project is a good for preprocessing images. Because of its extensibility some more complicated processes can be added like face detection, Eyegaze or any other neural network based algorithm.

Depending on the application only functionality can be added, for example in the medical industry is need for image enhancement and after what recognition algorithm, organ detection or mineral recognition both based on the reflected infrared light.

In automotive the principle usage of infrared imaging is its night vision capability and seeing eyes even through sun glasses, most common used for different type detection for the driver behavior to increase the safety.

Community Feedback

The commutity asked consisting of students and professors replied positive, the project is complex that had a lot a of effort put into it. Just by understanting a flow of an existing project and extending it both hardware and software side is challenging, it is always easier to do something from sratch.

Other big point is the interfacing of the Raspberry Pi camera. These products are widely spread in the market the Raspberry Pa has a lot of different cameras with different resolutions, color space, functionalities. This project showcases that it is possible to connect any commercial camera with CSI-2 standard connector to the Zybo board.

The architecture part is extensible the algorithm showcased are relatively simple ones used today in preprocessing tehniques for some beefy algorithm based on neural network for different type of detections mostly. But int is possible to replace or add an existing block to some more complicated algorithm.

References

Silverman, J. (1993, November). Signal-processing algorithms for display and enhancement of IR images. In *Infrared Technology XIX* (Vol. 2020, pp. 440-451). International Society for Optics and Photonics.

Marcello V., Piervincenzo R. (2011), Algorithms *for infrared image processing*, The University of Milan, www.politesi.polimi.it

Tanji E., Ookubo S. (January, 2015). *Infrared Camera Image Processing Technology and Examples of Applications*. NEC Technical Journal, Vol.9 No.1

Gonzalez, R. C., & Woods, R. E. (2002). Digital image processing [M]. *Publishing house of electronics industry*, *141*(7).

Vega-Rodríguez, M. A., Sánchez-Pérez, J. M., & Gómez-Pulido, J. A. (2002, July). An FPGA-based implementation for median filter meeting the real-time requirements of automated visual inspection systems. In *Proc. 10th Mediterranean Conf. Control and Automation*.

Sowmya, S., & Paily, R. (2011, February). FPGA implementation of image enhancement algorithms. In *2011 International Conference on Communications and Signal Processing* (pp. 584-588). IEEE.

Chandrashekar, M., Kumar, U. N., Reddy, K. S., & Raju, K. N. (2009). FPGA implementation of high-speed infrared image enhancement. *International Journal of Electronic Engineering Research*, *1*(3), 279-285.

İlk, H. G., Jane, O., & İlk, Ö. (2011). The effect of Laplacian filter in adaptive unsharp masking for infrared image enhancement. *Infrared Physics & Technology*, *54*(5), 427-438.

Jiang, L. J., Ng, E. Y. K., Yeo, A. C. B., Wu, S., Pan, F., Yau, W. Y., ... & Yang, Y. (2005). A perspective on medical infrared imaging. *Journal of medical engineering & technology*, *29*(6), 257-267.

