Prof. Dr.-Ing. Dr. h.c. J. Becker

Digital Hardware Design Laboratory (DHL)

Description for the Design of the Feature Extraction

Institut für Technik der Informationsverarbeitung, Karlsruher Institut für Technologie (KIT)

1 System Structure and Components

In the digital Hardware Design Laboratory (DHL), students are introduced to the techniques of FPGA-based hardware design. Just like laboratory circuit design (Labor Schaltungsdesign) and the Laboratory Software Engineering, DHD is part of the ITIV laboratory project on "autonomously driving TivSeg". Each of the laboratories deals with a different part of the TivSeg system with DHL focusing on the image processing chain.

In the use case scenario of autonomously driving TivSegs, a TivSeg driving in front is marked with a two-colored pattern. An FPGA-based image processing system mounted on a following TivSeg needs to recognize this pattern in input images by extracting similar prominent areas, the so-called *Regions*. The digital representations of these Regions are called *Features* and can be described by their start/end positions in horizontal and vertical direction. In the Software Engineering Laboratory, the extracted Regions/Features are used to detect changes in the relative position of two TivSegs in software. This information is used to generate control signals for the engines, such that the pursuing TivSeg does not lose sight of the pattern and continuously follows the other TivSeg.

In the autonomous TivSeg project, a Microsoft Kinect camera is used for image capturing. The image processing chain extracts two color channels, which roughly correspond to the two colors used for the pattern. Each color channel is converted into a black/white image serving as input for the so-called Region-Growing algorithm, which infers rectangular Regions from the coherent areas in the image.

The overall system of the ITIV laboratory project is shown in Figure 1. The digital Hardware Design (DHD) Laboratory mainly focusses on the system components in the green frame.

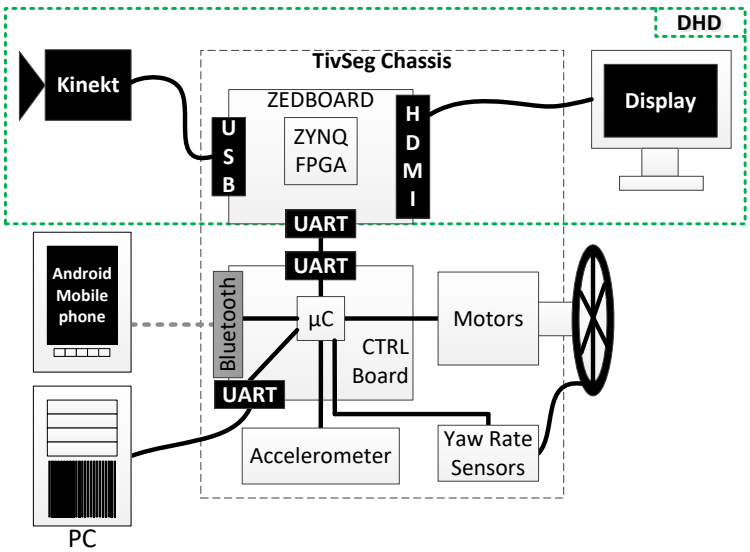

Figure 1: System overview of the ITIV project laboratories

In the DHD laboratory, the ZEDBOARD (http://zedboard.org) is used as target platform. The board is equipped with a System-on-Chip (SoC) from the Xilinx Zynq family, which integrates reconfigurable FPGA logic and two ARM Cortex M9 processor cores on a single chip. The FPGA part of the Zynq is called Processing Logic (PL), whereas the ARM cores form the Processing System (PS). Both subsystems are connected by bus interfaces.

The goal of the laboratory is the realization of the FPGA-logic part of the pattern recognition. For that matter, the ARM processors forward pixel data from the Kinect camera to the *Feature Extraction Core*, which will be implemented in this laboratory. This core extracts the Features from the input image and later returns them back to the processor. Each Feature is represented by a rectangle which bounds an equally colored area in the image.

A structural overview of the Zynq system realizing the image processing chain is shown in Figure 2. The transfer of pixel data from the ARM processors to the Feature Extraction Core (AXI Filter) is implemented in the one case using a register based interface, which can be accessed via the AXI Lite Bus (a simple peripheral AXI bus). As an alternative, pixel transfer can be accelerated using Direct Memory Access (DMA). This is why an AXI Stream interface is part of the project templates as shown in the lower branch of Figure 2Figure 1. The DDR memory, which is used for the DMA, resides outside of the Zynq Chip and can be accessed using the on-chip DDR Memory Controller.

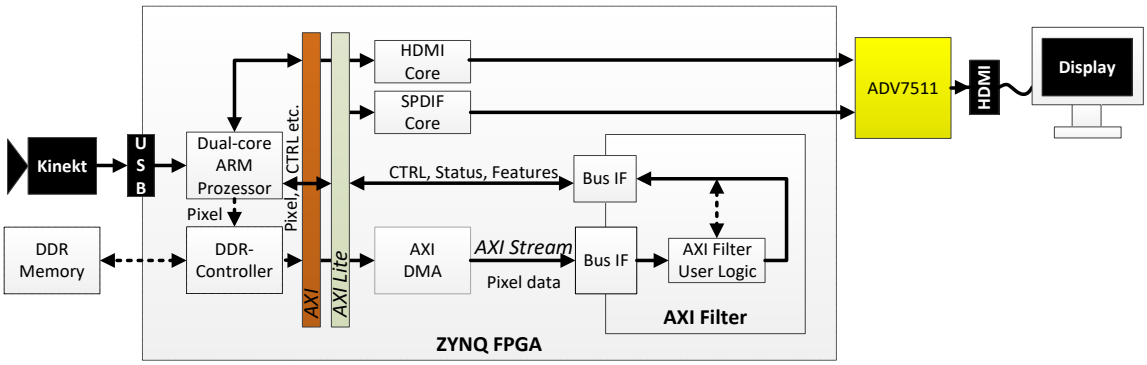

Figure 2: Structure of the FPGA-design realizing the Feature Extraction

The overall virtual data path resulting from the structure in Figure 2 in conjunction with the provided ARM-software is shown in Figure 3.

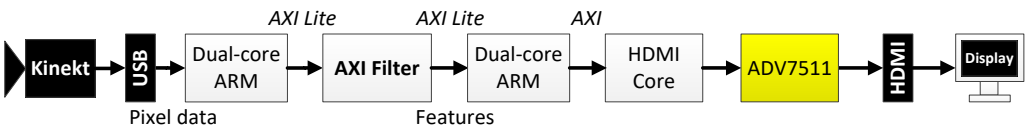

Figure 3: Data flow of the image processing chain

Figure 4 shows the general integration of the Feature Extraction Module into the design and the interconnection to the AXI bus. The actual bus interface is realized using AXI interfacing cores provided by Xilinx (the AXI Lite IPIF and AXI Stream IPIF cores). The user logic module is accessed in AXI Lite templates through a register-based interface. Such a template was extended using an AXI Stream interface to enable efficient transmission of pixel data through the AXI Direct Memory Access (DMA) Core. The top level structure of the actual Feature Extraction Pipeline Hardware is illustrated in Figure 5.

