**Infra-red Based Image Processing**

**Szilárd Hegedűs**

**szilard.hegedus@student.unitbv.ro**

**Submitted for the 2019 Digilent Design Contest Europe**

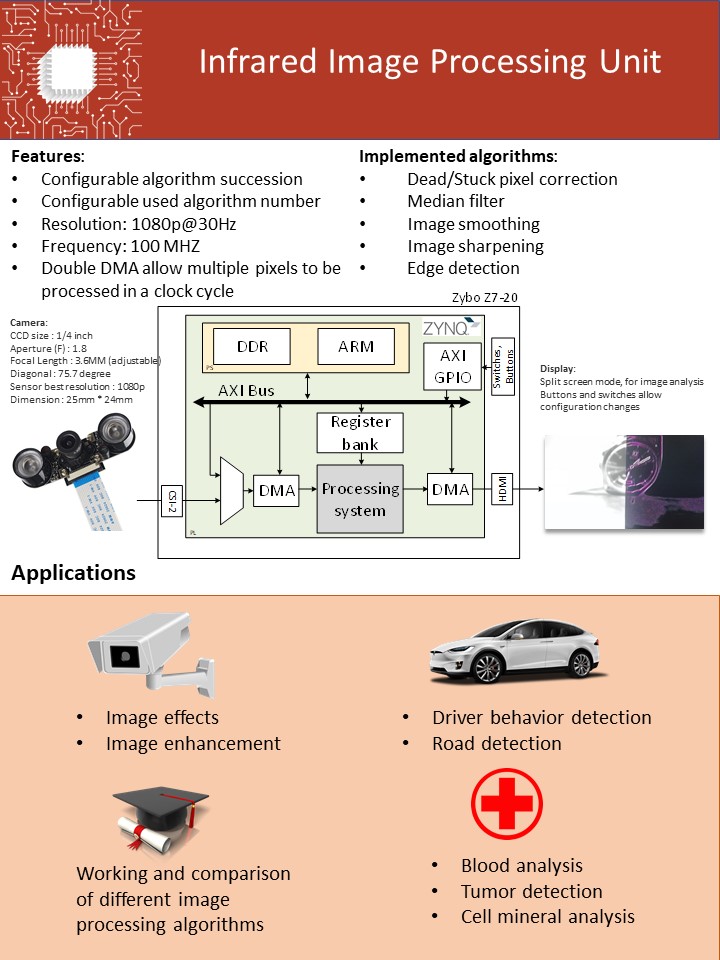
**May 3, 2019**

**Advisor: Dan Nicula**

**Transilvania University**

**Brasov, Romania**

****



**Contents**

[Introduction 4](#_Toc7791701)

[Background 5](#_Toc7791702)

[Design 7](#_Toc7791703)

[Features and Specifications 7](#_Toc7791704)

[Design Overview 10](#_Toc7791705)

[Interfaces 13](#_Toc7791706)

[Detailed Design Description 16](#_Toc7791707)

[Module Description 18](#_Toc7791708)

[Line Buffer Module 19](#_Toc7791709)

[Smoothing filter module 21](#_Toc7791710)

[Dead pixel correction module 23](#_Toc7791711)

[Laplace filter 27](#_Toc7791712)

[Sharpening filter 29](#_Toc7791713)

[Median filter module 31](#_Toc7791714)

[Selector module 34](#_Toc7791715)

[AXI2FRAME 35](#_Toc7791716)

[Module Description 39](#_Toc7791717)

[FIFO2FRM\_3MAP Module 42](#_Toc7791718)

[STS\_INTERRUPTS module 45](#_Toc7791719)

[STS\_INTERRUPTS 45](#_Toc7791720)

[Software 48](#_Toc7791721)

[Interrupt Handlers 48](#_Toc7791722)

[Camera configuration 50](#_Toc7791723)

[Filter configuration 51](#_Toc7791724)

[Discussion 53](#_Toc7791725)

[References 54](#_Toc7791726)

[Appendix A: Name of Source Code Files HDL 55](#_Toc7791727)

# 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

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

1. Configurations can be changed only when block is disabled
2. Support: Maximum Image width 2048 pixels; minimum image width 4 pixel
3. 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 2
4. 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
5. **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

1. 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

1. Switch configs can be changed only when module is disabled
2. SW3 changes between original i9mage and camera input
3. 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**

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

**Filter features**

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

#### **Additional features**:

1. Sharpening filter has a coefficient of the mask that is added to the input image, the number is an integer on 4 bits
2. Dead pixel correction uses a threshold for the permitted maximum error

See *Table 1* and *Table 3* 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 3, Input/output format (for 1 algorithm)

The processed, output image, has the same size as the input but its borders are junk data (Figure 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 1:

|  |  |  |
| --- | --- | --- |
|  | | **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.   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | |  |  |  | |
| **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: |
| **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. |
| **Edge detection** | | Detects edges by using a Laplacian filter, using the kernel below: |

Table 8, Algorithm description



Figure 4, 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 3 for the interfaces:



Figure 5, ir\_filters architecture



Figure 6, 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 7, 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 8, Input/Output format (for 1 algorithm)

The processed, output image, has the same size as the input but its borders are junk data (Figure 2, 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 9 Top module interfaces

**IR\_FILTERS internal design**



Figure 10 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 11 Line buffer interfaces

**Line buffer internal design**



Figure 12, 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 13, line\_buffer timing diagram



Figure 14, Sliding window working principle

The windows move in raster order, as shown in *Figure* 9, Sliding window movement, 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 1, for pixel inputs.



Figure 15, smooth\_filter3x3 internal design

The module convolves 9 input pixels with the mask .