Arm Holdings, AMBA AXI and ACE™ Protocol Specification, www.arm.com

Arm Holdings, AMBA 4 AXI4-Stream Protocol, www.arm.com

Appendix A: Name of Source Code Files HDL

```
// Project     : ir_filters
// Module Name : axi_stream2Frame
// Author      : Szilard Hegedus
// Created     : 01/21/2019
//---------------------------------------------------------------------------------
------------------
// Description : Converts AXI4 Stream interface to Frame interface
//---------------------------------------------------------------------------------
------------------
// Modification history :
// 11/15/2018 (SH): Initial version
//---------------------------------------------------------------------------------
------------------


module axi_stream2frame#(
  parameter DATA_WIDTH = 24
)(
  input                         clk                     , // Syste clock
  input                         rst_n                    , // Asynchronous reset active
low
//-------------------------- Configuration interface ------------------------------
---------------
  input  [11:0]                 cfg_img_w               , // Image width
  input  [11:0]                 cfg_img_h               , // Image width
//-------------------------- AXI-Stream interface ---------------------------------
---------------
  input                         m_axi_stream_tuser      , // Start of frame
  input                         m_axi_stream_tvalid     , // Slave has valid data to be
transferred
  input                         m_axi_stream_tlast      , // End of frame
  input     [DATA_WIDTH-1:0]  m_axi_stream_tdata     , // Data transferred from slave
to master
  output                        m_axi_stream_tready     , // Master is ready to receive
the data
// ----------------------------- Frame Interface ----------------------------------
---------------
  output reg                     s_frm_val               , // Master has valid data to
be transferred
  input                          s_frm_rdy               , // Slave is ready to receive
the data
  output reg [DATA_WIDTH-1:0] s_frm_data           , // Data transferred from master
to slave
  output reg                    s_frm_sof               , // Start of Frame
  output reg                    s_frm_eof               , // End of Frame
  output reg                    s_frm_sol               , // Start of Line
  output reg                    s_frm_eol                 // End of Line
);

reg [11:0] pix_cnt ;
reg [11:0] line_cnt;
```

```
wire invalrdy;
wire outvalrdy;

assign invalrdy = m_axi_stream_tvalid & m_axi_stream_tready;
assign outvalrdy = s_frm_rdy & s_frm_val;

assign m_axi_stream_tready = s_frm_rdy;

always@(posedge clk or negedge rst_n)
if(~rst_n                         ) pix_cnt <= 11'd0         ; else
if(m_axi_stream_tuser & invalrdy ) pix_cnt <= 11'd0         ; else
if(m_axi_stream_tlast & invalrdy ) pix_cnt <= 11'd0         ; else
if(invalrdy                       ) pix_cnt <= pix_cnt + 1'd1;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) line_cnt <= 11'd0          ; else
if(m_axi_stream_tuser & invalrdy) line_cnt <= 11'd0          ; else
if(m_axi_stream_tlast & invalrdy) line_cnt <= line_cnt + 1'd1;


always@(posedge clk or negedge rst_n)
if(~rst_n                                ) s_frm_sol <= 1'b0; else
if(outvalrdy & s_frm_sol                 ) s_frm_sol <= 1'b0; else
if(m_axi_stream_tuser & invalrdy         ) s_frm_sol <= 1'b1; else
if(outvalrdy & s_frm_eol & (~s_frm_eof)) s_frm_sol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                                                              ) s_frm_eof <=
1'b0; else
if(outvalrdy & s_frm_eof                                               ) s_frm_eof <=
1'b0; else
if((line_cnt == (cfg_img_h - 1'd1))  & m_axi_stream_tlast & invalrdy) s_frm_eof <=
1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                            ) s_frm_val <= 1'b0; else
if(s_frm_rdy & (~m_axi_stream_tvalid)) s_frm_val <= 1'b0; else
if(invalrdy                          ) s_frm_val <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) s_frm_eol <= 1'b0; else
if(outvalrdy & s_frm_eol        ) s_frm_eol <= 1'b0; else
if(m_axi_stream_tlast & invalrdy) s_frm_eol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                         ) s_frm_sof <= 1'b0; else
if(outvalrdy & s_frm_sof         ) s_frm_sof <= 1'b0; else
if(m_axi_stream_tuser  & invalrdy) s_frm_sof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n  ) s_frm_data <= {(DATA_WIDTH){1'b0}}; else
if(invalrdy) s_frm_data <= m_axi_stream_tdata  ;

endmodule //axi_stream2Frame
```

```
//------------------------------------------------------------------------------------
------------------
// Project     : AXI2FRAME
// Module Name : AXI2FIFO
// Author      : SZILARD HEGEDUS
// Created     : 03/02/2018
//------------------------------------------------------------------------------------
------------------
// Description : Reads data on AXI interface and pushes it to FIFO
//------------------------------------------------------------------------------------
------------------
// Modification history :
// 03/02/2018 (SH): Initial version
//------------------------------------------------------------------------------------
------------------

module axi2fifo#(
  parameter ADDR_WIDTH = 32,
  parameter USEDW_BITS = 11
)(
// -------------------------------------------- System IF -------------------------
------------------
  input                         clk                 , // System clock
  input                          rst_n                , // Asynchronous reset active
low
// ----------------------------------------- AXI inputs ---------------------------
------------------
  input                         arready             , // Address ready
  input                  [63:0] rdata               , // Read data
  input                         rlast               , // Last data beat in transfer
  input                         rvalid              , // Valid data
  output reg  [ADDR_WIDTH-1:0] araddr              , // Address
  output reg              [7:0] arlen               , // Burst length
  output                  [1:0] arburst             , // Burst type
  output                    [2:0] arsize               , // Number of bytes in each
transfer
  output reg                    arvalid             , // Address valid
  output reg                    rready              , // Read ready
// ------------------------------------- Configuration Interface inputs ------------
-------------------
  input                         cfg_blk_en          , // Block enable
  input                  [15:0] cfg_img_width       , // Image width
  input                  [15:0] cfg_img_height      , // Image height
  input                 [15:0] cfg_stride          , // The address distance between
the first address of successive â€œhorizontalâ€? reads
  input       [ADDR_WIDTH-1:0] cfg_map_ba          , // Channel base address
  input                   [7:0] cfg_max_burst_length, // Maximum burst length
  input                         cfg_reverse_pixel   , // Data is is big/small endian
// ------------------------------------- FIFO inputs ------------------------------
-------------------
  input       [USEDW_BITS-1:0] fifo_words_used     , // Used word in FIFO
  input                         fifo_full           , // Full indicator
  input                         fifo_empty          , // Empty indicator
// ------------------------------------- FIFO outputs -----------------------------
-------------------
```

```
  output reg                    fifo_push          , // Push
  output reg             [63:0] fifo_data          , // Output data
// ------------------------------------ Status IF outputs --------------------------
------------------
  output reg                    sts_done             // Done interrupt
 );

reg [15:0] line_cnt     ;//count the number of lines
reg [15:0] pix_cnt      ;//count the number of pixels for a line
reg        cfg_blk_en_d ;

wire               start            ; // Start at posedge enable
wire [15:0]        stride_incr      ; // Addres increment at last line
wire [USEDW_BITS:0]fifo_cnt         ; // Words in fifo
wire               lastreq_from_line; // Last request

reg                req_in_progress  ; // Request in progree
wire               fifo_rdy         ; // Fifo ready
wire               fifo_in_rst      ;

//data requested only when enough space available in fifo to store

assign  lastreq_from_line  =  (pix_cnt  <  cfg_max_burst_length)  &  (|pix_cnt)
; // Last request form line
assign fifo_cnt        = {fifo_full, fifo_words_used}                              ;
// Number of words in fifo
assign start           = cfg_blk_en & (~cfg_blk_en_d)                              ;
//Start at posedge enable
assign stride_incr      = lastreq_from_line ?  (cfg_stride - cfg_img_width) : 16'd0
; // Increment or jump stride positions
assign arsize           = 2'd3                                                     ;
// Size is bus 8 for AXI 64
assign arburst          = 2'd1                                                     ;
// Set burst to incremental
assign  fifo_rdy                     =  fifo_cnt  <  (({1'b1,{USEDW_BITS{1'b0}}})  -
cfg_max_burst_length); // Fifo ready if more than a burst space is vailable

assign fifo_in_rst       = fifo_empty & fifo_full;

always@(posedge clk or negedge rst_n)
if(~rst_n) rready <= 1'b0                       ; else
           rready <= cfg_blk_en & (~fifo_in_rst);

always@(posedge clk or negedge rst_n)
if(~rst_n                  ) req_in_progress <= 1'd0;else
if(start | (rvalid & rlast)) req_in_progress <= 1'd0;else // Reset on start or last
valid data in burst
if(arvalid                ) req_in_progress <= 1'd1;     // Set on first valid data
from burst

always@(posedge clk or negedge rst_n)
if(~rst_n                    ) sts_done <= 1'd0      ;else
if(start                     ) sts_done <= 1'd0      ;else // Reset done on start
if((~|pix_cnt) & (~|line_cnt)) sts_done <= cfg_blk_en;     // Set done when line and
pixel cnt are 0
```

```
always@(posedge clk or negedge rst_n)
if(~rst_n   ) pix_cnt <= 16'd0          ; else // Set register 0 on reset
if(start    ) pix_cnt <= cfg_img_width  ; else // Load preloaded value on poesedge
enable
if(~|pix_cnt) pix_cnt <= cfg_img_width  ; else // Load preloaded value when pixel
counter is 0
if(rvalid   ) pix_cnt <= pix_cnt - 16'd8;      // Decrement on each valid data

//Verify image read
always@(posedge clk or negedge rst_n)
if(~rst_n                    ) line_cnt <= 16'd0           ; else // Set register
0 on reset
if(start                     ) line_cnt <= cfg_img_height  ; else // Load preloaded
value
if(rvalid & (pix_cnt == 16'd8)) line_cnt <= line_cnt - 16'd1;        // Decrement
register on valid data when pix_cnt resets

// Address generator
always@(posedge clk or negedge rst_n)
if(~rst_n                                                      ) araddr <= 32'd0
; else // Set register 0 on reset
if(start                                                  ) araddr <= cfg_map_ba
; else // Load preloaded value
if(cfg_blk_en & rvalid & rlast & (~sts_done)) araddr <= (araddr + ({(arlen +
1'd1),3'd0})) + stride_incr;     // Increment address

always@(posedge clk or negedge rst_n)
if(~rst_n                ) arlen <= 8'd0                                        ;
else // Set register 0 on reset
if(start                                 ) arlen <= cfg_max_burst_length - 1'd1
; else // Load preloaded value on posedge enable
if(rlast & rvalid & cfg_blk_en) arlen <= lastreq_from_line ? pix_cnt :
(cfg_max_burst_length - 1'd1);       // Set lentgh to max burst size or remaining
pixels number on last request

//Delay enable
always@(posedge clk or negedge rst_n)
if(~rst_n) cfg_blk_en_d <= 1'b0      ;else
           cfg_blk_en_d <= cfg_blk_en;

//Output data
always@(posedge clk or negedge rst_n)
if(~rst_n) fifo_data <= 64'd0; else
//Reverse bytes on corresponding configuration
if(rvalid)          fifo_data           <=          cfg_reverse_pixel           ?
{rdata[7:0],rdata[15:8],rdata[23:16],rdata[31:24],rdata[39:32],rdata[47:40],rdata[
55:48],rdata[63:56]} : rdata;


// Generate push signal
always@(posedge clk or negedge rst_n)
if(~rst_n                    ) fifo_push <= 1'b0  ; else
if(cfg_blk_en & (~fifo_in_rst)) fifo_push <= rvalid;    //Push each valid data
```

```
always@(posedge clk or negedge rst_n)
if(~rst_n                                                           ) arvalid <=
1'b0                 ;else // Set valid to 0 on reset
if(arvalid & arready                                                ) arvalid <=
1'b0                 ;else // Resewhen address was taken
if(start                                                            ) arvalid <=
1'b1                 ;else // Set on posedge enalbe
if(fifo_rdy               &               ((~req_in_progress)                |
// Or fifo has enough space and there is no request in progress
     (rvalid & rlast & ~((pix_cnt == 16'd8) & (line_cnt == 16'd1))))) arvalid <=
cfg_blk_en & ~sts_done;     // Or the last pixel is read

 endmodule

//---------------------------------------------------------------------------
------------------
// Project     : AXI2FRAME
// Module Name : AXI2FRAME
// Author      : SZILARD HEGEDUS
// Created     : 05/02/2018
//---------------------------------------------------------------------------
------------------
// Description : Read 3 maps on AXI and output on single channel Frame IF
//---------------------------------------------------------------------------
------------------
// Modification history :
// 05/02/2018 (SH): Initial version
// 26/02/2019 (SH): Made FIFO external
//---------------------------------------------------------------------------
------------------

 module axi2frame#(
  parameter MEM_WIDTH    = 64,
  parameter ADDR_WIDTH   = 32,
  parameter USEDW_BITS   = 11
)(
// --------------------------------------------- System IF ------------------------
------------------
input                       clk              , // System clock
input                       rst_n            , // Asynchronous reset active low
input                       axi0_arready     , // Channel 0 Address ready
input        [MEM_WIDTH-1:0] axi0_rdata      , // Channel 0 Read data
input                         axi0_rlast        , // Channel 0 Last data beat in
transfer
input                       axi0_rvalid      , // Channel 0 Valid
input                  [1:0] axi0_rresp      , // Channel 0 AXI response
output      [ADDR_WIDTH-1:0] axi0_araddr     , // Channel 0 Address
output                 [7:0] axi0_arlen      , // Channel 0 Burst length
output                 [1:0] axi0_arburst    , // Channel 0 Burst type
output                  [2:0] axi0_arsize       , // Channel 0 Number of bytes in
each transfer
output                      axi0_arvalid     , // Channel 0 Address valid
output                      axi0_rready      , // Channel 0 Read ready
```

```
// ----------------------------------------- AXI Channel 1 ------------------------
---------------------
input                        axi1_arready      , // Channel 1 Address ready
input        [MEM_WIDTH-1:0] axi1_rdata        , // Channel 1 Read data
input                          axi1_rlast        , // Channel 1 Last data beat in
transfer
input                        axi1_rvalid       , // Channel 1 Valid data
input                  [1:0] axi1_rresp        , // Channel 1 AXI response