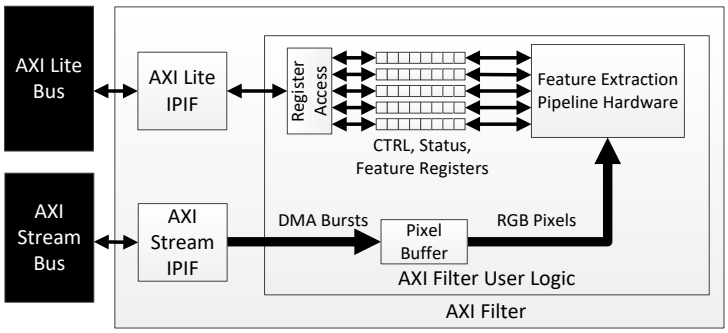

Figure 4: Interconnect between the Feature Extraction Pipeline and the AXI bus

The top structure of the image processing chain (Feature Extraction Pipeline Hardware) is composed of the components illustrated in Figure 5.

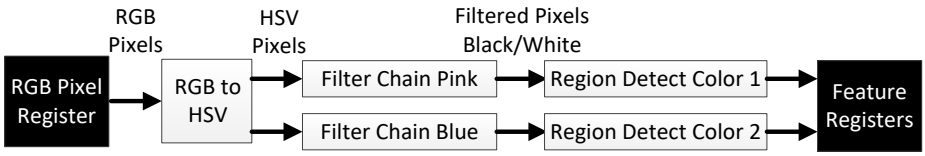

Figure 5: Top level structure of the Feature Extraction

The pattern that is mounted on the TivSeg in the front is shown on the left hand side of Figure 6. The RGB color space is less suitable for detecting the patter, because fluctuations of the ambient light generally affect all coordinates of a color vector in the RGB cube (Figure 6, middle). In contrast, the HSV (Hue Saturation Value) color space contains a separate coordinate for the brightness (Value), such that changes in ambient light only affect this parameter (as long as the ambient light can be assumed to be white). The HSV color space can be visualized as a color cone (Figure 6, right hand side) with the hue corresponding to the angle, the saturation representing the radius and the value matching the height. Since the pattern relies on color differences with large saturation values, the value coordinate can be ignored during pattern recognition. Given that, the influence of ambient light fluctuations is eliminated through the use of HSV coordinates. For that reason, the first step in the image processing chain is the conversion of the input RGB image to HSV color space.

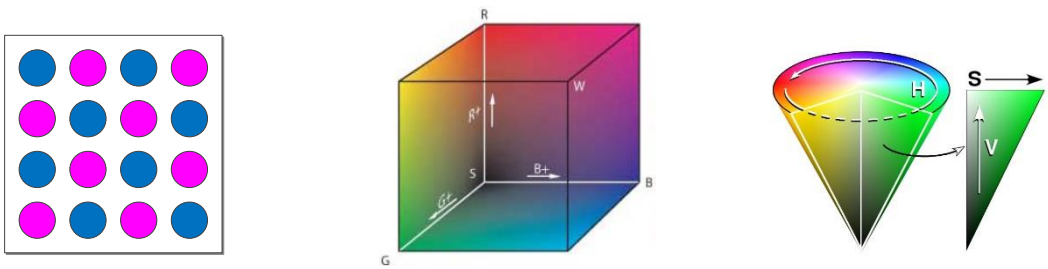

Figure 6: Left: pattern mounted on the TivSegs; Middle: RGB cube[1]; HSV cone[2]

Because the later processing steps rely on black/white images as input, the image data of selected color channels are classified and represented by a single bit per pixel. Since the pattern contains two relevant colors, two black/white images are generated while each image corresponds to one of the two colors. The left side of Figure 7 shows the two black/white images generated from the pattern in Figure 6 when pixels of the corresponding color are considered white.

After that, a simple noise reduction is applied on the black/white images by outputting a '1'-pixel only if there are enough '1'-pixels present in a selected area of the input image (e.g. 3x3

matrix of pixels). This reduces the probability of erroneous pattern detection caused by noise and smaller disturbance.

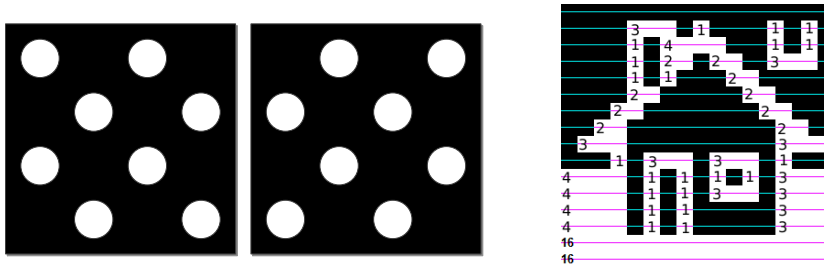

Figure 7: Left, middle: ideal images after the color filtering for blue/pink; Right: run-length encoding[3]

After noise filtering, the pixel data is passed over to the so called *Region Detect Core*. The core is composed of the components shown in Figure 8. The fist module is the *Run-length Encoder* (RLE), which operates similar to the transmission encoding used in old fax machines. The technique uses start and end markers for each individual series of subsequent active pixels ("1", white in Figure 7) within the same line. A set of markers produced by the RLE is referred to as *Run*. The application of run-length encoding allows large image areas with the same color to be stored efficiently and simplifies processing in the subsequent *Label Selection* algorithm.

The purpose of the Label Selection algorithm is to detect connected areas in the image and assign a unique number to each of them. The preceding run-length encoder helps to reduce the number of decisions to be made in the process, because a run is always a sequence of connected pixels, which can be entirely assigned to one region without considering individual pixels. During label selection, each line of the image is compared with the previous line to decide if the current run belongs to an existing region (update), if a new region has to be created (new feature) or if two previously separated regions have to be merged (e.g. in case of the "U" in Figure 7, upper right corner).

The *Feature Calculation* block stores and manages the known regions by creating, changing or merging entries in the component's memory according to the operations (New_Feature, Update, Merge) specified by the Label Selection algorithm. After completion of the Feature Calulation step, the resulting features are accessible to the ARM processor via the register set.

