FPGA Implementation of a Contrast Enhancement Algorithm with Discriminative Filtering

Degree Programme: Grau en Enginyeria de Sistemes Electrònics
Author: Roger Olivé Muñiz
Directors: Chunyan Wang and Jordi Madrenas Boadas
Year: 2013
Index

Collaborations ................................................................................................................................. 7
Appreciation ......................................................................................................................................... 8
Resum del treball ................................................................................................................................ 9
Resumen del proyecto ....................................................................................................................... 10
Abstract ............................................................................................................................................. 11
1. Introduction ..................................................................................................................................... 12
   1.1 Context ......................................................................................................................................... 12
   1.2 Motivation and objectives ........................................................................................................... 13
   1.3 Report structure ........................................................................................................................... 14
2. Background in the contrast enhancement algorithm ...................................................................... 15
   2.1 General overview ......................................................................................................................... 16
   2.2 Block descriptions ......................................................................................................................... 17
       Histogram equalization ................................................................................................................... 17
       Low-pass filtering ........................................................................................................................... 18
       Classification: binarization of images ............................................................................................. 20
       Mask correction ............................................................................................................................. 21
3. Hardware description ....................................................................................................................... 22
   3.1 Requirements and specifications .................................................................................................. 22
   3.2 System structure ........................................................................................................................... 23
   3.3 Detailed block structure ................................................................................................................. 26
       CLAHE block ................................................................................................................................ 26
       Binary masks block ......................................................................................................................... 31
       Filtering block ............................................................................................................................... 32
4. Implementation results ...................................................................................................................... 35
   4.1 Test pictures .................................................................................................................................. 35
       Picture 1 ......................................................................................................................................... 35
       Picture 2 ......................................................................................................................................... 37
       Picture 3 ......................................................................................................................................... 38
# FPGA Implementation of a Contrast Enhancement Algorithm

## 4. Summary

### 4.2 Summary

Conclusions

---

## 5. Conclusions

### 5.1 Project results

Future work

---

## 6. Annexes

### A. Matlab codes

Algorithm Matlab Implementation (Author: Badrun Nahar)

Script to convert bitmaps to .coe format (suitable for recording into ROM with Xilinx Coregen)

Script to read and show image from RAM dump (.mem Modelsim file)

### B. Project VHDL code

Binary_correction_int.vhd

binary_generator_int2.vhd

clahe_complete4.vhd

cche_clipping_int4.vhd

clipping_wrapper_int2.vhd

filter_system_int2.vhd

filter4c.vhd

histogram_int3.vhd

histogram_wrapper_int2.vhd

main3.vhd

median_filter2.vhd

tiling_int3.vhd

transform_interp17.vhd

### C. Modelsim simulations

Global timeline: main entity view

Global timeline: CLAHE block

Global timeline view: binary mask generation

Global time-line: filter block view

## 7. References
Index of figures

Figure 1. Updated Gantt diagram with the time spent on each project step. ....................................................... 14
Figure 2. Block diagram of the algorithm. [3] ........................................................................................................ 16
Figure 3. Bilinear interpolation applied to AHE. Blue zones are bilinearly interpolated, green zones are just linearly interpolated and red zones are left untouched. .................................................................................. 18
Figure 4. Histogram clipping. In this example, the excess is distributed uniformly across the histogram........... 18
Figure 5. BMM filtering operation diagram [3]. .................................................................................................... 19
Figure 6. Kernels used in the first stage of the BMM filter. From left to right: center pixel, \( W_0 \) and \( W_0 \) \[3\] .......... 19
Figure 7. Graphical representation of the thresholding classification process. The dashed lines represent the boundaries of the classification region [3]. ....................................................................................... 20
Figure 8. Group one pattern examples. The rest of the patterns can be obtained shifting or rotating them ........................................................................................................................................... 21
Figure 9. Group two pattern examples. The rest of the patterns can be obtained shifting or rotating them ........................................................................................................................................... 21
Figure 10. Group 3 patterns. ..................................................................................................................................... 21
Figure 11. Block diagram of the system and the execution of its blocks. It shows which steps have been pipelined or set up to run in parallel. ......................................................................................................... 24
Figure 12. Detailed structure of the main entity (the top level entity). ................................................................. 25
Figure 13. Simplified vision of the data flow in the clahe entity. ........................................................................... 27
Figure 14. Simplified diagram of the histogram_wrapper entity structure. ......................................................... 29
Figure 15. Simplified view of the clipper and cdf calculator block. ...................................................................... 29
Figure 16. Representation of the 100 histogram RAMs arranged according to their spatial position in the image. Tiles with the same indexes represent duplicated/quadruplicated tiles (light/dark blue and red respectively). When interpolating, in the places where the tile is duplicated in one direction, there will not be a visible interpolation. ...................................................................................... 31
Figure 17. Diagram of the working principle of the window generator [10]. ......................................................... 32
Figure 18. Group 4 patterns. ..................................................................................................................................... 32
Figure 19. Simplified representation of the data alignment and multiplexing in the filtering stages. All the inputs and outputs are zero padded. Bear in mind that the filter_testbench entity has some logic not represented in this diagram. ........................................................................................................ 33
Figure 20. Diagram of optimal sorting networks for \( n=8 \) and \( n=3 \). Bear in mind that, because 8 is not an odd number, the median in that case is the average of the 2 central values. ................................................................. 34
Figure 21. Original picture 1 and its histogram. .................................................................................................... 35
Figure 22. Picture 1 CLAHE processed. At left using the hardware design; at right using the Matlab script........ 35
Figure 23. Picture 1 final selectively filtered result. At left, using the hardware design; at right using Matlab........................................................................................................................................ 36
Figure 24. At left, histogram of the hardware output for picture 1; at right, histogram of the Matlab script's output obtained with the same image.