output      [ADDR_WIDTH-1:0] axi1_araddr       , // Channel 1 Address
output                 [7:0] axi1_arlen        , // Channel 1 Burst length
output                 [1:0] axi1_arburst      , // Channel 1 Burst type
output                  [2:0] axi1_arsize        , // Channel 1 Number of bytes in
each transfer
output                       axi1_arvalid      , // Channel 1 Address valid
output                       axi1_rready       , // Channel 1 Read ready
// ----------------------------------------- AXI Channel 2 ------------------------
---------------------
input                        axi2_arready      , // Channel 2 Address ready
input        [MEM_WIDTH-1:0] axi2_rdata        , // Channel 2 Read data
input                          axi2_rlast        , // Channel 2 Last data beat in
transfer
input                        axi2_rvalid       , // Channel 2 Valid data
input                  [1:0] axi2_rresp        , // Channel 2 AXI response
output      [ADDR_WIDTH-1:0] axi2_araddr       , // Channel 2 Address
output                 [7:0] axi2_arlen        , // Channel 2 Burst length
output                 [1:0] axi2_arburst      , // Channel 2 Burst type
output                  [2:0] axi2_arsize        , // Channel 2 Number of bytes in
each transfer
output                       axi2_arvalid      , // Channel 2 Address valid
output                       axi2_rready       , // Channel 2 Read ready
// ------------------------------------ Configuration Interface inputs ------------
-------------------
input                         cfg_blk_en          , // Block enable
input                  [15:0] cfg_img_width       , // Image width
input                  [15:0] cfg_img_height      , // Image height
input                  [15:0] cfg_stride          , // The address distance between
the first address of successive â€œhorizontalâ€? reads
input        [ADDR_WIDTH-1:0] cfg_map0_ba         , // Channel 0 base address
input        [ADDR_WIDTH-1:0] cfg_map1_ba         , // Channel 1 base address
input        [ADDR_WIDTH-1:0] cfg_map2_ba         , // Channel 2 base address
input                         cfg_map0_en         , // Channel 0 enable
input                         cfg_map1_en         , // Channel 1 enable
input                         cfg_map2_en         , // Channel 2 enable
input                   [7:0] cfg_max_burst_length, // Maximum burst length
input                         cfg_reverse_byte    , // Data is is big/small endian
input                         cfg_int_ack         , // Interrupt acknowledge
//------------------------------------  FIFO Interface----------------------------
-----------------
input                        fifo_ch0_empty      ,
input [MEM_WIDTH-1:0]        fifo_ch0_popdata    ,
output                       fifo_ch0_pop        ,
output [MEM_WIDTH-1:0]       fifo_ch0_pushdata   ,
input [USEDW_BITS-1:0]       fifo_ch0_usedwords  ,
output                       fifo_ch0_push       ,
input                        fifo_ch0_full       ,
input                        fifo_ch1_empty      ,
input [MEM_WIDTH-1:0]        fifo_ch1_popdata    ,
output                       fifo_ch1_pop        ,
output [MEM_WIDTH-1:0]       fifo_ch1_pushdata   ,
input [USEDW_BITS-1:0]       fifo_ch1_usedwords  ,
output                       fifo_ch1_push       ,
input                        fifo_ch1_full       ,
input                        fifo_ch2_empty      ,
input [MEM_WIDTH-1:0]        fifo_ch2_popdata    ,
output                       fifo_ch2_pop        ,
output [MEM_WIDTH-1:0]       fifo_ch2_pushdata   ,
input [USEDW_BITS-1:0]       fifo_ch2_usedwords  ,
output                       fifo_ch2_push       ,
input                        fifo_ch2_full       ,
// ------------------------------------ Status IF outputs -------------------------
------------------
output                      sts_axi_error        , // Axi error
output                      sts_read_done        , // Read done interrupt
output  reg                 sts_idle             , // Module in idle state
output  reg                 sts_frm_int          , // Interrupt
//-------------------------------Frame IF-------------------------------------------
---------
output                      frm_val              , // Frame data valid
output                [23:0]frm_data             , // Frame data
output                      frm_sof              , // Frame start of frame
output                      frm_eof              , // Frame end of frame
output                      frm_sol              , // Frame start of line
output                      frm_eol              , // Frame end of line
input                       frm_rdy
 );

wire                  sts_done0        ;
wire                  sts_done1        ;
wire                  sts_done2        ;

wire start;
reg cfg_blk_en_d;
wire vga_rst_rd;
reg frm_eof_d;

assign vga_rst_rd = frm_eof_d & (~frm_eof); // Self-reset on negedge eof

assign sts_axi_error = (cfg_map0_en & (axi0_rresp != 0)) | (cfg_map1_en & (axi1_rresp
!= 0)) | (cfg_map2_en & (axi2_rresp != 0)); // Error if response not 0
assign sts_read_done = (sts_done0 | (~cfg_map0_en)) & (sts_done1 | (~cfg_map1_en))
& (sts_done2 | (~cfg_map2_en))               ; // Read done when all sts_done is 1
assign   start                               =    cfg_blk_en    &    (~cfg_blk_en_d)
; //Start at posedge enable

 //Read done interrupt
always@(posedge clk or negedge rst_n)
if(~rst_n)                   sts_frm_int <= 1'b0      ;else
if(cfg_int_ack | vga_rst_rd) sts_frm_int <= 1'b0       ;else // Reset on interrupt
ack or vga read reset
```

```
if(sts_read_done & frm_eof ) sts_frm_int <= cfg_blk_en;     // Set when done reading
memory and frame sent last pixel

always@(posedge clk or negedge rst_n)
if(~rst_n)                   sts_idle <= 1'b0;else
if(start)                    sts_idle <= 1'b0;else // Reset on posedge enalbe
if(sts_read_done & frm_eof) sts_idle <= 1'b1;     // Set on done reading memory and
and frame sent last pixel


 //Delay enable
always@(posedge clk or negedge rst_n)
if(~rst_n) cfg_blk_en_d <= 1'b0      ;else
           cfg_blk_en_d <= cfg_blk_en;

always@(posedge clk or negedge rst_n)
if(~rst_n) frm_eof_d <= 1'b0   ;else
           frm_eof_d <= frm_eof;

 axi2fifo#(
  .ADDR_WIDTH (ADDR_WIDTH),
  .USEDW_BITS (USEDW_BITS)
 )axi2fifo0(
   // -------------------------------------------- System IF ---------------------
---------------------
  .clk                 (clk                      ), // System clock
  .rst_n               (rst_n                    ), // Asynchronous reset active low
  // ---------------------------------------- AXI ----------------------------------
-------------
  .arready             (axi0_arready             ), // Address ready
  .rdata               (axi0_rdata               ), // Read data
  .rlast               (axi0_rlast               ), // Last data beat in transfer
  .rvalid              (axi0_rvalid              ), // Valid data
  .araddr              (axi0_araddr              ), // Address
  .arlen               (axi0_arlen               ), // Burst length
  .arburst             (axi0_arburst             ), // Burst type
  .arsize                (axi0_arsize                ), // Number of bytes in each
transfer
  .arvalid             (axi0_arvalid             ), // Address valid
  .rready              (axi0_rready              ), // Read ready
  // ---------------------------------- Configuration Interface inputs -----------
--------------------
  .cfg_blk_en          (cfg_map0_en & cfg_blk_en & (~vga_rst_rd)), // Block enable
  .cfg_img_width       (cfg_img_width            ), // Image width
  .cfg_img_height      (cfg_img_height           ), // Image height
  .cfg_stride          (cfg_stride               ), // The address distance between
the first address of successive â€œhorizontalâ€? reads
  .cfg_map_ba          (cfg_map0_ba              ), // Channel base address
  .cfg_max_burst_length(cfg_max_burst_length     ), // Maximum burst length
  .cfg_reverse_pixel   (cfg_reverse_byte         ), // Data is is big/little endian
  // ------------------------------------ FIFO inputs -----------------------------
---------------------
  .fifo_words_used     (fifo_ch0_usedwords       ), // Used word in FIFO
  .fifo_full           (fifo_ch0_full            ),
  .fifo_empty          (fifo_ch0_empty           ), // FIFO empty
  // ------------------------------------ FIFO outputs -----------------------------
---------------------
  .fifo_push           (fifo_ch0_push           ), // Push
  .fifo_data           (fifo_ch0_pushdata       ), // Output data
  // ------------------------------------ Status IF outputs ------------------------
--------------------
  .sts_done            (sts_done0               )  // Done interrupt
 );

  axi2fifo#(
  .ADDR_WIDTH (ADDR_WIDTH),
  .USEDW_BITS (USEDW_BITS)
 )axi2fifo1(
// ---------------------------------------------- System IF -------------------------
------------------
  .clk                 (clk                     ), // System clock
  .rst_n               (rst_n                   ), // Asynchronous reset active low
// -------------------------------------------- AXI ----------------------------------
-----------
  .arready             (axi1_arready            ), // Address ready
  .rdata               (axi1_rdata              ), // Read data
  .rlast               (axi1_rlast              ), // Last data beat in transfer
  .rvalid              (axi1_rvalid             ), // Valid data
  .araddr              (axi1_araddr             ), // Address
  .arlen               (axi1_arlen              ), // Burst length
  .arburst             (axi1_arburst            ), // Burst type
  .arsize                (axi1_arsize              ), // Number of bytes in each
transfer
  .arvalid             (axi1_arvalid            ), // Address valid
  .rready              (axi1_rready             ), // Read ready
// ------------------------------------- Configuration Interface inputs ------------
-------------------
  .cfg_blk_en          (cfg_map1_en & cfg_blk_en & (~vga_rst_rd)), // Block enable
  .cfg_img_width       (cfg_img_width           ), // Image width
  .cfg_img_height      (cfg_img_height          ), // Image height
  .cfg_stride          (cfg_stride               ), // The address distance between
the first address of successive â€œhorizontalâ€? reads
  .cfg_map_ba          (cfg_map1_ba             ), // Channel base address
  .cfg_max_burst_length(cfg_max_burst_length    ), // Maximum burst length
  .cfg_reverse_pixel   (cfg_reverse_byte        ), // Data is is big/small endian
// ------------------------------------- FIFO inputs -------------------------------
-------------------
  .fifo_words_used     (fifo_ch1_usedwords      ), // Used word in FIFO
  .fifo_full           (fifo_ch1_full           ),
  .fifo_empty          (fifo_ch1_empty          ), // FIFO empty
// ------------------------------------- FIFO outputs ------------------------------
-------------------
  .fifo_push           (fifo_ch1_push           ), // Push
  .fifo_data           (fifo_ch1_pushdata       ), // Output data
// ------------------------------------- Status IF outputs -------------------------
------------------
  .sts_done            (sts_done1               )  // Done interrupt
 );
```

```
axi2fifo#(
  .ADDR_WIDTH (ADDR_WIDTH),
  .USEDW_BITS (USEDW_BITS)
 )axi2fifo2(
   // -------------------------------------------- System IF ----------------------
---------------------
  .clk                 (clk                     ), // System clock
  .rst_n               (rst_n                   ), // Asynchronous reset active low
  // ----------------------------------------- AXI inputs -------------------------
--------------------
  .arready             (axi2_arready            ), // Address ready
  .rdata               (axi2_rdata              ), // Read data
  .rlast               (axi2_rlast              ), // Last data beat in transfer
  .rvalid              (axi2_rvalid             ), // Valid data
  .araddr              (axi2_araddr             ), // Address
  .arlen               (axi2_arlen              ), // Burst length
  .arburst             (axi2_arburst            ), // Burst type
  .arsize                (axi2_arsize               ), // Number of bytes in each
transfer
  .arvalid             (axi2_arvalid            ), // Address valid
  .rready              (axi2_rready             ), // Read ready
  // ------------------------------------ Configuration Interface inputs ----------
---------------------
  .cfg_blk_en          (cfg_map2_en & cfg_blk_en & (~vga_rst_rd)), // Block enable
  .cfg_img_width       (cfg_img_width           ), // Image width
  .cfg_img_height      (cfg_img_height          ), // Image height
  .cfg_stride          (cfg_stride              ), // The address distance between
the first address of successive â€œhorizontalâ€? reads
  .cfg_map_ba          (cfg_map2_ba             ), // Channel base address
  .cfg_max_burst_length(cfg_max_burst_length    ), // Maximum burst length
  .cfg_reverse_pixel   (cfg_reverse_byte        ), // Data is is big/little endian
  // ------------------------------------ FIFO inputs ------------------------------
---------------------
  .fifo_words_used     (fifo_ch2_usedwords      ), // Used word in FIFO
  .fifo_full           (fifo_ch2_full           ),
  .fifo_empty          (fifo_ch2_empty          ), // FIFO empty
  // ------------------------------------- FIFO outputs ----------------------------
---------------------
  .fifo_push           (fifo_ch2_push           ), // Push
  .fifo_data           (fifo_ch2_pushdata       ), // Output data
  // ------------------------------------ Status IF outputs -----------------------
--------------------
  .sts_done            (sts_done2               )  // Done interrupt
 );

 fifo2frm_3map#(
   .FIFO_DATA_WIDTH(MEM_WIDTH)
 )fifo2frm(
//-------------------------------System IF-----------------------------------------
-----------------
   .clk              (clk           ), // System clock
   .rst_n            (rst_n         ), // Asynchronous reset active low
//-------------------------------FIFO inputs---------------------------------------
-----------------
   .fifo_ch0_empty  (fifo_ch0_empty   ), // FIFO empty
   .fifo_ch1_empty  (fifo_ch1_empty   ), // FIFO empty
   .fifo_ch2_empty  (fifo_ch2_empty   ), // FIFO empty
   .fifo_ch0_full   (fifo_ch0_full    ), // FIFO full
   .fifo_ch1_full   (fifo_ch1_full    ), // FIFO full
   .fifo_ch2_full   (fifo_ch2_full    ), // FIFO full
   .fifo_ch0_popdata(fifo_ch0_popdata ), // FIFO data
   .fifo_ch1_popdata(fifo_ch1_popdata ), // FIFO data
   .fifo_ch2_popdata(fifo_ch2_popdata ), // FIFO data
//------------------------------Configuration IF inputs-------------------------------
-----------------
   .cfg_blk_en    (cfg_blk_en  & (~vga_rst_rd)),
   .cfg_map0_en   (cfg_map0_en   ), // Channel 0 enable
   .cfg_map1_en   (cfg_map1_en   ), // Channel 1 enable
   .cfg_map2_en   (cfg_map2_en   ), // Channel 2 enable
   .cfg_img_width (cfg_img_width ), // Image width
   .cfg_img_height(cfg_img_height), // Image height
//------------------------------Frame IF inputs-------------------------------------
-----------------
   .frm_rdy       (frm_rdy       ), // Frame ready
//------------------------------FIFO outputs----------------------------------------
-----------------
   .fifo_ch0_pop  (fifo_ch0_pop  ), // FIFO pop
   .fifo_ch1_pop  (fifo_ch1_pop  ), // FIFO pop
   .fifo_ch2_pop  (fifo_ch2_pop  ), // FIFO pop