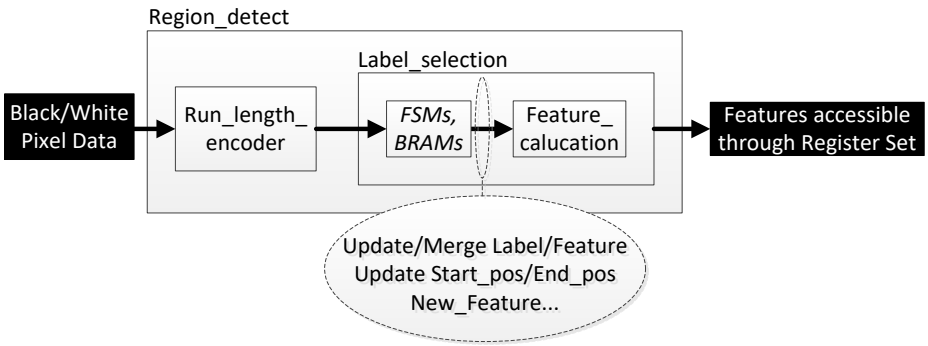

Figure 8: Modules of the Region Detect Core

2 Provided Material

For the laboratory, a **template design** consisting of the Vivado Project including a source file hierarchy and a configuration for the ARM-Processor is provided. The following components are given as they are (compare Figure 9, white boxes): internal structure, AXI interfaces and pixel buffer. In addition, the initial structure of the filter hw module is given.

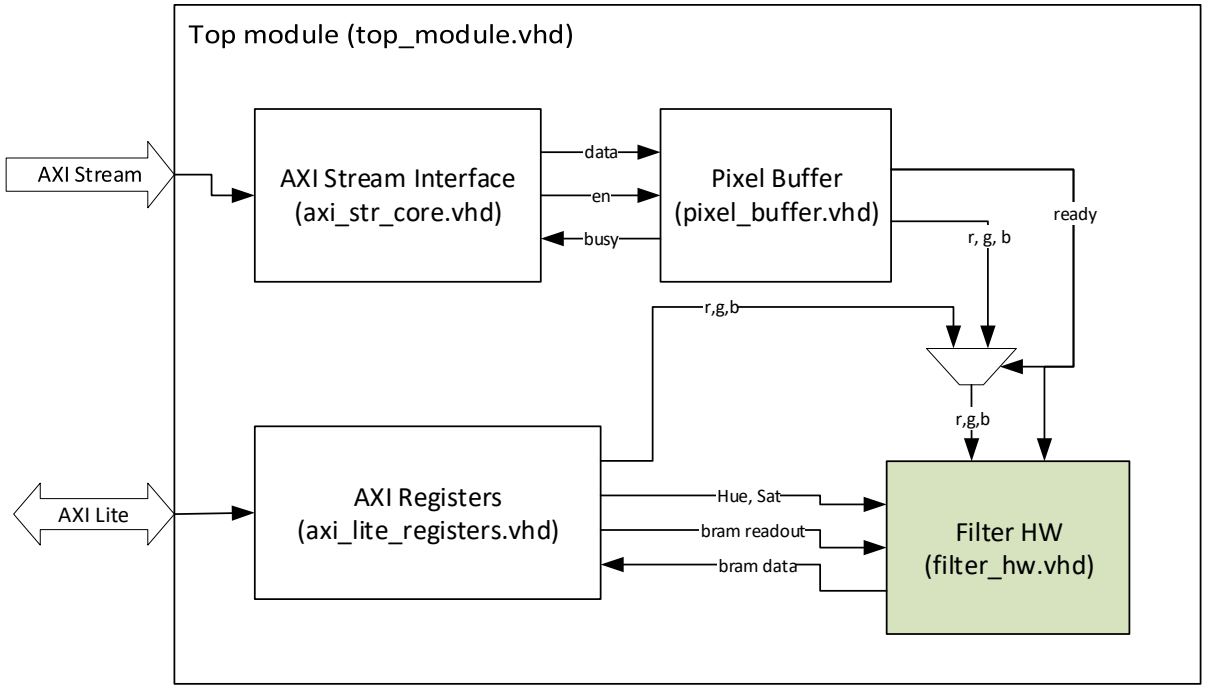

Figure 9 internal stucture of the AXI filter componoent

The top module combines all submodules into a single IP core, which can be instantiated in a Vivado project as shown in session 4. The derived block design can finally be synthesized into the full filter system. To boot the ARM processors of the Zynq, a Linux system including a kernel image is provided on a **SD-Card**. To run the overall system, it is solely necessary to add a hardware binary to the SD Card created from the Vivado design.

To ease testing of modules, the template folder contains several predefined **testbenches** for different components. If necessary, additional testbenches for your components can be written by the laboratory participants.

3 Tasks and Approach

In the laboratory, an image processing pipeline should be developed on top of the provided templates and the SD-Card containing the provided Linux system.

3.1 Guidelines Coding

3.1.1 Template Basics

Most templates for the components of the filter structure already define the entity. Sometimes first bits of code are given as well. Parts, which have to be added by yourself are clearly marked with two comments: `--STUDENT CODE HERE` and `--STUDENT CODE until HERE`. Please add your code between these comments only. This makes it easier to separate your code from the template.

3.1.2 Coding und Bad Smells

In case of disadvantageous logic modelling in VHDL, that the logic description

- Is not synthesizable at all
- Leads to an inefficient realization (long synthesis times, high consumption of resources, bad timing = long asynchronous signal paths between flip-flops)
- Works in simulations but not on the actual hardware (Simulation mismatch)

Such problematic HDL descriptions should be absolutely avoided, for example:

- Not synthesizable
  - Wait until (should be avoided), wait for X ns (not synthesizable)
- Inefficient or not realizable
  - Description of large *Memories (Arrays) without using Block-RAMs* (BRAM) → leads to large Complex Logic-Block (CLB)-Memories („distributed RAM“)
  - *Simultaneous reading/writing* of Arrays (*CLB*-Memories) in different locations **Solution:** better use Dual-Ported Block RAMs or instantiate the same BRAM multiple times if absolutely essential.
  - Writing shift-registers/Arrays on a *variable Location* (Behaviour: „Pointer on shift-register“)
  - Multiplication/Division with two *variable Inputs* without using self-written components or hard-blocks.
    Tip: Reduce calculations to multiplications/divisions by the power of two = shift operation.
  - Extensive asynchronous processes with a large number of dependencies and input variables
- Simulation Mismatches
  - *Extensive use of variables* within processes (a:=b, if X; c :=b if Y …)
  - An *asynchronous process* depends on signals written by other asynchronous processes

    Note: Input values of asynchronous processes should always originate from synchronous processes/signals! Use *pipelining* instead of asynchronous process dependencies

- *Write to BRAMs using asynchronous signals* (Data Input/Write Address Port). This does not work in most of the cases. Asynchronously setting the read-address in contrast mostly works if the address calculation is not too time-consuming. Anyway, it is strongly recommended to drive inputs of Hard-Block IPs like BRAMs by signals written in synchronous processes.

Further important ***coding guidelines can be found in the additional materials*** for the laboratory. Guidelines focusing on Verilog can also be transferred to VHDL in most of the cases. For information on the instantiation of hard-blocks such as BRAMs it is recommended to take a look on the ug901 Vivado Synthesis Guide, especially on the Chapter "***HDL Coding Techniques***".