Figure 25. Original picture 2 and its histogram.

Figure 26. Picture 2 CLAHE processed. At left using the hardware design; at right using the Matlab script.

Figure 27. Picture 2 final selectively filtered result. At left, using the hardware design; at right using Matlab.

Figure 28. At left, histogram of the hardware output for picture 2; at right, histogram of the Matlab script's output obtained with the same image.

Figure 29. Original picture 3 and its histogram.

Figure 30. Picture 3 CLAHE processed. At left using the hardware design; at right using the Matlab script.

Figure 31. Picture 3 final selectively filtered result. At left, using the hardware design; at right using Matlab.

Figure 32. At left, histogram of the hardware output for picture 3; at right, histogram of the Matlab script's output obtained with the same image.
Collaborations
Appreciation

I want to thank all the people that made possible this work. Chunyan Wang and Jordi Madrenas to give me the possibility to face this project and the challenge it represented. Thanks to Josep Pegueroles and Núria Orduña to give me last impulse to jump in and live this experience. Without them, this project would probably not exist. Also, thanks to Ted Obuchowicz for his valuable help in VHDL and Badrun Nahar to orient me in order to understand her work. Last, but not least, I want to mention my parents Albert and Pilar, my family in general and new and old friends, who helped, suffered me and showed me their support when I really needed it.
Resum del treball

Aquest projecte és la continuació d’una recerca enfocada a millorar els resultats aportats per tècniques populars de millora del contrast d’imatges. Hi ha instantànies que són preses sota condicions molt pobres d’adquisició, com ara escenes amb un rang dinàmic molt gran o imatges mèdiques que requereixen aquestes tècniques per tal de revelar certs detalls que d’altra manera restarien amagats a l’ull humà. El problema és que aquests algorisme (com ara l’equalització adaptativa d’histograma amb contrast limitat o CLAHE per les sigles en anglès) poden revelar no només el detall de la imatge, sinó també el soroll que s’hi oculta, fent difícil distingir el que és rellevant del que és informació inventada pel sensor.

Com a part de l’activitat de recerca en processat d’imatge de Concordia University, un algorisme capaç de millorar aquests resultats sota certes condicions va ser desenvolupat per una estudiant com a tesi final de màster. Aquest algorisme genera màscares binàries que intenten detectar el soroll de la imatge a partir de l’original. Després, una versió de la imatge amb contrast millorat amb CLAHE és filtrada pas-baix només en els pixels detectats com a candidats a tenir soroll. La feina exposada aquí està basada en aquella tesi i estudia el comportament, rendiment i possibilitats de l’algorisme com a implementació en FPGA. El llenguatge de descripció escollit va ser VHDL.

Per tal de fer-ho, es va escollir una metodologia de disseny top-down bottom-up. El primer pas va ser la documentació i procés d’aprenentatge per entendre l’algorisme, la seva implementació inicial en Matlab i els conceptes de processat d’imatge que hi ha al darrere (filtrat pas-baix, equalització d’histograma, classificació, etc.).

Després d’això, i seguint l’aproximació de disseny esmentada, el sistema va ser dividit en parts aïllades que van ser implementades i provades per separat utilitzant Modelsim. El pas següent va ser dissenyar blocs de més alt nivell utilitzant aquells components i finalment l’entitat de més alt nivell va ser construïda per a ajuntar-ho tot. El disseny ha estat destinat a una placa FPGA Xilinx Virtex 6.