//-------------------------------Frame IF outputs-----------------------------------
-----------------
   .frm_val       (frm_val       ), // Frame data valid
   .frm_data      (frm_data      ), // Frame data
   .frm_sof       (frm_sof       ), // Frame start of frame
   .frm_eof       (frm_eof       ), // Frame end of frame
   .frm_sol       (frm_sol       ), // Frame start of line
   .frm_eol       (frm_eol       )  // Frame end of line
 );

 endmodule

// Project     : ir_filters
// Module Name : axi_stream2Frame
// Author      : Szilard Hegedus
// Created     : 01/21/2019
//---------------------------------------------------------------------------------
------------------
// Description : Converts AXI4 Stream interface to Frame interface
//---------------------------------------------------------------------------------
------------------
// Modification history :
// 11/15/2018 (SH): Initial version
//---------------------------------------------------------------------------------
------------------


module fifo2frame#(
  parameter DATA_WIDTH = 24
)(
  input                         clk                     , // Syste clock
  input                         rst_n                 , // Asynchronous reset active
low
  input                        sw_rst                ,
//------------------------- Configuration interface -------------------------------
---------------
  input  [15:0]                cfg_img_w             , // Image width
  input  [15:0]                cfg_img_h             , // Image width
//------------------------- FIFO RD interface -------------------------------------
------------
  output reg                   fifo_pop              , // Start of frame
  input     [DATA_WIDTH-1:0]  fifo_popdata          , // Slave has valid data to be
transferred
  input                        fifo_empty            , // End of frame
  input                       fifo_full             , // Data transferred from slave
to master
  input                        fifo_almost_empty     ,
  input                        fifo_almost_full      ,
// ------------------------------ Frame Interface ---------------------------------
---------------
  output reg                    s_frm_val              , // Master has valid data to
be transferred
  input                         s_frm_rdy             , // Slave is ready to receive
the data
  output     [DATA_WIDTH-1:0] s_frm_data           , // Data transferred from master
to slave
  output reg                   s_frm_sof             , // Start of Frame
  output reg                   s_frm_eof             , // End of Frame
  output reg                   s_frm_sol             , // Start of Line
  output reg                   s_frm_eol               // End of Line
);

reg [11:0] pix_cnt ;
reg [11:0] line_cnt;

wire fifo_rst_state;

assign fifo_rst_state = (~((~fifo_full) & (~fifo_empty)));

wire outvalrdy;

assign outvalrdy = s_frm_rdy & s_frm_val;

reg fifo_loaded;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) fifo_loaded <= 1'b0; else
if(sw_rst                        ) fifo_loaded <= 1'b0; else
if(fifo_rst_state &(~fifo_loaded)) fifo_loaded <= 1'b0; else
if(fifo_almost_full              ) fifo_loaded <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                                  ) pix_cnt <= 11'd0         ; else
if(pix_cnt == (cfg_img_w - 1'd1) & outvalrdy) pix_cnt <= 11'd0         ; else
if(outvalrdy & fifo_loaded                  ) pix_cnt <= pix_cnt + 1'd1;
```

```
always@(posedge clk or negedge rst_n)
if(~rst_n                                                                  )
line_cnt <= 11'd0          ; else
if((line_cnt == (cfg_img_h - 1'd1)) & (pix_cnt == (cfg_img_w - 1'd1)) & outvalrdy)
line_cnt <= 11'd0          ; else
if((pix_cnt == (cfg_img_w - 1'd1)) & outvalrdy & fifo_loaded                )
line_cnt <= line_cnt + 1'd1;


always@(posedge clk or negedge rst_n)
if(~rst_n                                                                  )
s_frm_sol <= 1'b0; else
if(outvalrdy & s_frm_sol                                                   )
s_frm_sol <= 1'b0; else
if(~fifo_rst_state & ~fifo_loaded & fifo_almost_full                       )
s_frm_sol <= 1'b1; else
if((line_cnt == (cfg_img_h - 1'd1)) & (pix_cnt == (cfg_img_w - 1'd1)) & outvalrdy)
s_frm_sol <= 1'b1; else
if(outvalrdy & s_frm_eol & (~s_frm_eof)                                    )
s_frm_sol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                                                                  )
s_frm_eof <= 1'b0; else
if(outvalrdy & s_frm_eof                                                   )
s_frm_eof <= 1'b0; else
if((line_cnt == (cfg_img_h - 1'd1))  & (pix_cnt == (cfg_img_w - 2'd2)) & outvalrdy)
s_frm_eof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                            ) s_frm_val <= 1'b0; else
if(s_frm_rdy & s_frm_val & (~fifo_pop)) s_frm_val <= 1'b0; else
if(s_frm_rdy & fifo_loaded           ) s_frm_val <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                                    ) s_frm_eol <= 1'b0; else
if(outvalrdy & s_frm_eol                     ) s_frm_eol <= 1'b0; else
if((pix_cnt == (cfg_img_w - 2'd2)) & outvalrdy) s_frm_eol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                                                                  )
s_frm_sof <= 1'b0; else
if(outvalrdy & s_frm_sof                                                   )
s_frm_sof <= 1'b0; else
if(~fifo_rst_state & ~fifo_loaded & fifo_almost_full                       )
s_frm_sof <= 1'b1; else
if((line_cnt == (cfg_img_h - 1'd1)) & (pix_cnt == (cfg_img_w - 1'd1)) & outvalrdy)
s_frm_sof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                                                  ) fifo_pop <= 1'd0
; else
if(fifo_almost_empty & fifo_pop                            ) fifo_pop <= 1'd0
; else
```

```
if(fifo_almost_full  &  (~fifo_loaded)  &  (~fifo_rst_state))  fifo_pop  <=  1'd1
; else
if(fifo_loaded                                           ) fifo_pop <= s_frm_rdy &
s_frm_val;

assign s_frm_data = fifo_popdata;

endmodule //axi_stream2Frame

//--------------------------------------------------------------------------------
------------------
// Project     : AXI2FRAME
// Module Name : FIFO2FRM_3MAP
// Author      : SZILARD HEGEDUS
// Created     : 05/02/2018
//--------------------------------------------------------------------------------
------------------
// Description : Converts 3 channel input into single channel output on FI
//--------------------------------------------------------------------------------
------------------
// Modification history :
// 05/02/2018 (SH): Initial version
//--------------------------------------------------------------------------------
------------------


 module fifo2frm_3map#(
   parameter FIFO_DATA_WIDTH = 64
 )(
//-------------------------------System IF-----------------------------------------
-----------------
   input                           clk             , // System clock
   input                            rst_n           , // Asynchronous reset active
low
//-------------------------------FIFO inputs---------------------------------------
-----------------
   input                           fifo_ch0_empty  , // FIFO empty
   input                           fifo_ch1_empty  , // FIFO empty
   input                           fifo_ch2_empty  , // FIFO empty
   input                           fifo_ch0_full   , // FIFO empty
   input                           fifo_ch1_full   , // FIFO empty
   input                           fifo_ch2_full   , // FIFO empty
   input      [FIFO_DATA_WIDTH-1:0]fifo_ch0_popdata, // FIFO data
   input      [FIFO_DATA_WIDTH-1:0]fifo_ch1_popdata, // FIFO data
   input      [FIFO_DATA_WIDTH-1:0]fifo_ch2_popdata, // FIFO data
   output reg                      fifo_ch0_pop     , // FIFO pop
   output reg                      fifo_ch1_pop     , // FIFO pop
   output reg                      fifo_ch2_pop     , // FIFO pop
//------------------------------Configuration IF inputs----------------------------
-----------------
   input                           cfg_blk_en      , // Block enable
   input                           cfg_map0_en     , // Channel 0 enable
   input                           cfg_map1_en     , // Channel 1 enable
   input                           cfg_map2_en     , // Channel 2 enable
   input                     [15:0]cfg_img_width   , // Image width
   input                     [15:0]cfg_img_height  , // Image height
//-------------------------------Frame IF    ---------------------------------------
-----
  output reg                      frm_val          , // Frame data valid
  output reg                [23:0]frm_data         , // Frame data
  output reg                      frm_sof          , // Frame start of frame
  output reg                      frm_eof          , // Frame end of frame
  output reg                      frm_sol          , // Frame start of line
  output reg                      frm_eol          , // Frame end of line
  input                           frm_rdy            // Frame ready

 );

reg [FIFO_DATA_WIDTH-1:0]data0        ; // Axi channle 0 data
reg [FIFO_DATA_WIDTH-1:0]data1        ; // Axi channle 1 data
reg [FIFO_DATA_WIDTH-1:0]data2        ; // Axi channle 2 data
reg                [15:0]pixel_cnt    ; // Pixel counter
reg                [15:0]line_cnt     ; // Line counter
wire                     start        ; // Start on posedge enable
wire                     cfg_map_en   ; // Maps are enabled
wire                     pop_en       ; // Enable pop
reg                      cfg_blk_en_d ; // Delay block enable
reg                      fifo_ch_pop_d; // Delay pop
reg                [3:0] nr_byte      ; // Count revieved bytes from input
wire                     frm_valrdy   ; // val & rdy
reg                      sts_frm_done ; // Indicates frame sent


assign start      = cfg_blk_en & (~cfg_blk_en_d)        ; // Start on posedge enable
assign frm_valrdy = frm_val & frm_rdy                   ;
assign cfg_map_en = (cfg_map0_en|cfg_map1_en|cfg_map2_en); // Enable block when at
least one of the maps is enabled
assign pop_en     = (fifo_ch0_empty ^ cfg_map0_en) & (fifo_ch1_empty ^ cfg_map1_en)
& (fifo_ch2_empty ^ cfg_map2_en) // Pop when enabled fifos are not empty
                     & (nr_byte < 1) & (~sts_frm_done); // And frame not done and
last byte was recieved from input

//Extract last byte from data
always@(posedge clk or negedge rst_n)
if(~rst_n                    ) data0 <= {FIFO_DATA_WIDTH{1'd0}}          ;else //
Set data 0 on reset
if(~cfg_map0_en              ) data0 <= {FIFO_DATA_WIDTH{1'd0}}          ;else //
Set data 0 on reset
if(fifo_ch_pop_d & frm_valrdy) data0 <= fifo_ch0_popdata                 ;else //
Load data from fifo after pop
if(frm_valrdy                ) data0 <= {8'd0,data0[FIFO_DATA_WIDTH-1:8]};      //
Get last byte from data

always@(posedge clk or negedge rst_n)
if(~rst_n                    ) data1 <= {FIFO_DATA_WIDTH{1'd0}}          ;else //
Set data 0 on reset
if(~cfg_map0_en              ) data1 <= {FIFO_DATA_WIDTH{1'd0}}          ;else //
Set data 0 on reset
if(fifo_ch_pop_d & frm_valrdy) data1 <= fifo_ch1_popdata                 ;else //
Load data from fifo after pop
```

```
if(frm_valrdy                 ) data1 <= {8'd0,data1[FIFO_DATA_WIDTH-1:8]};     //
Get last byte from data

always@(posedge clk or negedge rst_n)
if(~rst_n                     ) data2 <= {FIFO_DATA_WIDTH{1'd0}}           ;else //
Set data 0 on reset
if(~cfg_map0_en               ) data2 <= {FIFO_DATA_WIDTH{1'd0}}           ;else //
Set data 0 on reset
if(fifo_ch_pop_d & frm_valrdy) data2 <= fifo_ch2_popdata                   ;else //
Load data from fifo after pop
if(frm_valrdy                 ) data2 <= {8'd0,data2[FIFO_DATA_WIDTH-1:8]};     //
Get last byte from data

//-------------------------------------------------- Pop signal ---------------------
-------------------------------------
always@(posedge clk or negedge rst_n)
if(~rst_n      ) fifo_ch0_pop <= 1'b0       ;else
if(fifo_ch0_pop) fifo_ch0_pop <= 1'b0       ;else // Reset pop after 1 cycle
if(pop_en      ) fifo_ch0_pop <= cfg_map0_en;     // Set on block and pop enable

always@(posedge clk or negedge rst_n)
if(~rst_n      )fifo_ch1_pop <= 1'b0       ;else
if(fifo_ch1_pop)fifo_ch1_pop <= 1'b0       ;else // Reset pop after 1 cycle
if(pop_en      )fifo_ch1_pop <= cfg_map1_en;     // Set on block and pop enable

always@(posedge clk or negedge rst_n)
if(~rst_n      ) fifo_ch2_pop <= 1'b0       ;else
if(fifo_ch2_pop) fifo_ch2_pop <= 1'b0       ;else // Reset pop after 1 cycle
if(pop_en      ) fifo_ch2_pop <= cfg_map2_en;     // Set on block and pop enable

//-------------------------------------- Internal registers -------------------------
-------------------
always@(posedge clk or negedge rst_n)
if(~rst_n ) sts_frm_done <= 1'b0      ;else
if(start  ) sts_frm_done <= 1'b0      ;else // Reset on posedge enable
if(frm_eof) sts_frm_done <= cfg_map_en;     // Set at end of frame


always@(posedge clk or negedge rst_n)
if(~rst_n                        ) frm_sol <= 1'b0      ;else
if(frm_sol & frm_valrdy          ) frm_sol <= 1'b0      ;else // Reset after 1 cycle
if((frm_eol & frm_valrdy) | start) frm_sol <= cfg_map_en;      // Set at posedge
enbble

always@(posedge clk or negedge rst_n)
if(~rst_n    ) frm_sof <= 1'b0      ;else
if(frm_valrdy) frm_sof <= 1'b0      ;else // Reset on val&rdy
if(start     ) frm_sof <= cfg_map_en;     // Set on posedge enable

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) frm_eol <= 1'b0      ;else
if(frm_eol & frm_valrdy          ) frm_eol <= 1'b0      ;else // Reset after 1 cycle
if((pixel_cnt == 2) & frm_valrdy) frm_eol <= cfg_map_en;      // Set when at last
pixel
```

```
always@(posedge clk or negedge rst_n)
if(~rst_n                                         ) frm_eof <= 1'b0       ;else //
Set default 0
if((frm_eof & frm_valrdy) | start                 ) frm_eof <= 1'b0       ;else //
Reset on
if((line_cnt == 1) & (pixel_cnt == 2) & frm_valrdy) frm_eof <= cfg_map_en;       //
Set

// Line counter
always@(posedge clk or negedge rst_n)
if(~rst_n                              ) line_cnt <= 11'd0           ;else // Set
default 0
if(start                               ) line_cnt <= cfg_img_height  ;else // Load
image height on start, posedge enalbe
if(frm_valrdy & frm_eol & ~sts_frm_done) line_cnt <= line_cnt - 1'd1 ;       //
Decrement on val&rdy

//Pixel counter
always@(posedge clk or negedge rst_n)
if(~rst_n                       ) pixel_cnt <= 11'd0          ;else // Set default
value to 0
if(start | (frm_eol & frm_valrdy)) pixel_cnt <= cfg_img_width   ;else // Load image
width on posedge enalbe or end of line
if(frm_valrdy & ~sts_frm_done    ) pixel_cnt <= pixel_cnt - 1'd1;      // Decrement
on val&rdy

// Number the bytes separated from data
always@(posedge clk or negedge rst_n)
if(~rst_n      ) nr_byte <= 4'd0          ;else // Set default value to 0
if(fifo_ch0_pop) nr_byte <= 4'd8          ;else // Load image width on start or end
of line
if(frm_valrdy  ) nr_byte <= nr_byte - 1'd1;     // Decrement otherwise



//--------------------------------------------------- Frame interface signals ------
---------------------------------------
always@(posedge clk or negedge rst_n)
if(~rst_n                    ) frm_val <= 1'd0;else
if((nr_byte == 1) & frm_rdy) frm_val <= 1'd0;else // Reset valid on ready and after
last byte was separated in the shift register