3.1.3 Conversion/Casting Integer/Std_logic_vector/Unsigned/Signed

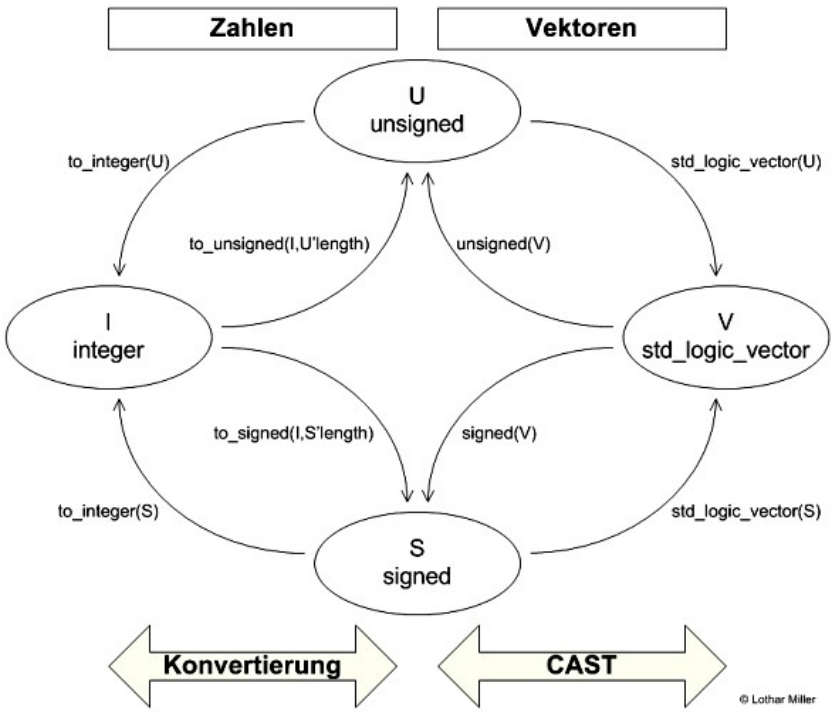

Figure 10: Conversion/Casting between data types in VHDL

3.2 Implementation and Simulation

During the seven afternoons of the laboratory, the processing chain should be developed and simulated component-wise. Participants should use the provided testbenches for the developed components and if necessary verify individual modules with their own tesbenches. All results (realization, expected behavior and test outcome) should be documented in the laboratory reports (see laboratory bulletin for details).

There is a sketch of the structure of the Feature Extraction Template in Figure 11. The sketch shows that for example the Subsystem convert_filter.vhd consist of the component rgb2hsv.vhd and filter_chain.vdh. The RGB2HSV converter in rgb2hsv.vhd reuses the divider div_16_8_8.vhd which has been developed in exercise 3.

You can find all files at:

.\template\hdl

After the individual components where tested, the overall system is simulated incrementally by testing distinct subsystems consisting of multiple components (according to Figure 12). When

testing the subsystem “Converter Filter”, Filter_Chain has to be instantiated twice. When testing Region_detect, it also has to be instantiated twice. The successful simulation of the components and subsystems needs to be demonstrated to the supervisors.

The Testbenches are at:

.\template\testbenches

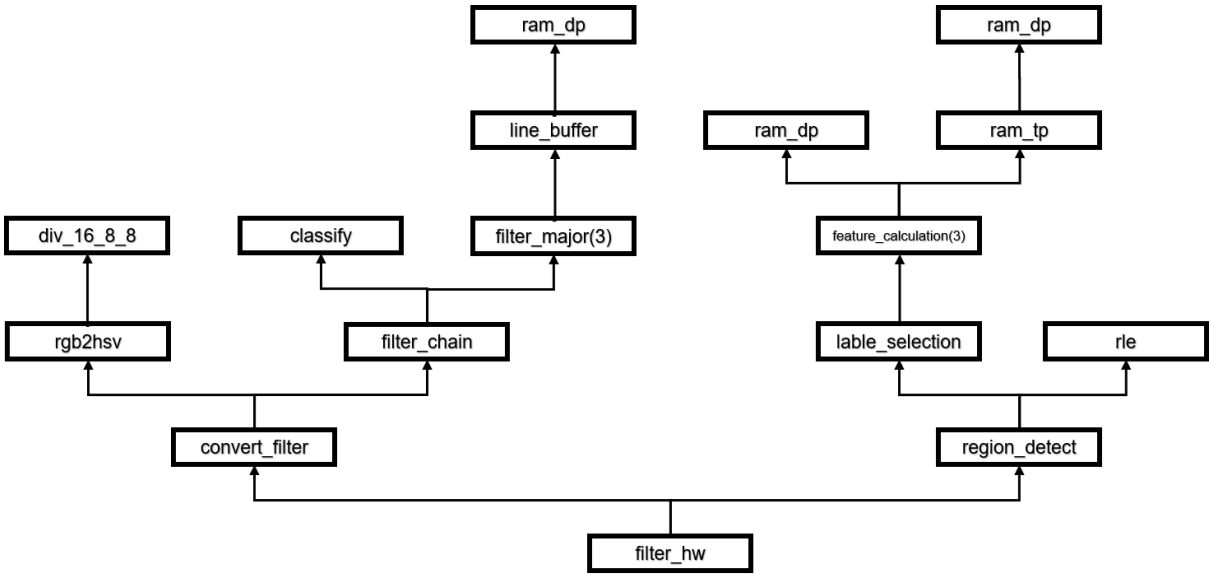

Figure 11: VHDL component structure

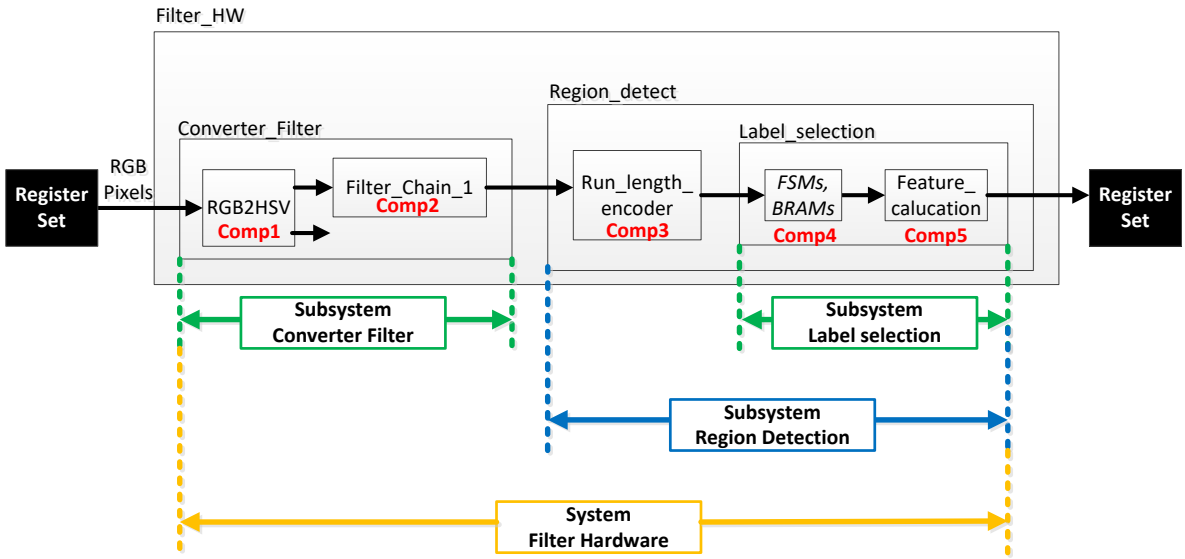

Figure 12: Subdivision of the system in subsystems for structuring and testing purposes

3.3 Individual Components

The Filter Hardware component is controlled via the register set. *Enable* activates the component, the RGB values are valid if data_rdy is on high-level, the minimum/maximum values for hue and saturation are established before starting the actual image processing and stay constant afterwards. Each time an image has been processed, a Resetn (low-active) switches the system back to the initial state. For that reason, it is important to have a well-defined initial state with each signal holding a known and reasonable value.

3.3.1 RGB2HSV

This component is supposed to realize the conversion from RGB to HSV. For that purpose, the algorithm for color space conversion presented in the following should be implemented as VHDL logic. In this context it is important to consider *the differences between sequential programming in C and the* ***simultaneous execution in hardware***. Especially the use of **Pipeline-stages** is beneficial to delay intermediate results and provide them as input for further calculations in later clock cycles.

Since divisions with a divisor, which is *not a power of two*, cannot be synthesized by the VHDL compiler without further ado, the divider developed in exercise 3 should be used here. The divider divides a 17-bit two's complement signed by an 8-bit unsigned integer value and provides a 9-bit two's complement signed as result (this preserves an 8-bit unsigned resolution for further calculations after the division).

To convert pixel data to the HSV color space, the equations listed below should be used. It is important to notice, that these equations assume RGB and SV values to be in the range of *[0, 1]*. In the hardware implementation however, 8-bit integer numbers are to be used for those values. In case of the hue angle, the implementation should use a 9-bit integer representation in degrees (0° - 360°). Furthermore, all the intermediate values such as the results of the divisions should be represented as integer values of a defined bit width. To implement the equations correctly and efficiently you thus have to make appropriate adaptions.

Precondition: $R, G, B \in [0, 1]$

$$MAX := \max(R, G, B),\ MIN := \min(R, G, B)$$

$$H := \begin{cases} 0, & \text{falls } MAX = MIN \Leftrightarrow R = G = B \\ 60° \cdot \left(0 + \frac{G-B}{MAX-MIN}\right), & \text{falls } MAX = R \\ 60° \cdot \left(2 + \frac{B-R}{MAX-MIN}\right), & \text{falls } MAX = G \\ 60° \cdot \left(4 + \frac{R-G}{MAX-MIN}\right), & \text{falls } MAX = B \end{cases}$$

$$\text{falls } H < 0° \text{ dann } H := H + 360°$$

$$S_{\text{HSV}} := \begin{cases} 0, & \text{falls } MAX = 0 \Leftrightarrow R = G = B = 0 \\ \frac{MAX-MIN}{MAX}, & \text{sonst} \end{cases}$$

$$V := MAX$$

Postcondition: $H \in [0°, 360°],\ S, V, L \in [0, 1]$

3.3.1.1 Working with the template

This section explains how to work with the template. You can use this procedure for all other components.

- First open rgb2hsv.vhd (Path see section 3.2)
- The entity is already completely defined and it is not necessary to edit it.

- Use the divider from the third exercise. To do so, use the keyword *component* to include the module and instantiate it twice for Hue and Saturation (this has already been done in the template). The next step is to wire the divider using the keyword *port map*.
- In the next TODO you should describe the behavior of an asynchronous reset.
- The last TODO is located under rising_edge(clk). There you should implement the presented algorithm for convert from the RGB to the HSV color space. It might be necessary to create more signals or functions.

After implementing the RGB2HSV component, it should be tested. It is not necessary to write your own testbench, since there is a predefined one at:

„\Feature_Extraction_Template_V2\Testbenches\rgb2hsv_tb.vhd“

The testbench defines in the beginning two new types “rgb_t” and “hsv_t” and two arrays “rgb_data” and “hsv_data” made of these types (each with the size of 1000). The testing logic is split into the two processes STIMULI_GEN and RESULT_TEST. Before using the testbench, the component under test must first be included. The tested component is also called “unit under test” which is why the instance is called “UUT”.

In the STIMULI_GEN the UUT is fed in each clock cycle with a RGB pixel from rgb_data. The RESULT_TEST process monitors the result_rdy ouput. When the output is a high level (“1”), the HSV values from the UUT are compared with the correct HSV values form hsv_data array.

With the commands *assert*, *report* and *severity* the data will be compared and an appropriate text will be printed with a given category (severity).

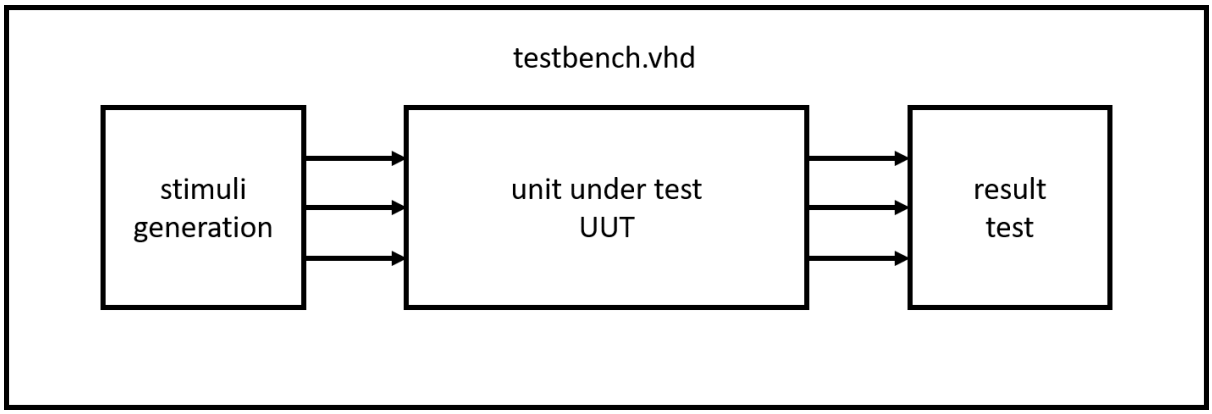

Figure 13: Possible structure of a testbench

More general information about testbenchs are given in „Overview: Testbenches and logic simulation in VHDL“

To test with Modelsim:

- Run Modelsim
- change detection with: cd <new folder> (in the terminal)
- run the Modelsim-script (DO-file) with: do run.do (in the terminal)

3.3.2 Filter Chain

This component consists of two digital filters: classify (black/white filter) and filter major (noise filter).