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



Figure 16, 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 17, 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.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |



Figure 18, 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 19, laplace\_filter3x3 internal design

The output is computed using the formula below:

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 20 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 21 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.



Figure 22, 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 23, Sharpening filter timing diagram

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

### Median filter module

**Median filter interfaces**



Figure 24, median\_filter block diagram

**Median filter internal design**



Figure 25 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 26 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 27 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 28 Selector module interfaces

**Selector module internal design**



Figure 29, 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 30, 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 31, 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 32, 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**

1. AXI\_BUS\_SIZE – AXI bus size (only 64)

**AXI IF features**

1. The module uses three different AXI read channels
2. Independent base address configuration for all 3 AXI channels
3. Uses AXI4 reduced interface
4. ARBUSRT is stuck at 2’d1 – only incremental supported
5. ARSIZE must be 3 (64-bit data width)

### 

**Functionality Features**

1. 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.
2. 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**.
3. Input data format for AXI on 64 bits:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Data** | **Bytes** | |  | | --- | | **Bytes** | | **YTE7** | | **BYTE6** | | **BYTE5** | | **BYTE4** | | **BYTE3** | | **BYTE2** | | **BYTE1** | | **BYTE0** | |
| **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 |

1. Output data format:

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Channel 2** | **Channel 1** | **Channel 0** |
| **Data [23:0]** | Map1 | Map1 | Map0 |

**Limitations**

1. No pending request
2. Max image size 2047x2047
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 33, axi2fifo module interface

**Module Parameters**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **AXI\_BUS\_SIZE** | AXI bus size 64 |

**AXI2FIFO module internal design**



Figure 34, 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 35, axi2fifo timing diagram

### FIFO2FRM\_3MAP Module

**FIFO2FRM\_3MAP module interface**



Figure 36,fifo2frm\_3map module interface

**Module parameters**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **FIFO\_DATA\_WIDTH** | Input data width |

**FIFO2FRM\_3MAP module internal design**



Figure 37, 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 38, 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 39, 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 40, fifo2frm\_3map timing diagram

### STS\_INTERRUPTS module

STS\_INTERRUPTS **module interfaces**



Figure 2.4, STS\_INTERRUPTS module interfaces

**Status module internal design**



Figure 41, sts\_interrupts internal design

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



Figure 42, 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 43, 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 44, 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 45, DMA memcpy working principle

The DMA1 from Figure 36, 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 46, 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.

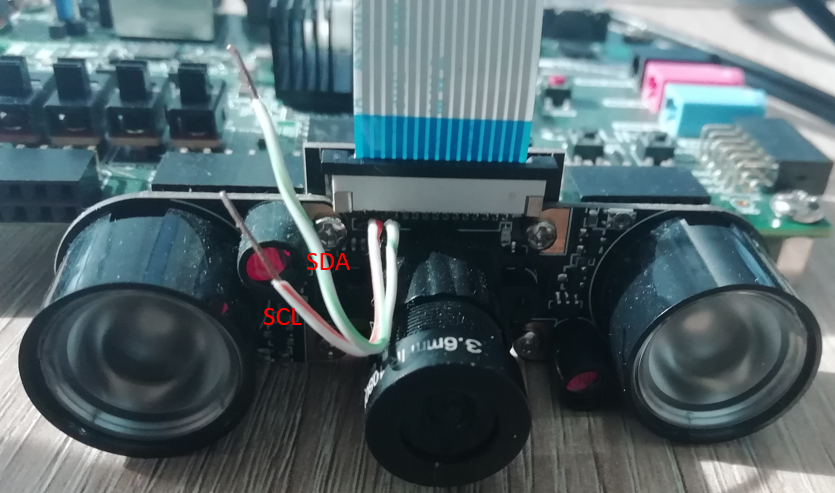


Figure 48, I2C pins on the camera

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.

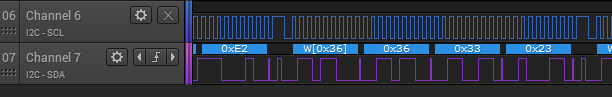


Figure 49, I2C decoded configuration

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.

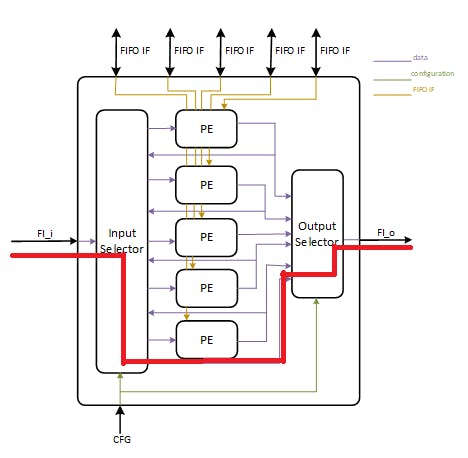


Figure 50, Configuration for transparent

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.

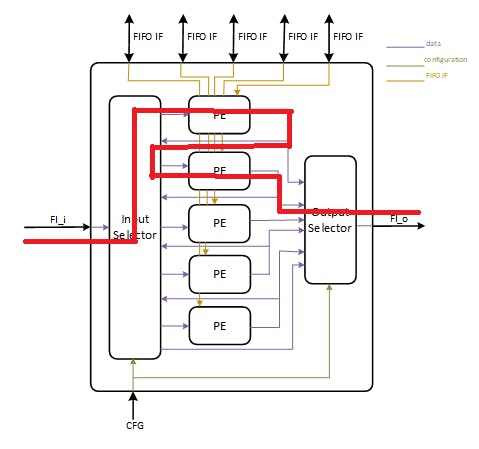


Figure 51, configuration for two filter succession

## 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](http://www.arm.com)

Arm Holdings, AMBA 4 AXI4-Stream Protocol, [www.arm.com](http://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