if(fifo_ch_pop_d             ) frm_val <= 1'd1;     // First valid data 1 cicle after
pop

always@(posedge clk or negedge rst_n)
if(~rst_n    ) frm_data <= 24'd0                               ; else
if(frm_valrdy) frm_data <= {data2[7:0], data1[7:0], data0[7:0]}; // Combine 3 channel
data

//Delay enable
always@(posedge clk or negedge rst_n)
if(~rst_n) cfg_blk_en_d <= 1'd0      ;else
           cfg_blk_en_d <= cfg_blk_en;
```

```
//Delay pop
always@(posedge clk or negedge rst_n)
if(~rst_n      ) fifo_ch_pop_d <= 1'd0; else
if(fifo_ch0_pop) fifo_ch_pop_d <= 1'd1; else
if(frm_valrdy  ) fifo_ch_pop_d <= 1'd0;


endmodule

// Project     : ir_filters
// Module Name : intr_gen
// Author      : Szilard Hegedus
// Created     : 01/21/2019
//---------------------------------------------------------------------------
------------------
// Description : Generates interrupt from input pulse stimulus
//---------------------------------------------------------------------------
------------------
// Modification history :
// 11/15/2018 (SH): Initial version
//---------------------------------------------------------------------------
------------------

module intr_gen(
  input      clk     , // System clock
  input      rst_n   , // Reset active low
  input      stimulus, // Input stimulus
  input      intr_ack, // Interrupt acknowledge
  output reg intr      // Interrupt
);

always@(posedge clk or negedge rst_n)
if(~rst_n  ) intr <= 1'b0; else // Reset at hardware reset
if(intr_ack) intr <= 1'b0; else // Reset at acknowledge
if(stimulus) intr <= 1'b1;      // Set at input stimulus


endmodule

// Project     : ir_filters
// Module Name : laplace_filter_1px
// Author      : Szilard Hegedus
// Created     : 11/15/2018
//---------------------------------------------------------------------------
// Description : Applies 3x3laplace filter
//               ____________
//              |   |   |   |
//              | 0 |-1 | 0 |
//              |___|___|___|
//              |   |   |   |
//              |-1 | 4 |-1 |
//              |___|___|___|
//              |   |   |   |
//              | 0 |-1 | 0 |
//              |___|___|___|
//
//-------------------------------------------------------------------------
// Modification history :
// 11/15/2018 (SH): Initial version
// 02/04/2019 (SH): Replaced 4 pixel/cycle to 1 pixel/cycle to integrate into
Pcam5c_demo reference design

module laplace_filter_1px#(
parameter DATA_WIDTH = 8
)(
input                         clk       ,
input                         rst_n     ,
//-----------------------------Input Frame Interface-----------------------------------
---------
input                        in3x3_val , // Master has valid data to be transferred
output                        in3x3_rdy ,  // Slave is ready to receive the data
input      [9*DATA_WIDTH-1:0] in3x3_data, // Data transferred from master to slave
input                         in3x3_sof , // Start of frame
input                         in3x3_sol , // Start of line
input                         in3x3_eol , // End of line
input                         in3x3_eof , // End of frame
//-----------------------------Output Frame Interface----------------------------------
----------
output reg                   out_val   , // Master has valid data to be transferred
input                         out_rdy   , // Slave is ready to receive the data
output reg [  DATA_WIDTH-1:0] out_data  , // Data transferred from master to slave
output reg                    out_sof   , // Start of frame
output reg                    out_sol   , // Start of line
output reg                    out_eol   , // End of line
output reg                    out_eof     // End of frame
);

//------------------------------Internal signals-------------------------------------
----------
wire [DATA_WIDTH-1:0] p00; //Pixel in window
wire [DATA_WIDTH-1:0] p01; //Pixel in window
wire [DATA_WIDTH-1:0] p02; //Pixel in window
wire [DATA_WIDTH-1:0] p10; //Pixel in window
wire [DATA_WIDTH-1:0] p11; //Pixel in window
wire [DATA_WIDTH-1:0] p12; //Pixel in window
wire [DATA_WIDTH-1:0] p20; //Pixel in window
wire [DATA_WIDTH-1:0] p21; //Pixel in window
wire [DATA_WIDTH-1:0] p22; //Pixel in window

wire [DATA_WIDTH+1:0] sum;

wire invalrdy;

assign invalrdy =  in3x3_rdy & in3x3_val;
assign in3x3_rdy = out_rdy;

assign p00 = in3x3_data[9*DATA_WIDTH-1:8*DATA_WIDTH];
assign p01 = in3x3_data[8*DATA_WIDTH-1:7*DATA_WIDTH];
assign p02 = in3x3_data[7*DATA_WIDTH-1:6*DATA_WIDTH];
assign p10 = in3x3_data[6*DATA_WIDTH-1:5*DATA_WIDTH];
assign p11 = in3x3_data[5*DATA_WIDTH-1:4*DATA_WIDTH];
assign p12 = in3x3_data[4*DATA_WIDTH-1:3*DATA_WIDTH];
assign p20 = in3x3_data[3*DATA_WIDTH-1:2*DATA_WIDTH];
assign p21 = in3x3_data[2*DATA_WIDTH-1:1*DATA_WIDTH];
assign p22 = in3x3_data[1*DATA_WIDTH-1:0*DATA_WIDTH];


assign sum = ({p11, 3'b0} + {p11, 2'b0}) - {p01, 1'b0} - {p10, 1'b0} - {p12, 1'b0}
- {p21, 1'b0} - p02 - p20 - p22 - p00;

always@(posedge clk or negedge rst_n)
if(~rst_n                                                )    out_data    <=    8'd0
; else
if(in3x3_val & in3x3_rdy) out_data <= sum[DATA_WIDTH+1] ? 0 : ((sum[DATA_WIDTH : 0]
> {DATA_WIDTH{1'b1}}) ? {DATA_WIDTH{1'b1}} : sum); // Recieve only the top 8 pixels,
that will be the result of division by 16

always@(posedge clk or negedge rst_n)
if(~rst_n                          ) out_eof <= 1'b0; else
if(out_rdy & out_val & out_eof     ) out_eof <= 1'b0; else
if(in3x3_eof & in3x3_val & in3x3_rdy) out_eof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                          ) out_sof <= 1'b0; else
if(out_rdy & out_val & out_sof     ) out_sof <= 1'b0; else
if(in3x3_sof & in3x3_val & in3x3_rdy) out_sof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                          ) out_eol <= 1'b0; else
if(out_rdy & out_val & out_eol     ) out_eol <= 1'b0; else
if(in3x3_eol & in3x3_val & in3x3_rdy) out_eol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                          ) out_sol <= 1'b0; else
if(out_rdy & out_val & out_sol     ) out_sol <= 1'b0; else
if(in3x3_sol & in3x3_val & in3x3_rdy) out_sol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                ) out_val <= 1'b0; else
if(out_rdy & (~in3x3_val)) out_val <= 1'b0; else
if(invalrdy              ) out_val <= 1'b1;


endmodule

//---------------------------------------------------------------------------
// Project     : ir_filters
// Module Name : line_buffer
// Author      : Szilard Hegedus
// Created     : 09/28/2018
//----------------------------------------------------------------------------
------------------
// Description : // Description : Creates 3x6 matrix for 3x3 1px per cycle filter
modules
//               frame input: 1 pixels/cycle
//               frame output: 3x3 pixels window/cycle
//               output image size is equal to the input image size.
//               The input image is bordered with cfg_bkg color
//
//                           ____                ____             ____
//                          |    |               |    |           |    |
//frm_data ---------------->|    |----1pixel---->|    |--1pixel-->|    |--1pixel-->
//                          |>   | |             |>   |           |>   |
//                          |____| |             |____|           |____|
//                                 |
//      |--------------------------|
//      |   ____________     ____                ____             ____
//      |  |            |   |    |               |    |           |    |
//       ->|   FIFO     |-->|    |----1pixel---->|    |--1pixel-->|    |--1pixel-->
//         |____________| | |>   |               |>   |           |>   |
//                        | |____|               |____|           |____|
//                        |
//     |------------------|
//     |    ____________     ____                ____             ____
//     |   |            |   |    |               |    |           |    |
//     |-->|   FIFO     |-->|    |----1pixel---->|    |--1pixel-->|    |--1pixel-->
//         |____________|   |>   |               |>   |           |>   |
//                          |____|               |____|           |____|
//
//---------------------------------------------------------------------------------
------------------
// Modification history :
// 09/28/2018 (SH): Initial version
// 11/19/2018 (SH): Added configurable background value
// 01/28/2019 (SH): Rewrite to output 3x3 matrix instead of 3x3, removed 1 px per
cycle feature
//---------------------------------------------------------------------------------
------------------

module line_buffer#(
  parameter DATA_WIDTH = 8 ,
  parameter USEDW_BITS = 10     // Number of bits for address inside FIFO (depth =
2^USEDW_BITS)

)(
  input                           clk           , // System clock
  input                           rst_n         , // Asynchronous reset active low
  input                           sw_rst        , // Software reset
//--------------------------------Configuration------------------------------------
------------------
  input      [DATA_WIDTH-1:0]     cfg_bkg       , // Background border value
//---------------------------------Input frame interface----------------------------
-----------------
  input                            frm_val       , // Master has valid data to be
transferred
  output reg                      frm_rdy       , // Slave is ready to receive the
data
  input      [DATA_WIDTH-1:0]      frm_data      , // Data transferred from master
to slave
  input                           frm_sof       , // Start of Frame
  input                           frm_eof       , // End of Frame
  input                             frm_sol       , // Start of line
  input                             frm_eol       , // End of line

//--------------------------------Output frame interface--------------------------
------------------
  output reg                           win_val     , // Master has valid data to be
transferred
  input                           win_rdy    , // Slave is ready to receive the data
  output reg [9*DATA_WIDTH-1:0]    win_data    , // Data transferred from master to
slave
  output reg                      win_sof    , // Start of Frame
  output reg                      win_eof    , // End of Frame
  output reg                      win_sol    , // Start of line
  output reg                      win_eol    , // End of line

//-----------------------------------FIFO interface-------------------------------
-----------------
  output reg                      fifo_push     , // Master pushes data frm to FIFO
  output reg [2*DATA_WIDTH-1:0]   fifo_pushdata , // Data stored into FIFO
  input                           fifo_full     , // FIFO full
  output reg                      fifo_pop      , // Master pops data from FIFO
  input      [2*DATA_WIDTH-1:0]   fifo_popdata  , // Data retrieved from FIFO
  input                           fifo_empty    , // FIFO empty
  input      [USEDW_BITS-1  :0]   fifo_usedwords, // Used words frm FIFO
  output reg                      fifo_clr        // Clear Fifo
);

//-------------------------------- Internal registers/signals ---------------------
--------------------------
//Registers for the 3x6 window
reg [DATA_WIDTH-1:0] line0_mid     ;
reg [DATA_WIDTH-1:0] line1_mid     ;
reg [DATA_WIDTH-1:0] line2_mid     ;
reg [DATA_WIDTH-1:0] line0_left    ;
reg [DATA_WIDTH-1:0] line1_left    ;
reg [DATA_WIDTH-1:0] line2_left    ;
reg                    frm_first_line; // First line flag
reg                    last_line     ; // Last line flag
reg                    win_first_line; // First line flag
reg [  USEDW_BITS-1:0] window_cnt    ; // Count number of windows inputed frm row
reg                    in_frm        ;
reg [  USEDW_BITS-1:0] pix_in_line   ;
reg [             1:0] valrdy_cnt    ;
reg                    win_last_line ; // last line received from input
reg [             1:0] frm_sof_d     ;
reg                    mask_sol      ; // Mask data window right side
reg                    mask_eol      ; // Mask data window left side
reg                    last_push     ;

wire                    frmvalrdy    ; // input valrdy
wire                    winvalrdy    ; // output valrdy
wire                    pipe_en      ; // Pipe enable
wire                    set_eol      ;
wire                    set_sol      ;
reg                     set_initial_pop  ;
```

```
wire                    initial_pop  ;
reg                     set_eof      ;
reg                     fifo_in_rst;
//--------------------------------------------------------------------------------
------------------
//                                          Code
//--------------------------------------------------------------------------------
------------------

assign frmvalrdy = frm_val & frm_rdy                ; // input valid ready
assign winvalrdy = win_rdy & win_val                ; // output valid ready
assign pipe_en   = frmvalrdy | (last_line & win_rdy); // Enable pipe at input valrdy
and at last line when data is recieved

assign set_eol = winvalrdy & (~frm_first_line)  & (window_cnt == 1);
assign set_sol = winvalrdy & (~|window_cnt) & ~fifo_empty & ~frm_first_line;

always@(posedge clk or negedge rst_n)
if(~rst_n                         ) set_initial_pop <= 1'b0; else
if(set_initial_pop & fifo_usedwords) set_initial_pop <= 1'b0; else
if(frmvalrdy & frm_sof            ) set_initial_pop <= 1'b1;

assign initial_pop = set_initial_pop | (frmvalrdy & frm_eol & frm_first_line &
(~last_line));

//------------------------------- Intermediate registers ---------------------------
-------

// Flag indicating the first input line, where no action is taken at the output
always@(posedge clk or negedge rst_n)
if(~rst_n   ) frm_sof_d <= 2'd0                  ; else
if(frmvalrdy) frm_sof_d <= {frm_sof_d[0],frm_sof};       // Reset first line flag
after first valid eol

always@(posedge clk or negedge rst_n)
if(~rst_n                 ) fifo_in_rst <= 1'd0; else
if(~(fifo_empty & fifo_full)) fifo_in_rst <= 1'd0; else
if(fifo_clr                ) fifo_in_rst <= 1'd1;      // Reset first line flag
after first valid eol

always@(posedge clk or negedge rst_n)
if(~rst_n                         ) frm_first_line <= 1'b1; else
if(sw_rst                         ) frm_first_line <= 1'b1; else
if(frmvalrdy & frm_sol & (~frm_sof)) frm_first_line <= 1'b0; else // Reset first
line flag after first valid eol
if(winvalrdy & win_eof            ) frm_first_line <= 1'b1; else
if(frmvalrdy & frm_sof             ) frm_first_line <= 1'b1;      // Set start of
frame flag at valid sof

always@(posedge clk or negedge rst_n)
if(~rst_n                         ) set_eof <= 1'b0; else
if(sw_rst                         ) set_eof <= 1'b0; else
if(win_eof & winvalrdy            ) set_eof <= 1'b0; else
if(last_line & winvalrdy & win_eol ) set_eof <= 1'b1;
```

```
// Flag for output last line, for emptying the fifo content
always@(posedge clk or negedge rst_n)
if(~rst_n             ) last_line <= 1'b0; else
if(sw_rst             ) last_line <= 1'b0; else
if(winvalrdy & win_eof) last_line <= 1'b0; else // Reset at sof
if(frmvalrdy & frm_eof) last_line <= 1'b1;      // Set at eof

always@(posedge clk or negedge rst_n)
if(~rst_n             ) win_last_line <= 1'b0; else
if(sw_rst             ) win_last_line <= 1'b0; else
if(winvalrdy & win_eof) win_last_line <= 1'b0; else // Reset at sof
if(last_line & set_eol) win_last_line <= 1'b1;      // Set at eof

always@(posedge clk or negedge rst_n)