3.3.2.1 Classify

The module classify verifies every incoming pixel. If the hue and saturation values lie in between hue_min/sat_min and hue_max/sat_max, the output value is set to “1”, if not the filter returns “0”.

3.3.2.2 Filter Major

To generate a low noise image the pixel data is low-pass filtered by the filter major component. On the left side in Figure 14 there is shown an ideal image, on the right side is a noisy image. The noise may result from fine grained image details or image errors in the camera. The basic idea is to suit the real (right) image to the ideal image. This especially reduces the number of runs generated by the run length encoder and the number of regions after the label selection.

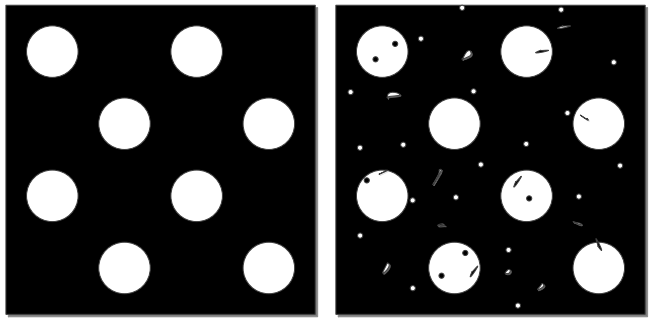

Figure 14: Comparison of an ideal image after color space filtering and a real image with noise incidents, which cause a multiplicity of small regions (white dots and artefacts)

The noise filter decides if an output pixel should be “1” or a “0” based on a quadratic 3x3 Matrix, which is shifted over the whole image (discrete convolution). Thereby, an input parameter determines the minimum number of pixels with value “1” inside of a 3x3 block, which are required to produce an output of “1”.

Before starting to compute output pixels, two lines of the input image must be buffered (two so called *line buffers* with block-RAM are required). The line buffers should realize the first in, first out (FIFO) principle similar to a shift register. In contrast to shift registers, the line buffers should be implemented this behavior using a BRAM. However, data in a BRAM cannot be easily moved from one address to the next, which is why the line buffers should be realized using to the ring buffer principle (more details can be found in the following section).

After initially filling the buffers, the decision between the values “0” and “1” starts at the third black/white pixel in the third line. To simplify implementation, the buffers are not cleared at the end of a line. Instead the filtering process directly continues while the input data is buffered for a delay of two lines before using it for a decision on pixel values.

Using this procedure, each of the components shrinks the output image by two lines while sequentially two pixels are cut from the data stream.

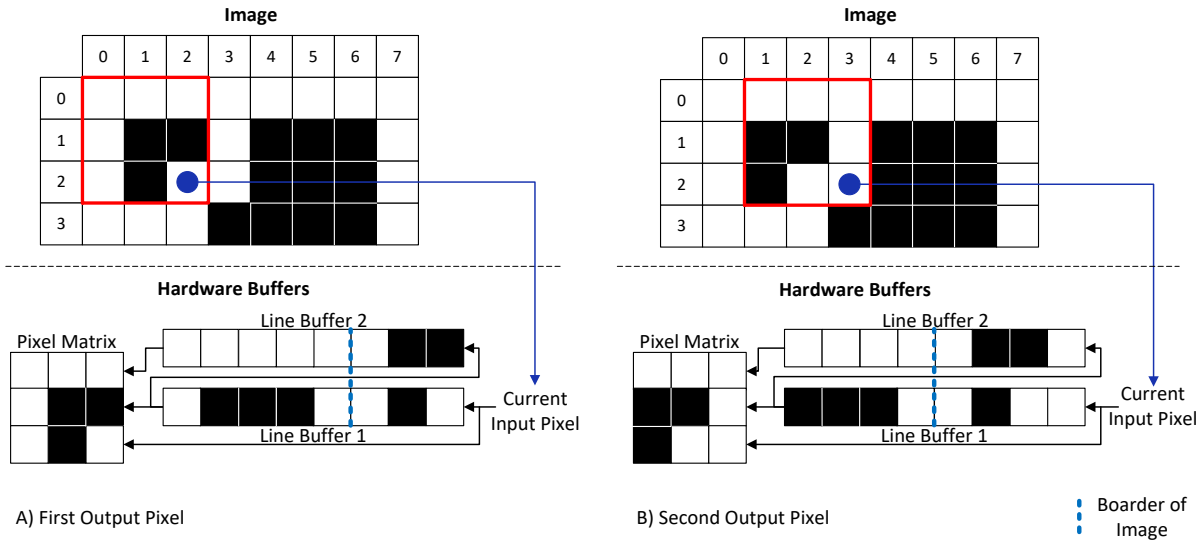

Figure 15: Image filtering using a 3x3 matrix and a threshold of PIXEL_COUNT=4, the Output pixel is the pixel at position (3, 3) of the matrix.A) and B) show consecutive clock cycles. In the upper part of the figure die input picture is shown, below how the entries of the filter matrix (Pixel Matrix) are filled with pixel data. The blue dashed lines show where the left image boarder would occur for the pixel data inside the line buffers

After separate filtering of the hue and saturation images, the output pixels of both filter components are combined using the logical AND conjunction. Subsequently another noise filtering is applied, to further reduce noise artefacts in the combined output image.

Overall the signal flow is shown in Figure 16:

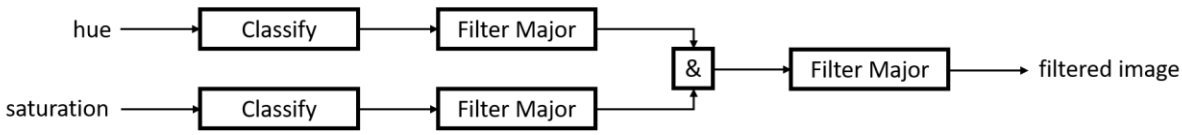

Figure 16: Signal flow diagram of Filter Chain

The size of the final output image of the component is reduced by four lines in the height and four pixels sequentially. This circumstance is compensated by the software and has to be considered in the design of the following components (e.g. pixel or line counters).

3.3.2.3 Working with the template

Figure 11 shows that Filter Chain consists of the components classify and filter major. Again, filter major consists of line buffer, which needs ram_db. Therefore, it is useful to work bottom up.