if(~rst_n             ) win_first_line <= 1'b0; else
if(sw_rst             ) win_first_line <= 1'b0; else
if(winvalrdy & win_eol) win_first_line <= 1'b0; else // Reset at eol
if(frmvalrdy & win_sof) win_first_line <= 1'b1;      // Set at sof

always@(posedge clk or negedge rst_n)
if(~rst_n                                                  ) pix_in_line <=
{USEDW_BITS{1'd0}}              ; else
if(sw_rst                                                  ) pix_in_line <=
{USEDW_BITS{1'd0}}              ; else
if(frm_first_line  &  (~|pix_in_line)  &  frm_eol  &  frmvalrdy)  pix_in_line  <=
fifo_usedwords + fifo_push + 2'd2;       // Number of pixels in a line, compensate
the initial pop

always@(posedge clk or negedge rst_n)
if(~rst_n                           ) window_cnt <= 9'd0                       ;
else
if(sw_rst                           ) window_cnt <= 9'd0                       ;
else
if(frm_first_line & frm_eol & frmvalrdy) window_cnt <= fifo_usedwords + fifo_push +
2'd1; else // Load on first eol
if((~|window_cnt) & winvalrdy        ) window_cnt <= pix_in_line - 1'd1          ;
else // Load when not frm first line and the counter is 0
if(winvalrdy | (last_line & win_rdy)   ) window_cnt <= window_cnt - 1'b1          ;
// Decrement at each valid output

always@(posedge clk or negedge rst_n)
if(~rst_n             ) in_frm <= 1'd0; else
if(sw_rst             ) in_frm <= 1'd0; else
if(frm_eof & frmvalrdy) in_frm <= 1'd0; else // Reset at eof
if(frm_sof             ) in_frm <= 1'd1;      // Set in current frame flag when at
sof

always@(posedge clk or negedge rst_n)
if(~rst_n                         ) valrdy_cnt <= 2'd0                         ;
else
if(sw_rst                         ) valrdy_cnt <= 2'd0                         ;
else
if(frmvalrdy & frm_sof             ) valrdy_cnt <= 2'd0                         ;
else
```

```
if(~|valrdy_cnt & winvalrdy       ) valrdy_cnt <= 2'd0                              ;
else
if((~frm_first_line)  &  (~last_line))  valrdy_cnt  <=  valrdy_cnt  +  frmvalrdy  -
winvalrdy;        // Count the number of new elements in the 3 input registers,
increment when pipe is enabled decrement when data recieved

//Direct line pipe
always@(posedge clk or negedge rst_n)
if(~rst_n             ) line0_left <= {DATA_WIDTH{1'b0}}; else
if(sw_rst             ) line0_left <= {DATA_WIDTH{1'b0}}; else
if(win_eof & winvalrdy) line0_left <= cfg_bkg           ; else
if(pipe_en              ) line0_left <= frm_data           ;      // Delay input at
pipe_en

always@(posedge clk or negedge rst_n)
if(~rst_n             ) line0_mid <= {DATA_WIDTH{1'b0}}; else
if(sw_rst             ) line0_mid <= {DATA_WIDTH{1'b0}}; else
if(win_eof & winvalrdy) line0_mid <= {DATA_WIDTH{1'b0}}; else
if(pipe_en              ) line0_mid <= line0_left         ;      // Delay input at
pipe_en

//Second line pipe
always@(posedge clk or negedge rst_n)
if(~rst_n             ) line1_left <= {DATA_WIDTH{1'b0}}          ; else
if(sw_rst             ) line1_left <= {DATA_WIDTH{1'b0}}          ; else
if(win_eof & winvalrdy) line1_left <= cfg_bkg                     ; else
if(pipe_en            ) line1_left <= fifo_popdata[DATA_WIDTH-1:0];   // Delay input
at pipe_en

always@(posedge clk or negedge rst_n)
if(~rst_n             ) line1_mid <= {DATA_WIDTH{1'b0}}; else
if(sw_rst             ) line1_mid <= {DATA_WIDTH{1'b0}}; else
if(win_eof & winvalrdy) line1_mid <= {DATA_WIDTH{1'b0}}; else
if(pipe_en              ) line1_mid <= line1_left         ;       // Delay input at
pipe_en

  //Third line pie
always@(posedge clk or negedge rst_n)
if(~rst_n            ) line2_left <= {DATA_WIDTH{1'b0}}                          ; else
if(sw_rst            ) line2_left <= {DATA_WIDTH{1'b0}}                          ; else
if(win_eof & winvalrdy) line2_left <= cfg_bkg                                    ; else
if(pipe_en               )  line2_left  <=  fifo_popdata[2*DATA_WIDTH-1:  DATA_WIDTH];
// Delay input at pipe_en

always@(posedge clk or negedge rst_n)
if(~rst_n             ) line2_mid <= {DATA_WIDTH{1'b0}}; else
if(sw_rst             ) line2_mid <= {DATA_WIDTH{1'b0}}; else
if(win_eof & winvalrdy) line2_mid <= {DATA_WIDTH{1'b0}}; else
if(pipe_en              ) line2_mid <= line2_left         ;      // Delay input at
pipe_en

//------------------------------ fifo interface logic -----------------------------
-----
always@(posedge clk or negedge rst_n)
if(~rst_n                                    ) last_push <= 1'b0; else
if(sw_rst                                   ) last_push <= 1'b0; else
if( win_eof                                 ) last_push <= 1'b0; else //Concatenate
the middle register values
if((last_line   &   winvalrdy   &   (window_cnt   ==   2'd2)))   last_push   <=   1'b1;
//Concatenate the middle register values

always@(posedge clk or negedge rst_n)
if(~rst_n                                ) fifo_pushdata <= {(2*DATA_WIDTH){1'b0}}
; else
if(sw_rst                                 ) fifo_pushdata <= {(2*DATA_WIDTH){1'b0}}
; else
if(frmvalrdy | (last_line & winvalrdy & (window_cnt == 2'd2)))
fifo_pushdata <= {line1_left, line0_left}; //Concatenate the middle register values

// Pop signals
always@(posedge clk or negedge rst_n)
if(~rst_n                  ) fifo_pop <= 1'b0                                        ;
else
if(sw_rst                  ) fifo_pop <= 1'b0                                        ;
else
if(fifo_empty              ) fifo_pop <= 1'b0                                        ;
else // Reset when fifo is empty or is at the last element
if(~frm_first_line            ) fifo_pop <= frmvalrdy | initial_pop | (last_line &
win_rdy);  //Pop at first eol and sol with no sof to prepare the data

always@(posedge clk or negedge rst_n)
if(~rst_n                 ) fifo_push <= 1'b0     ; else
if(sw_rst                 ) fifo_push <= 1'b0     ; else
if(~(frm_sof  &  frmvalrdy))  fifo_push  <=  frmvalrdy  |  (last_line  &  winvalrdy  &
(window_cnt == 2'd2) & (~win_last_line));

always@(posedge clk or negedge rst_n)
if(~rst_n              ) fifo_clr <= 1'b0; else
if(fifo_clr            ) fifo_clr <= 1'b0; else
if(frm_sof & frmvalrdy) fifo_clr <= 1'b1;
//---------------------------- Output frame interface control signal logic -------
--------------------------
//Valid signal
always@(posedge clk or negedge rst_n)
if(~rst_n                        ) win_val <= 1'b0                                   ;
else
if(sw_rst                        ) win_val <= 1'b0                                   ;
else
if(frm_first_line     |     (winvalrdy     &     win_eof))     win_val     <=     1'b0
; else
if(last_line   &   (~fifo_empty)                               )   win_val   <=   1'b1
; else // Last line alway valid, no output dependence
                                           win_val  <=  (valrdy_cnt   +  frmvalrdy  -
winvalrdy) >= 2;      // Valid when the input 3 registers have 3 elements

// RDY
always@(posedge clk or negedge rst_n)
if(~rst_n                                                       )  frm_rdy  <=  1'b0
; else
```

```
if(fifo_in_rst  |  (frmvalrdy  &  frm_sof)  |  fifo_clr      )  frm_rdy  <=  1'b0
; else
if(last_line  |  (frmvalrdy  &  frm_eof)                            )  frm_rdy  <=  1'b0
; else // Set start of frame flag at valid sof
if(~((~fifo_full)  &  (~fifo_empty))                                )  frm_rdy  <=  1'b1
; else //Ready when fifo is not in reset state
if(frm_first_line  &  ~fifo_full                                    )  frm_rdy  <=  1'b1
; else //Ready when fifo is not in reset state
if(winvalrdy   &   win_eof   |   (~in_frm   &   ~frm_first_line))   frm_rdy   <=   1'b1
; else
                                                       frm_rdy   <=   (valrdy_cnt   +
frmvalrdy - winvalrdy) < 3; //Or the input registers are not populated

// SOL
always@(posedge clk or negedge rst_n)
if(~rst_n                                                  ) win_sol <= 1'b0; else
if(sw_rst                                                  ) win_sol <= 1'b0; else
if(winvalrdy & win_sol                                     ) win_sol <= 1'b0; else //
Reset after one valrdy
if(frmvalrdy & frm_sof                                     ) win_sol <= 1'b1; else //
Set at input sof
if(winvalrdy & win_eol & ((~frm_first_line) | (~win_eof))) win_sol <= 1'b1;      //
Set at window counter reset

// EOL
always@(posedge clk or negedge rst_n)
if(~rst_n                ) win_eol <= 1'b0; else
if(sw_rst                ) win_eol <= 1'b0; else
if(winvalrdy & win_eol   ) win_eol <= 1'b0; else // Reset after the one valrdy
if(set_eol               ) win_eol <= 1'b1;      // Set before window counter reset

// EOF
always@(posedge clk or negedge rst_n)
if(~rst_n              ) win_eof <= 1'b0; else
if(sw_rst              ) win_eof <= 1'b0; else
if(winvalrdy & win_eof) win_eof <= 1'b0; else // Reset after the one valrdy
if(set_eol & set_eof  ) win_eof <= 1'b1;      // Set at last line when fifo is empty

// SOF
always@(posedge clk or negedge rst_n)
if(~rst_n              ) win_sof <= 1'b0; else
if(sw_rst              ) win_sof <= 1'b0; else
if(winvalrdy & win_sof) win_sof <= 1'b0; else // Reset after sending last valid data
if(frmvalrdy & frm_sof) win_sof <= 1'b1;      // Set at last line when fifo is empty

//------------------------------ output frame interface data -----------------------
----------
always@(posedge clk or negedge rst_n)
if(~rst_n             ) mask_sol <= 1'b0; else
if(sw_rst             ) mask_sol <= 1'b0; else
if(pipe_en & mask_sol) mask_sol <= 1'b0; else // Reset after the one valrdy
if(set_sol | win_sof ) mask_sol <= 1'b1;      // Set at last line when fifo is empty

always@(posedge clk or negedge rst_n)
if(~rst_n             ) mask_eol <= 1'b0; else
if(sw_rst            ) mask_eol <= 1'b0; else
if(pipe_en & mask_eol) mask_eol <= 1'b0; else // Reset after the one valrdy
if(set_eol           ) mask_eol <= 1'b1;      // Set at last line when fifo is empty

//
//   cfg_bkg cfg_bkg cfg_bkg cfg_bkg cfg_bkg cfg_bkg cfg_bkg cfg_bkg
//   cfg_bkg   data    data    data    data    data    data  cfg_bkg
//   cfg_bkg   data    data    data    data    data    data  cfg_bkg
//   cfg_bkg   data    data    data    data    data    data  cfg_bkg
//   cfg_bkg   data    data    data    data    data    data  cfg_bkg
//   cfg_bkg   data    data    data    data    data    data  cfg_bkg
//   cfg_bkg   data    data    data    data    data    data  cfg_bkg
//   cfg_bkg   data    data    data    data    data    data  cfg_bkg
//   cfg_bkg cfg_bkg cfg_bkg cfg_bkg cfg_bkg cfg_bkg cfg_bkg cfg_bkg
//
always@(posedge clk or negedge rst_n)
if(~rst_n             ) win_data <= {(9*DATA_WIDTH){1'b0}} ;else
if(sw_rst             ) win_data <= {(9*DATA_WIDTH){1'b0}} ;else
if(frm_sof & frmvalrdy) win_data <= {9{cfg_bkg}}           ;else
if(pipe_en) win_data<={win_data[8*DATA_WIDTH-1:7*DATA_WIDTH],line2_mid, line2_left,
                       win_data[5*DATA_WIDTH-1:4*DATA_WIDTH],line1_mid, line1_left,
                       win_data[2*DATA_WIDTH-1:  DATA_WIDTH], line0_mid, line0_left
};else
if(win_last_line)begin
        if(set_sol) win_data <= {cfg_bkg , line2_mid, line2_left, //left-down corner
                                 cfg_bkg , line1_mid, line1_left,
                                  {3{cfg_bkg}}};else
       if(set_eol)win_data<={win_data[8*DATA_WIDTH-1:7*DATA_WIDTH],line2_mid,
cfg_bkg , //right-down corner
                        win_data[5*DATA_WIDTH-1:4*DATA_WIDTH],line1_mid,cfg_bkg ,
                        {3{cfg_bkg}}                                         };else

 win_data<={win_data[8*DATA_WIDTH-1:7*DATA_WIDTH],line2_mid,line2_left, // down row
            win_data[5*DATA_WIDTH-1:4*DATA_WIDTH], line1_mid, line1_left,
           {3{cfg_bkg}}                                         };end else
if(mask_sol) win_data <= {cfg_bkg, line2_mid, line2_left, // left column
                          cfg_bkg, line1_mid, line1_left,
                          cfg_bkg, line0_mid, line0_left }; else
//Mask right border
 if(mask_eol)win_data<={win_data[8*DATA_WIDTH-1:7*DATA_WIDTH],line2_mid,cfg_bkg,//
right column
                       win_data[5*DATA_WIDTH-1:4*DATA_WIDTH], line1_mid, cfg_bkg,
                     win_data[2*DATA_WIDTH-1: DATA_WIDTH],line0_mid,cfg_bkg }; else


win_data<={win_data[ 8*DATA_WIDTH-1:7*DATA_WIDTH], line2_mid, line2_left, // middle
            win_data[5*DATA_WIDTH-1:4*DATA_WIDTH], line1_mid, line1_left,
            win_data[ 2*DATA_WIDTH-1:  DATA_WIDTH], line0_mid, line0_left };

endmodule // line_buffer

// Project     : ir_filters
// Module Name : median_outer_4px
// Author      : Szilard Hegedus
// Created     : 11/15/2018
```

```
//-------------------------------------------------------------------------------
------------------
// Description : Connects Median outer for 3x3 window
//-------------------------------------------------------------------------------
------------------
// Modification history :
// 11/15/2018 (SH): Initial version
// 02/04/2019 (SH): Replaced 4 pixel/cycle to 1 pixel/cycle to integrate into
Pcam5c_demo reference design
//-------------------------------------------------------------------------------
------------------

module median_filter_1px#(
parameter DATA_WIDTH    = 8
)(
input                            clk       , // System clock
input                            rst_n     , // Asynchronous reset active low
//-----------------------------Input Frame Interface--------------------------------
input                           in3x3_val , // Master has valid data to be transferred
output                           in3x3_rdy , // Slave is ready to receive the data
input      [3*3*DATA_WIDTH-1:0] in3x3_data, // Data transferred from master to slave
input                            in3x3_sof , // Start of Frame
input                            in3x3_eof , // End of Frame
input                            in3x3_sol , // Start of Line
input                            in3x3_eol , // End of Line
//-----------------------------Output Frame Interface-------------------------------
output                          out_val   , // Master has valid data to be transferred
input                            out_rdy   , // Slave is ready to receive the data
output     [  DATA_WIDTH-1:0]   out_data  , // Data transferred from master to slave
output                           out_sof   , // Start of Frame
output                           out_eof   , // End of Frame
output                           out_sol   , // Start of Line
output                           out_eol     // End of Line
);

//-----------------------------Internal signals------------------------------------


wire hor_val ;
wire vert_val;

wire [3*DATA_WIDTH-1 : 0]vert0_data;
wire [3*DATA_WIDTH-1 : 0]vert1_data;
wire [3*DATA_WIDTH-1 : 0]vert2_data;

wire [3*DATA_WIDTH-1 : 0]hor00_data;
wire [3*DATA_WIDTH-1 : 0]hor01_data;
wire [3*DATA_WIDTH-1 : 0]hor02_data;


wire [3*DATA_WIDTH-1 : 0]diag0_data;

wire vert0_rdy;
wire vert1_rdy;
wire vert2_rdy;
wire hor0_rdy;

wire win1_sol;
wire win1_eol;
wire win1_sof;
wire win1_eof;
wire win0_sol;
wire win0_eol;
wire win0_sof;
wire win0_eof;


assign out_data = diag0_data[2*DATA_WIDTH-1:DATA_WIDTH];

//Verical sort
median_line_sort#(
  .DATA_WIDTH(DATA_WIDTH)
)vert0(
  .clk      (clk                                  ),
  .rst_n    (rst_n                                ),
  .pix2     (in3x3_data[9*DATA_WIDTH-1: 8*DATA_WIDTH]),
  .pix1     (in3x3_data[6*DATA_WIDTH-1: 5*DATA_WIDTH]),
  .pix0     (in3x3_data[3*DATA_WIDTH-1: 2*DATA_WIDTH]),
  .win_val  (in3x3_val                            ),
  .win_rdy  (in3x3_rdy                            ),
  .win_sol  (in3x3_sol                            ),
  .win_eol  (in3x3_eol                            ),
  .win_sof  (in3x3_sof                            ),
  .win_eof  (in3x3_eof                            ),
  .sort_val (vert_val                             ),
  .sort_rdy (vert2_rdy                            ),
  .sort_data(vert2_data                           ),
  .sort_sol (win0_sol                             ),
  .sort_eol (win0_eol                             ),
  .sort_sof (win0_sof                             ),
  .sort_eof (win0_eof                             )
);

median_line_sort#(
  .DATA_WIDTH(DATA_WIDTH)
)vert1(
  .clk      (clk                                  ),
  .rst_n    (rst_n                                ),
  .pix2     (in3x3_data[8*DATA_WIDTH-1: 7*DATA_WIDTH]),
  .pix1     (in3x3_data[5*DATA_WIDTH-1: 4*DATA_WIDTH]),
  .pix0     (in3x3_data[2*DATA_WIDTH-1:   DATA_WIDTH]),
  .win_val  (in3x3_val                            ),
  .win_rdy  (                                     ),
  .win_sol  (in3x3_sol                            ),
  .win_eol  (in3x3_eol                            ),
  .win_sof  (in3x3_sof                            ),
  .win_eof  (in3x3_eof                            ),
  .sort_val (                                     ),
  .sort_rdy (vert1_rdy                            ),
  .sort_data(vert1_data                           ),
  .sort_sol (                                        ),
  .sort_eol (                                        ),
  .sort_sof (                                        ),
  .sort_eof (                                        )
);

median_line_sort#(
  .DATA_WIDTH(DATA_WIDTH)
)vert2(
  .clk      (clk                                     ),
  .rst_n    (rst_n                                   ),
  .pix2     (in3x3_data[7*DATA_WIDTH-1: 6*DATA_WIDTH]),
  .pix1     (in3x3_data[4*DATA_WIDTH-1: 3*DATA_WIDTH]),
  .pix0     (in3x3_data[  DATA_WIDTH-1:            0]),
  .win_val  (in3x3_val                               ),
  .win_rdy  (                                        ),
  .win_sol  (in3x3_sol                               ),
  .win_eol  (in3x3_eol                               ),
  .win_sof  (in3x3_sof                               ),
  .win_eof  (in3x3_eof                               ),
  .sort_val (                                        ),
  .sort_rdy (vert0_rdy                               ),
  .sort_data(vert0_data                              ),
  .sort_sol (                                        ),
  .sort_eol (                                        ),
  .sort_sof (                                        ),
  .sort_eof (                                        )
);

//Horizontal sort
median_line_sort#(
  .DATA_WIDTH(DATA_WIDTH)
)hor00(
  .clk      (clk                                    ),
  .rst_n    (rst_n                                  ),
  .pix2     (vert0_data[3*DATA_WIDTH-1:2*DATA_WIDTH]),
  .pix1     (vert1_data[3*DATA_WIDTH-1:2*DATA_WIDTH]),
  .pix0     (vert2_data[3*DATA_WIDTH-1:2*DATA_WIDTH]),
  .win_val  (vert_val                               ),
  .win_rdy  (vert0_rdy                              ),
  .win_sol  (win0_sol                               ),
  .win_eol  (win0_eol                               ),
  .win_sof  (win0_sof                               ),
  .win_eof  (win0_eof                               ),
  .sort_val (hor_val                                ),
  .sort_rdy (hor0_rdy                               ),
  .sort_data(hor00_data                             ),
  .sort_sol (win1_sol                               ),
  .sort_eol (win1_eol                               ),
  .sort_sof (win1_sof                               ),
  .sort_eof (win1_eof                               )
);

median_line_sort#(
  .DATA_WIDTH(DATA_WIDTH)
```

```
)hor01(
  .clk      (clk                              ),
  .rst_n    (rst_n                            ),
  .pix0     (vert0_data[2*DATA_WIDTH-1:DATA_WIDTH]),
  .pix1     (vert1_data[2*DATA_WIDTH-1:DATA_WIDTH]),
  .pix2     (vert2_data[2*DATA_WIDTH-1:DATA_WIDTH]),
  .win_val  (vert_val                         ),
  .win_rdy  (vert1_rdy                        ),
  .win_sol  (win0_sol                         ),
  .win_eol  (win0_eol                         ),
  .win_sof  (win0_sof                         ),
  .win_eof  (win0_eof                         ),
  .sort_val (                                 ),
  .sort_rdy (hor0_rdy                         ),
  .sort_data(hor01_data                       ),
  .sort_sol (                                 ),
  .sort_eol (                                 ),
  .sort_sof (                                 ),
  .sort_eof (                                 )
);

median_line_sort#(
  .DATA_WIDTH(DATA_WIDTH)
)hor02(
  .clk      (clk                   ),
  .rst_n    (rst_n                 ),
  .pix0     (vert0_data[DATA_WIDTH-1:0]),
  .pix1     (vert1_data[DATA_WIDTH-1:0]),
  .pix2     (vert2_data[DATA_WIDTH-1:0]),
  .win_val  (vert_val              ),
  .win_rdy  (vert2_rdy             ),
  .win_sol  (win0_sol              ),
  .win_eol  (win0_eol              ),
  .win_sof  (win0_sof              ),
  .win_eof  (win0_eof              ),
  .sort_val (                      ),
  .sort_rdy (hor0_rdy              ),
  .sort_data(hor02_data            ),
  .sort_sol (                      ),
  .sort_eol (                      ),
  .sort_sof (                      ),
  .sort_eof (                      )
);

// Diagonal sort
median_line_sort#(
  .DATA_WIDTH(DATA_WIDTH)
)diag0(
  .clk      (clk                                ),
  .rst_n    (rst_n                              ),
  .pix2     (hor00_data[  DATA_WIDTH-1:          0]),
  .pix1     (hor01_data[2*DATA_WIDTH-1:  DATA_WIDTH]),
  .pix0     (hor02_data[3*DATA_WIDTH-1:2*DATA_WIDTH]),
  .win_val  (hor_val                            ),
  .win_rdy  (hor0_rdy                           ),
  .win_sol  (win1_sol                               ),
  .win_eol  (win1_eol                               ),
  .win_sof  (win1_sof                               ),
  .win_eof  (win1_eof                               ),
  .sort_val (out_val                                ),
  .sort_rdy (out_rdy                                ),
  .sort_data(diag0_data                             ),
  .sort_sol (out_sol                                ),
  .sort_eol (out_eol                                ),
  .sort_sof (out_sof                                ),
  .sort_eof (out_eof                                )
);

Endmodule
// Project     : ir_filters
// Module Name : median_line_sort
// Author      : Szilard Hegedus
// Created     : 11/21/2018
//--------------------------------------------------------------------------------
// Description : Connects 4 pix_corr_1px modules to achieve 4px output
//--------------------------------------------------------------------------------
//              ______                      ______
// pix0 ---->|      |------------------->|      |---sort_sort[23:16]--->
//           | Comp |        ______      | Comp |
// pix1 ---->|______|---->|      |------>|______|---sort_sort[15: 8]--->
//                        | Comp |
// pix2------------------>|______|------------------sort_sort[8 : 0]--->
//
//
// Modification history :
// 11/15/2018 (SH): Initial version
//--------------------------------------------------------------------------------

module median_line_sort#(
parameter DATA_WIDTH =  8
)(
input                         clk      ,
input                         rst_n    ,
input      [DATA_WIDTH - 1:0] pix0     ,
input      [DATA_WIDTH - 1:0] pix1     ,
input      [DATA_WIDTH - 1:0] pix2     ,
input                         win_val  ,
output                        win_rdy  ,
input                         win_sol  ,
input                         win_eol  ,
input                         win_sof  ,
input                         win_eof  ,
output reg                    sort_val ,
input                         sort_rdy ,
output reg                    sort_sol ,
output reg                    sort_eol ,
output reg                    sort_sof ,
output reg                    sort_eof ,
output reg [3*DATA_WIDTH-1:0] sort_data
);
```

```
wire invalrdy;

wire [DATA_WIDTH-1:0] comp0_max;
wire [DATA_WIDTH-1:0] comp1_max;
wire [DATA_WIDTH-1:0] comp2_max;

wire [DATA_WIDTH-1:0] comp0_min;
wire [DATA_WIDTH-1:0] comp1_min;
wire [DATA_WIDTH-1:0] comp2_min;

assign win_rdy = sort_rdy;

assign invalrdy = win_val & win_rdy;

// Assign maximum and minimum values
assign {comp0_max, comp0_min} = (pix0      > pix1     ) ? {pix0     , pix1     } :
{pix1     , pix0      };
assign {comp1_max, comp1_min} = (comp0_min > pix2     ) ? {comp0_min, pix2     } :
{pix2     , comp0_min };
assign {comp2_max, comp2_min} = (comp0_max > comp1_max) ? {comp0_max, comp1_max} :
{comp0_max, comp1_max };

//Create data
always@(posedge clk or negedge rst_n)
if(~rst_n  ) sort_data <= {DATA_WIDTH{1'b0}}               ; else
if(invalrdy) sort_data <= {comp2_max, comp2_min, comp1_min};

//Control signals
always@(posedge clk or negedge rst_n)
if(~rst_n                        ) sort_eof <= 1'b0; else
if(sort_rdy & sort_val & sort_eof) sort_eof <= 1'b0; else
if(win_eof & win_val & win_rdy   ) sort_eof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) sort_sof <= 1'b0; else
if(sort_rdy & sort_val & sort_sof) sort_sof <= 1'b0; else
if(win_sof & win_val & win_rdy   ) sort_sof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) sort_eol <= 1'b0; else
if(sort_rdy & sort_val & sort_eol) sort_eol <= 1'b0; else
if(win_eol & win_val & win_rdy   ) sort_eol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) sort_sol <= 1'b0; else
if(sort_rdy & sort_val & sort_sol) sort_sol <= 1'b0; else
if((win_sol & win_val & win_rdy) ) sort_sol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n               ) sort_val <= 1'b0; else
if(sort_rdy & (~win_val)) sort_val <= 1'b0; else
if(invalrdy             ) sort_val <= 1'b1;
```

```
endmodule

// Project     : ir_filters
// Module Name : pix_corr_1px
// Author      : Szilard Hegedus
// Created     : 11/15/2018
//--------------------------------------------------------------------------------
// Description : Corrects dead pixels in a 3x3 window
//
//--------------------------------------------------------------------------------
// Modification history :
// 11/15/2018 (SH): Initial version
//  02/04/2019  (SH):  Replaced  4  pixel/cycle  to  1  pixel/cycle  to  integrate  into
Pcam5c_demo reference design
//--------------------------------------------------------------------------------

module pix_corr_1px#(
parameter DATA_WIDTH = 8
)(
input                         clk       ,
input                         rst_n     ,
input      [DATA_WIDTH-1:0]   cfg_thr   ,
input                        in3x3_val , // Master has valid data to be transferred
output                        in3x3_rdy , // Slave is ready to receive the data
input      [9*DATA_WIDTH-1:0] in3x3_data, // Data transferred from master to slave
input                         in3x3_sof , // Start of frame
input                         in3x3_sol , // Start of line
input                         in3x3_eol , // End of line
input                         in3x3_eof , // End of frame
//-----------------------------Output Frame Interface------------------------------
output reg                    out_val , // Master has valid data to be transferred
input                         out_rdy , // Slave is ready to receive the data
output reg [  DATA_WIDTH-1:0] out_data, // Data transferred from master to slave
output reg                    out_sof , // Start of frame
output reg                    out_sol , // Start of line
output reg                    out_eol , // End of line
output reg                    out_eof   // End of frame
);

//-----------------------------Internal signals-----------------------------------
wire [DATA_WIDTH-1:0] p00; //Pixel in window
wire [DATA_WIDTH-1:0] p01; //Pixel in window
wire [DATA_WIDTH-1:0] p02; //Pixel in window
wire [DATA_WIDTH-1:0] p10; //Pixel in window
wire [DATA_WIDTH-1:0] p11; //Pixel in window
wire [DATA_WIDTH-1:0] p12; //Pixel in window
wire [DATA_WIDTH-1:0] p20; //Pixel in window
wire [DATA_WIDTH-1:0] p21; //Pixel in window
wire [DATA_WIDTH-1:0] p22; //Pixel in window

wire [DATA_WIDTH-1:0] max00;
wire [DATA_WIDTH-1:0] max01;
wire [DATA_WIDTH-1:0] max02;
wire [DATA_WIDTH-1:0] max10;
wire [DATA_WIDTH-1:0] max12;
wire [DATA_WIDTH-1:0] max20;
wire [DATA_WIDTH-1:0] max21;
wire [DATA_WIDTH-1:0] max22;

wire [DATA_WIDTH-1:0] min00;
wire [DATA_WIDTH-1:0] min01;
wire [DATA_WIDTH-1:0] min02;
wire [DATA_WIDTH-1:0] min10;
wire [DATA_WIDTH-1:0] min12;
wire [DATA_WIDTH-1:0] min20;
wire [DATA_WIDTH-1:0] min21;
wire [DATA_WIDTH-1:0] min22;

wire [DATA_WIDTH-1 : 0] diff00;
wire [DATA_WIDTH-1 : 0] diff01;
wire [DATA_WIDTH-1 : 0] diff02;
wire [DATA_WIDTH-1 : 0] diff10;
wire [DATA_WIDTH-1 : 0] diff12;
wire [DATA_WIDTH-1 : 0] diff20;
wire [DATA_WIDTH-1 : 0] diff21;
wire [DATA_WIDTH-1 : 0] diff22;

wire [DATA_WIDTH + 3:0] sum;
wire mux_sel;

wire invalrdy;

assign invalrdy =  in3x3_rdy & in3x3_val;
assign in3x3_rdy = out_rdy;

assign p00 = in3x3_data[9*DATA_WIDTH-1:8*DATA_WIDTH];
assign p01 = in3x3_data[8*DATA_WIDTH-1:7*DATA_WIDTH];
assign p02 = in3x3_data[7*DATA_WIDTH-1:6*DATA_WIDTH];
assign p10 = in3x3_data[6*DATA_WIDTH-1:5*DATA_WIDTH];
assign p11 = in3x3_data[5*DATA_WIDTH-1:4*DATA_WIDTH];
assign p12 = in3x3_data[4*DATA_WIDTH-1:3*DATA_WIDTH];
assign p20 = in3x3_data[3*DATA_WIDTH-1:2*DATA_WIDTH];
assign p21 = in3x3_data[2*DATA_WIDTH-1:1*DATA_WIDTH];
assign p22 = in3x3_data[1*DATA_WIDTH-1:0*DATA_WIDTH];

assign sum = p00 + p01 + p02 + p10 + p12 + p20 + p21 + p22;

assign {max00, min00} = (p00 > p11) ? {p00, p11} : {p11, p00};
assign {max01, min01} = (p01 > p11) ? {p01, p11} : {p11, p01};
assign {max02, min02} = (p02 > p11) ? {p02, p11} : {p11, p02};
assign {max10, min10} = (p10 > p11) ? {p10, p11} : {p11, p10};
assign {max12, min12} = (p12 > p11) ? {p12, p11} : {p11, p12};
assign {max20, min20} = (p20 > p11) ? {p20, p11} : {p11, p20};
assign {max21, min21} = (p21 > p11) ? {p21, p11} : {p11, p21};
assign {max22, min22} = (p22 > p11) ? {p22, p11} : {p11, p22};


assign diff00 = max00 - min00;
assign diff01 = max01 - min01;
assign diff02 = max02 - min02;
assign diff10 = max10 - min10;
assign diff12 = max12 - min12;
assign diff20 = max20 - min20;
assign diff21 = max21 - min21;
assign diff22 = max22 - min22;

assign mux_sel = (diff00 > cfg_thr) & (diff01 > cfg_thr) & (diff02 > cfg_thr) &
(diff10 > cfg_thr) & (diff12 > cfg_thr) & (diff20 > cfg_thr) & (diff21 > cfg_thr) &
(diff22 > cfg_thr);

always@(posedge clk or negedge rst_n)
if(~rst_n              ) out_data <= 8'd0                               ; else
if(in3x3_val & in3x3_rdy) out_data <= mux_sel ? sum[DATA_WIDTH+2:3] : p11;      //
Recieve only the top 8 pixels, that will be the result of division by 16

always@(posedge clk or negedge rst_n)
if(~rst_n                          ) out_eof <= 1'b0; else
if(out_rdy & out_val & out_eof     ) out_eof <= 1'b0; else
if(in3x3_eof & in3x3_val & in3x3_rdy) out_eof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                          ) out_sof <= 1'b0; else
if(out_rdy & out_val & out_sof     ) out_sof <= 1'b0; else
if(in3x3_sof & in3x3_val & in3x3_rdy) out_sof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                          ) out_eol <= 1'b0; else
if(out_rdy & out_val & out_eol     ) out_eol <= 1'b0; else