- First open classify.vhd (Path see section 3.2)
- The entity is already defined completely. It is not necessary to edit it.
- Your task is to write the architecture. To be synchronous to the other components like the RGB2HSV component, it is necessary to choose a suitable sensitivity list. In this process you can insert your logic implementation.
- There is no testbench given since this is a small component. However you can write one on your own one using the RGB2HSV testbench as template.
- In exercise 3 you have already used BRAMs. The component ram_dp is a dual ported Block RAM.
- Find an efficient way to implement a FIFO as ring buffer, when the read- and write pointer of the dual ported BRAM are available (see Figure 17). How many addresses does the line buffer need?

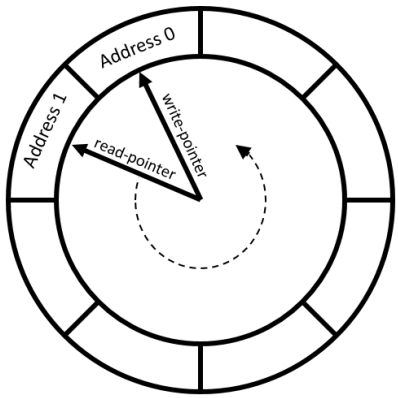

Figure 17: Line buffer implementation using the ring buffer principle

- Test your line buffer using the provided testbench.
- Open filter_major.vhd (Path see section 3.2)
- The component linebuffer is already included, instantiate it twice. Find a way to realize the filter shown in Figure 15. Use one or more clock-synchronous processes.
- Test your filter with the given filter_major testebench. Note that the size of the image will change and some pixel will get the value U (Uninitialized).
- Now, the subsystem filter_chain is complete. It is not necessary to test the whole subsystem because all components in this subsystem have been tested separately. It might be useful to calculate the size of the image after passing the subsystem.

3.3.3 Run-length Encoder

The run-length encoder extracts runs consisting of a start and end position from the input pixel stream. Additionally “end of line“ and “end of file” signals are generated to enable determination of the current line number in later processing steps. Without these signals, it would not be possible to determine the current line number in the run length encoded image if there exists a line without any runs.

The run-length encoder should take the 4 lines (LINES_LOST) that are lost during noise filtering into account when calculating the line numbers. As a convention for the line numbering, it is assumed that two lines are lost in the beginning of the image and two in the end. The same convention holds for the additional 4 pixels that are dropped by the filters: two pixels (PIXEL_OFFSET) are defined to be in the beginning and two in the end of the image. Figure 18 shows an example input image with the resulting runs while taking the lost lines and pixels into account.

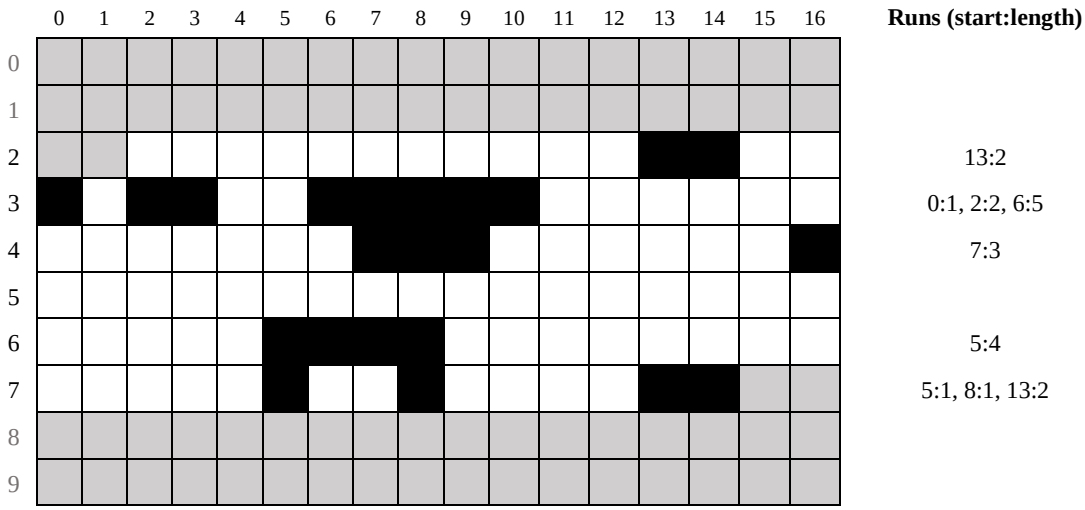

Figure 18: Example image with the resulting runs

3.3.4 Label Selection

This component together with the subsequent one are responsible for the actual feature extraction. The theoretical basics are given in the paper "*SoC Processor for Real-Time Object Labeling in Life Camera Streams with Low Line Level Latency*" which can be downloaded in the additional materials folder. For the implementation, consider the description of the respective component in the paper (Label Selection, Feature Calculation). The "Merger Resolution" component in the paper is not relevant here and does not need to be implemented.

Since the Label Selection component can easily become very extensive, have a look at the commonality of the different cases and plan a suitable implementation with the help of a diagram (state machine) before starting with the actual realization. Keep in mind to select an appropriate set of test signals to allow easy analysis of simulation runs.

During Label Selection, the runs are buffered line-wise and assigned to an appropriate region. Additionally each region gets a unique number (label). By comparing the position of runs in the current and the previous line, it is possible to determine, if a new region has to be created, if the region has to be extended by the current run or if two regions must be merged. In the latter case one region is expanded while the other one is discarded and marked as invalid. In this context it is favorable to keep the region with the lower label and discard the one with the higher label.

When implementing the component you will need an appropriate buffer structure to store the runs in a BRAM. To compare two runs, the BRAM must be read at two different addresses simultaneously. This requires two distinct read ports. At the same time the buffer must be able to store incoming runs from the RLE. Given this requirements, a triple ported BRAM is a reasonable choice to buffer the runs. Figure 19 shows how to implement such a triple ported BRAM using two dual ported BRAMs.

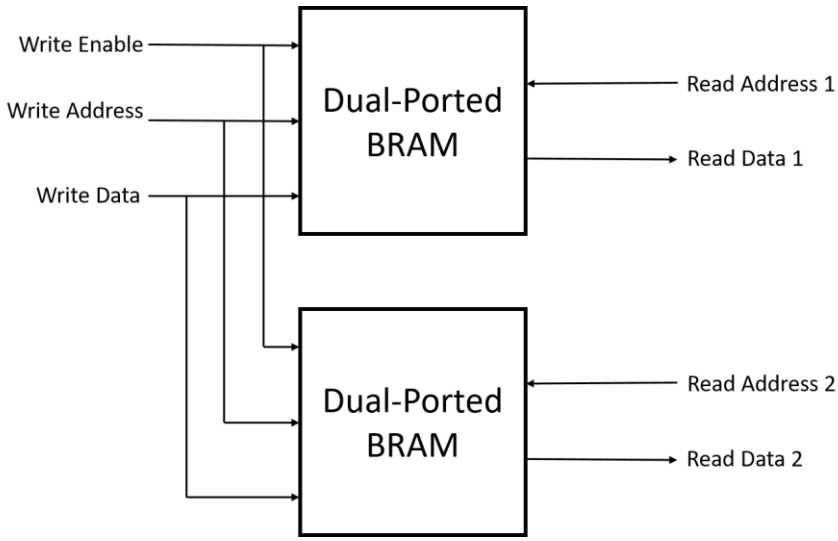

Figure 19: Triple ported BRAM consisting of dual ported BRAMs

3.3.4.1 Working with the template

- There is a template for the triple ported BRAM, open it and realize a triple ported BRAM according to Figure 19.
- The Feature Calculation is given. It is not necessary to change something.
- In Label Selection you must write a finite state machine (FSM). The input of the FSM are the runs from the run length encoder. The FSM decides if it has to create a new feature, update an existing feature or merge two features. The Feature Calculation component then applies the selected operation and saves the features in an internal BRAM. The BRAM is connected to the top level through Label Selection such that it is possible to access to the BRAM from above. This is important because you must set the IDLE signal in Label Selection. When IDLE is high, the Processing Unit will be allowed to read out the feature BRAMs.
- Open label_selection.vhd (Path see section 3.2)
- In the entity, the label selection has the runs as input and the feature BRAM connections as output.
- Write your FSM in the clock-synchronous processes.
- Test the component with the given region_detect testbench.
- Test the whole system with the given filter_hw testbench
- When the simulation was successful, you can start to test on the hardware (see next section)

3.3.5 Feature Calculation

This component stores information about existing regions and implements updating and merging operations for the stored regions. As soon as the features have been extracted completely, they can be read out via the AXI-Bus until the processing chain is loaded with a new image. The component uses two BRAMs with the first one holding the actual feature data and the second one indicating if a valid feature is stored at the corresponding address.

Feature Calculation has the following inputs:

- START_POS
- END_POS
- ROW_NUMBER
- UPDATE

- MERGE
- NEW_FEATURE

The first three represents the runs and the last three the operations.

Definition of the operations:

New Feature: The component will create a new feature with the borders and will write a “1” in the valid-BRAM.

Update: The component writes the Label-Address in the Regions-BRAM then it waits a clock-cycle to read out the borders. After that the component compares the borders and writes them into the BRAM.

Merge: The component writes the two Label-Address to the read pointers. It reads the borders of the two features and then component compares the borders and writes them into the BRAM.

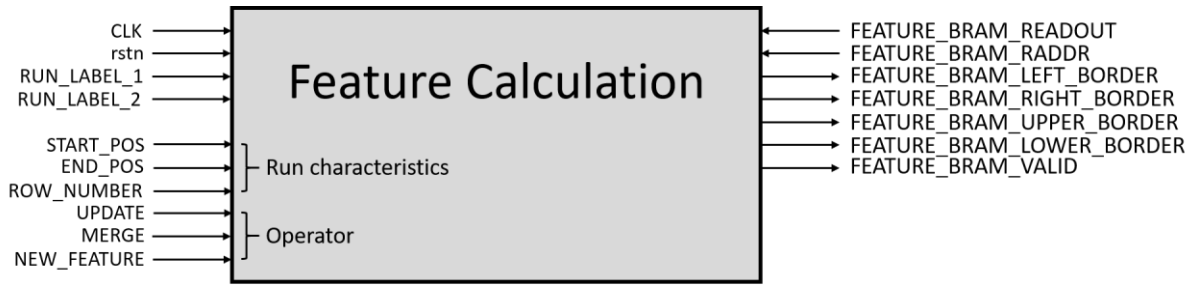

Figure 20: Feature Calculation

3.4 System integration and test

After successfully testing the system (on filter_hw level) using simulation, the system can be implemented and a Boot.bin file is generated. First, the system should be verified with the help of fixed images. If this test is successful, moving pictures from the Kinect camera can be used for further testing.

Therefore, please go ahead and generate a bitstream in your Vivado project. Then speak to one of the supervisors to test your implementation on the full system.

3.5 Useful Hints

3.5.1 AXI-Bus Interface of the Feature Extraction

Name	**Access**	**Address (Base + x)**	**Byte 3** **Bit 31 – 24** **BUS2IP_BE: 3.Bit**	**Byte 2** **Bit 23 – 16** **BUS2IP_BE: 2.Bit**	**Byte 1** **Bit 15 – 8** **BUS2IP_BE: 1.Bit**	**Byte 0** **Bit 7 – 0** **BUS2IP_BE: 0.Bit**
Status Control	RW	0x00	- Bit 31: HIGH	- Bit 16: reset (self clearing)	-	Bit0: idle Bit1: Pipeline enable
Pixel Data	W	0x04	-	B	G	R
H1 Max/Min	RW	0x08	h_max		h_min	
H2 Max/Min	RW	0x0c	h_max		h_min	
S1 Max/Min	RW	0x10	-	s_max	-	s_min
S2 Max/Min	RW	0x14	-	s_max	-	s_min
Readout Status	R	0x18	Bit31: valid_2 Bit26–24: count_2	count_2	Bit15: valid_1 Bit10-8: count_1	count_1
Readout_Region_1_Addr	RW	0x1c	-	Bit16: READOUT	Read_Addr	
Readout_Region_1_Left_Right	R	0x20	Left_Border		Right_Border	
Readout_Region_1_Upper_Lower	R	0x24	Upper_Border		Lower_Border	
Readout_Region_2_Addr	RW	0x28	-	Bit16: READOUT	Read_Addr	
Readout_Region_2_Left_Right	R	0x2c	Left_border		Right_Border	
Readout_Region_2_Upper_Lower	R	0x30	Upper_Border		Lower_Border	

Table 1: Register map of the Feature Extraction module

Register	*Bits/Name*	*Meaning*
Status Control	Status bits of the module	
	Bit 16: reset (self clearing)	Reset module = Soft Reset
	Bit1: Pipeline enable	Enable module
	Bit0: idle	Image processing completed / Module ready
Pixel Data	Possibility to input pixel data using the register interface. RGB color space	
H1/H2 Max/Min **S1/S2 Max/Min**	Max/Min thresholds for color and saturation which are used by the black/white filtering. Otherwise: black pixel	
Readout Status	Indicates if the selected BRAM address contains a valid feature and provides the label of the feature (Readout chain 1 und 2)	
	valid_2/valid_1	Does the address contain a valid feature?
	count_2/count_1	Label of the feature
Readout_Region	Interface to read out features from the Core	
	…__Addr	Read Adress and Read Enable
	Bit16: READOUT	Enable for read access
	Read_Addr	Address to read from the BRAM
	…_Left_Right	Horizontal start and end position of the feature
	…_Upper_Lower	Vertical start and end position of the feature

Table 2: Meaning of the registers in the register map