if(in3x3_eol & in3x3_val & in3x3_rdy) out_eol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                                                                  ) out_sol <=
1'b0; else
if(out_rdy & out_val & out_sol                                             ) out_sol <=
1'b0; else
if((in3x3_sol & in3x3_val & in3x3_rdy) | (out_rdy & out_val & out_eol)) out_sol <=
1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                ) out_val <= 1'b0; else
if(out_rdy & (~in3x3_val)) out_val <= 1'b0; else
if(invalrdy              ) out_val <= 1'b1;



endmodule

//--------------------------------------------------------------------------------
// Project     : IR_filters
// Module Name : selector_2i
// Author      : SZILARD HEGEDUS
// Created     : 10/29/2018
//--------------------------------------------------------------------------------
// Description : MUX2to1 with Frame interface input, and output
```

```
//---------------------------------------------------------------------------------
------------------
// Modification history :
// 01/28/2019 (SH):Initial version
//---------------------------------------------------------------------------------

module selector_2i#(
  parameter DATA_WIDTH    = 8
)(
  input                        clk         , // System clock
  input                        rst_n       , // Asynchronous reset active low
  input                        sel         , // Mux selection bit
//--------------------------------Input frame interface---------------------------
  input                      in0_frm_val , // Master has valid data to be transferred
  output                       in0_frm_rdy , // Slave is ready to receive the data
  input     [DATA_WIDTH-1:0] in0_frm_data, // Data transferred from master to slave
  input                        in0_frm_sof , // Start of Frame
  input                        in0_frm_eof , // End of Frame
  input                        in0_frm_sol , // Start of Line
  input                        in0_frm_eol , // End of Line

  input                      in1_frm_val , // Master has valid data to be transferred
  output                       in1_frm_rdy , // Slave is ready to receive the data
  input     [DATA_WIDTH-1:0] in1_frm_data, // Data transferred from master to slave
  input                        in1_frm_sof , // Start of Frame
  input                        in1_frm_eof , // End of Frame
  input                        in1_frm_sol , // Start of Line
  input                        in1_frm_eol , // End of Line
//--------------------------------Output frame interface--------------------------
  output                     out_frm_val , // Master has valid data to be transferred
  input                        out_frm_rdy , // Slave is ready to receive the data
  output [DATA_WIDTH-1:0]      out_frm_data, // Data transferred from master to slave
  output                       out_frm_sof , // Start of Frame
  output                       out_frm_eof , // End of Frame
  output                       out_frm_sol , // Start of Line
  output                       out_frm_eol   // End of Line
);


assign out_frm_val  = sel ? in1_frm_val : in0_frm_val;
assign out_frm_sol  = sel ? in1_frm_sol : in0_frm_sol;
assign out_frm_eol  = sel ? in1_frm_eol : in0_frm_eol;
assign out_frm_sof  = sel ? in1_frm_sof : in0_frm_sof;
assign out_frm_eof  = sel ? in1_frm_eof : in0_frm_eof;
assign out_frm_data = sel ? in1_frm_data : in0_frm_data;
assign in0_frm_rdy  = sel ? 1'b0 : out_frm_rdy;
assign in1_frm_rdy  = ~sel ? 1'b0 : out_frm_rdy;


endmodule //selector_2i


// Project     : ir_outers
// Module Name : laplace_filter_1px
// Author      : Szilard Hegedus
// Created     : 11/15/2018
//--------------------------------------------------------------------------------
------------------
// Description : Connects laplace_outer_1px modules and calculates sharpened image
//--------------------------------------------------------------------------------
------------------
// Modification history :
// 11/15/2018 (SH): Initial version
// 02/04/2019 (SH): Replaced 4 pixel/cycle to 1 pixel/cycle to integrate into
Pcam5c_demo reference design
//--------------------------------------------------------------------------------
------------------

module sharp_filter_1px#(
parameter DATA_WIDTH    = 8
)(
input                           clk       , // System clock
input                           rst_n     , // Asynchronous reset active low
input    [7:0]                  cfg_coef  ,
//-----------------------------Input Frame Interface--------------------------------
---------
input                          in3x3_val , // Master has valid data to be transferred
output                          in3x3_rdy , // Slave is ready to receive the data
input     [9*DATA_WIDTH-1:0]    in3x3_data, // Data transferred from master to slave
input                           in3x3_sof , // Start of Frame
input                           in3x3_eof , // End of Frame
input                           in3x3_sol , // Start of Line
input                           in3x3_eol , // End of Line
//-----------------------------Output Frame Interface-------------------------------
----------
output reg                     out_val  , // Master has valid data to be transferred
input                           out_rdy  , // Slave is ready to receive the data
output reg [  DATA_WIDTH-1:0]   out_data , // Data transferred from master to slave
output reg                      out_sof  , // Start of Frame
output reg                      out_eof  , // End of Frame
output reg                      out_sol  , // Start of Line
output reg                      out_eol    // End of Line
);

//-----------------------------Internal signals-------------------------------------
----------
reg [DATA_WIDTH-1:0] in_data_d;

wire                   lap_val ;
wire [DATA_WIDTH-1:0]  lap_data;
wire                   lap_sof ;
wire                   lap_eof ;
wire                   lap_sol ;
wire                   lap_eol ;

wire lap_rdy;
wire lap_rdy_d;
wire invalrdy;
wire lap_valrdy;
```

```
reg  [DATA_WIDTH:0] out_data_temp;
wire [  DATA_WIDTH-1:0] norm_data;
wire [2*DATA_WIDTH-1:0] mult_data;

reg                   lap_val_d ;
reg                   lap_sof_d ;
reg                   lap_eof_d ;
reg                   lap_sol_d ;
reg                   lap_eol_d ;


assign in3x3_rdy = out_rdy;

assign invalrdy = in3x3_rdy & in3x3_val;
assign lap_rdy  = out_rdy;
assign lap_rdy_d  = lap_rdy;

assign lap_valrdy = lap_val & lap_rdy;

assign lap_valrdy_d = lap_val_d & lap_rdy_d;

assign mult_data = lap_data * cfg_coef; // Multiply the data with the coeficcient

assign   norm_data    =    (out_data_temp    >    {1'b0,{(DATA_WIDTH){1'd1}}})    ?
{(DATA_WIDTH){1'd1}} : out_data_temp; // Normalize output data

//--------------------------------------------------- Pipe stage 0 ------------------
---------------------------------------------------

laplace_filter_1px#(
  .DATA_WIDTH(DATA_WIDTH)
)laplace_out(
  .clk       (clk       ), // System clock
  .rst_n     (rst_n     ), // Asynchronous reset active low
  .in3x3_val (in3x3_val ), // Master has valid data to be transferred
  .in3x3_rdy (          ), // Slave is ready to receive the data
  .in3x3_data(in3x3_data), // Data transferred from master to slave
  .in3x3_sof (in3x3_sof ), // Start of Frame
  .in3x3_eof (in3x3_eof ), // End of Frame
  .in3x3_sol (in3x3_sol ), // Start of Line
  .in3x3_eol (in3x3_eol ), // End of Line
  .out_val   (lap_val   ), // Master has valid data to be transferred
  .out_rdy   (lap_rdy   ), // Slave is ready to receive the data
  .out_data  (lap_data  ), // Data transferred from master to slave
  .out_sof   (lap_sof   ), // Start of Frame
  .out_eof   (lap_eof   ), // End of Frame
  .out_sol   (lap_sol   ), // Start of Line
  .out_eol   (lap_eol   )  // End of Line
);

always@(posedge clk or negedge rst_n)
if(~rst_n                ) in_data_d <= {(DATA_WIDTH){1'd0}}                    ;
else
if(in3x3_rdy & in3x3_val) in_data_d <= in3x3_data[5*DATA_WIDTH-1 : 4*DATA_WIDTH];
// Delay input data that will be added to the mask
```

```
//----------------------------------------------- Pipe stage 1 ---------------------
--------------------------------------------------
always@(posedge clk or negedge rst_n)
if(~rst_n              ) lap_val_d <= 1'b0; else
if(lap_rdy & (~lap_val)) lap_val_d <= 1'b0; else
if(lap_valrdy          ) lap_val_d <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) lap_sof_d <= 1'b0; else
if(out_rdy & out_val & lap_sof_d) lap_sof_d <= 1'b0; else
if(lap_valrdy & lap_sof         ) lap_sof_d <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) lap_eof_d <= 1'b0; else
if(out_rdy & out_val & lap_eof_d) lap_eof_d <= 1'b0; else
if(lap_valrdy & lap_eof          ) lap_eof_d <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) lap_sol_d <= 1'b0; else
if(out_rdy & out_val & lap_sol_d) lap_sol_d <= 1'b0; else
if(lap_valrdy & lap_sol         ) lap_sol_d <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                        ) lap_eol_d <= 1'b0; else
if(out_rdy & out_val & lap_eol_d) lap_eol_d <= 1'b0; else
if(lap_valrdy & lap_eol         ) lap_eol_d <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n    ) out_data_temp <= {(DATA_WIDTH + 1){1'b0}}                        ;
else
if(lap_valrdy) out_data_temp <= mult_data[2*DATA_WIDTH-1:DATA_WIDTH] + in_data_d;

//-------------------------------------------- Pipe stage 2 ------------------------
------------------------------------------------
always@(posedge clk or negedge rst_n)
if(~rst_n                      ) out_eof <= 1'b0; else
if(out_rdy & out_val & out_eof) out_eof <= 1'b0; else
if(lap_eof_d & lap_valrdy_d   ) out_eof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                      ) out_sof <= 1'b0; else
if(out_rdy & out_val & out_sof) out_sof <= 1'b0; else
if(lap_sof_d & lap_valrdy_d   ) out_sof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                      ) out_eol <= 1'b0; else
if(out_rdy & out_val & out_eol) out_eol <= 1'b0; else
if(lap_eol_d & lap_valrdy_d   ) out_eol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                      ) out_sol <= 1'b0; else
if(out_rdy & out_val & out_sol) out_sol <= 1'b0; else
if(lap_sol_d & lap_valrdy_d   ) out_sol <= 1'b1;
```

```
always@(posedge clk or negedge rst_n)
if(~rst_n                ) out_val <= 1'b0; else
if(out_rdy & (~lap_val_d)) out_val <= 1'b0; else
if(lap_valrdy_d          ) out_val <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n      ) out_data <= {(DATA_WIDTH){1'd0}}; else
if(lap_valrdy_d) out_data <= norm_data           ;

`ifdef DEBUG_ON
  `include "sharp_filter_debug.v"
`endif


Endmodule

// Project     : ir_filters
// Module Name : smooth_filter_1px
// Author      : Szilard Hegedus
// Created     : 10/26/2018
//---------------------------------------------------------------------------------
------------------
// Description : Applies 3x3 smoothing filter
//                ___________
//               |   |   |   |
//               | 1 | 2 | 1 |
//               |___|___|___|
//          1    |   |   |   |
//         -- x  | 2 | 4 | 2 |
//         16    |___|___|___|
//               |   |   |   |
//               | 1 | 2 | 1 |
//               |___|___|___|
//
//---------------------------------------------------------------------------------
------------------
// Modification history :
// 10/26/2018 (SH): Initial version
// 02/04/2019 (SH): Replaced 4 pixel/cycle to 1 pixel/cycle to integrate into
Pcam5c_demo reference design
//---------------------------------------------------------------------------------
------------------

module smooth_filter_1px#(
parameter DATA_WIDTH = 8
)(
input                         clk       ,
input                         rst_n     ,
input                        in3x3_val , // Master has valid data to be transferred
output                        in3x3_rdy , // Slave is ready to receive the data
input      [9*DATA_WIDTH-1:0] in3x3_data, // Data transferred from master to slave
input                         in3x3_sof , // Start of frame
input                         in3x3_sol , // Start of line
input                         in3x3_eol , // End of line
input                         in3x3_eof , // End of frame
//----------------------------Output Frame Interface--------------------------------
----------
output reg                    out_val , // Master has valid data to be transferred
input                         out_rdy , // Slave is ready to receive the data
output reg [  DATA_WIDTH-1:0] out_data, // Data transferred from master to slave
output reg                    out_sof , // Start of frame
output reg                    out_sol , // Start of line
output reg                    out_eol , // End of line
output reg                    out_eof   // End of frame
);

//-----------------------------Internal signals------------------------------------
----------
wire [DATA_WIDTH-1:0] p00; //Pixel in window
wire [DATA_WIDTH-1:0] p01; //Pixel in window
wire [DATA_WIDTH-1:0] p02; //Pixel in window
wire [DATA_WIDTH-1:0] p10; //Pixel in window
wire [DATA_WIDTH-1:0] p11; //Pixel in window
wire [DATA_WIDTH-1:0] p12; //Pixel in window
wire [DATA_WIDTH-1:0] p20; //Pixel in window
wire [DATA_WIDTH-1:0] p21; //Pixel in window
wire [DATA_WIDTH-1:0] p22; //Pixel in window

wire [DATA_WIDTH+4:0] sum;

wire invalrdy;

assign invalrdy =  in3x3_rdy & in3x3_val;
assign in3x3_rdy = out_rdy;

assign p00 = in3x3_data[9*DATA_WIDTH-1:8*DATA_WIDTH];
assign p01 = in3x3_data[8*DATA_WIDTH-1:7*DATA_WIDTH];
assign p02 = in3x3_data[7*DATA_WIDTH-1:6*DATA_WIDTH];
assign p10 = in3x3_data[6*DATA_WIDTH-1:5*DATA_WIDTH];
assign p11 = in3x3_data[5*DATA_WIDTH-1:4*DATA_WIDTH];
assign p12 = in3x3_data[4*DATA_WIDTH-1:3*DATA_WIDTH];
assign p20 = in3x3_data[3*DATA_WIDTH-1:2*DATA_WIDTH];
assign p21 = in3x3_data[2*DATA_WIDTH-1:1*DATA_WIDTH];
assign p22 = in3x3_data[1*DATA_WIDTH-1:0*DATA_WIDTH];

assign sum =  p00        + {p01, 1'd0} + p02         +
             {p10, 1'b0} + {p11, 2'b0} + {p12, 1'b0} +
                    p20        + {p21, 1'b0} + p22;

always@(posedge clk or negedge rst_n)
if(~rst_n               ) out_data <= 8'd0                                         ;
else
if(in3x3_val & in3x3_rdy) out_data <= (sum[12:4] > 8'd255) ?  8'd255 : sum[11:4];
// Recieve only the top 8 pixels, that will be the result of division by 16

always@(posedge clk or negedge rst_n)
if(~rst_n                          ) out_eof <= 1'b0; else
if(out_rdy & out_val & out_eof     ) out_eof <= 1'b0; else
if(in3x3_eof & in3x3_val & in3x3_rdy) out_eof <= 1'b1;
```

```
always@(posedge clk or negedge rst_n)
if(~rst_n                           ) out_sof <= 1'b0; else
if(out_rdy & out_val & out_sof      ) out_sof <= 1'b0; else
if(in3x3_sof & in3x3_val & in3x3_rdy) out_sof <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                           ) out_eol <= 1'b0; else
if(out_rdy & out_val & out_eol      ) out_eol <= 1'b0; else
if(in3x3_eol & in3x3_val & in3x3_rdy) out_eol <= 1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                                                                   ) out_sol <=
1'b0; else
if(out_rdy & out_val & out_sol                                              ) out_sol <=
1'b0; else
if((in3x3_sol & in3x3_val & in3x3_rdy) | (out_rdy & out_val & out_eol)) out_sol <=
1'b1;

always@(posedge clk or negedge rst_n)
if(~rst_n                ) out_val <= 1'b0; else
if(out_rdy & (~in3x3_val)) out_val <= 1'b0; else
if(invalrdy              ) out_val <= 1'b1;

endmodule
```