Els resultats han donat una imatge amb una millora de contrast molt similar a l’aportada pel codi Matlab original. Malgrat tot, segurament a causa de certs problemes de disseny, l’equalització de la imatge proporciona un resultat una mica més fosc que l’esperat, i sense utilitzar els nivells de gris més pròxims al blanc. El filtrat, d’altra banda, sembla funcionar com s’espera, i el resultat global final és difícil de distingir de l’original sense una comparativa cara a cara. A més, els temps de procés teòrics amb el disseny hardware estan molt per davant del codi original, i pot ser una alternativa viable per processat de video en temps real.
Resumen del proyecto

Este proyecto es la continuación de una investigación enfocada a mejorar los resultados aportados por técnicas populares de mejora del contraste de imágenes. Hay instantáneas que son tomadas bajo condiciones muy pobres de adquisición, como por ejemplo escenas con un rango dinámico muy grande o imágenes médicas que requieren estas técnicas para revelar ciertos detalles que de otra manera restarían escondidos al ojo humano. El problema es que estos algoritmos (como por ejemplo la ecualización adaptativa de histograma con contraste limitado o CLAHE por las siglas en inglés) pueden revelar no sólo el detalle de la imagen, sino también el ruido que se oculta, haciendo difícil distinguir el que es relevante del que es información inventada por el sensor. Como parte de la actividad de investigación en procesado de imagen de Concordia University, un algoritmo capaz de mejorar estos resultados bajo ciertas condiciones fue desarrollado por una estudiante como tesis final de máster. Este algoritmo genera máscaras binarias que intentan detectar el ruido de la imagen a partir del original. Después, una versión de la imagen con contraste mejorado con CLAHE es filtrada paso-bajo sólo en los píxeles detectados como candidatos a tener ruido. El trabajo expuesto aquí está basado en aquella tesis y estudia el comportamiento, rendimiento y posibilidades del algoritmo como implementación en FPGA. El lenguaje de descripción escogido fue VHDL.

Para hacerlo, se escogió una metodología de diseño top-down bottom-up. El primer paso fue la documentación y proceso de aprendizaje para entender el algoritmo, su implementación inicial en Matlab y los conceptos de procesado de imagen que hay detrás (filtrado paso-bajo, ecualización de histograma, clasificación, etc.).

Después de esto, y siguiendo la aproximación de diseño mencionada, el sistema fue dividido en partes aisladas que fueron implementadas y probadas por separado utilizando Modelsim. El paso siguiente fue diseñar bloques de más alto nivel utilizando aquellos componentes y finalmente la entidad de más alto nivel fue construida para juntarlo todo. El diseño ha sido destinado a una placa FPGA Xilinx Virtex 6.

Los resultados han dado una imagen con una mejora de contraste muy similar a la aportada por el código Matlab original. A pesar de todo, seguramente debido a ciertos problemas de diseño, la ecualización de la imagen proporciona un resultado algo más oscuro que el esperado, y sin utilizar los niveles de gris más próximos al blanco. El filtrado, por otro lado, parece funcionar como se espera, y el resultado global final es difícil de distinguir del original sin una comparativa frente a frente. Además, los tiempos de proceso teóricos con el diseño hardware están mucho por delante del código original, y puede ser una alternativa viable para procesado de video en tiempo real.
Abstract

This project is the continuation of a research work focused on improving the current popular techniques for contrast enhancement in images. There are stills produced under very poor acquisition conditions, such as high dynamic range images or medical images that require those techniques in order to reveal details that otherwise remain hidden to the human eye. The problem is, those contrast enhancement algorithms (like Contrast Limited Adaptive Histogram Equalization - CLAHE) can reveal not just the detail of the image but also the noise hidden in it, making it hard to distinguish between the relevant information and the invented detail.

As part of the research activity in image processing of Concordia University, an algorithm able to improve those results under certain conditions was developed by a student as her master thesis. This algorithm generates binary masks that attempt to detect the image noise using the source image. Then, a version of the image with CLAHE applied is low-pass filtered only where the pixels are detected as noise by those masks. The work in this project is based in that thesis, and it studies the behavior, performance and possibilities of the algorithm as a hardware (FPGA) implementation. The chosen description language was VHDL.

To do so, a top-down bottom-up design approach has been employed. The first step was documentation and learning process to understand the algorithm, its initial Matlab implementation and the image processing concepts behind it (low pass filtering, histogram equalization, classification, etc.).

After that, following the design approach, the system was divided in isolated parts that were implemented and tested separately, using Modelsim. Next, higher level blocks were designed using those components and finally the top level entity was built as well. The design was targeted to a Xilinx Virtex 6 FPGA board.

The results gave an image with a visually very similar contrast enhancement to the one provided by the original Matlab code. However, likely due to some design flaw(s), the equalization of the image provides a result a little bit darker than expected, and without using the levels of gray closest to white. The filtering, on the other hand, seems to work just as expected, and the overall result is hard to distinguish from the original without a comparison side to side. Also, the theoretical processing times with the hardware design are far ahead of the original software code, and it can be a viable alternative for real time video processing applications.
1. Introduction

In this chapter the work done in this project and the process through its elaboration are going to be introduced to the reader, as well as the reasons and motivations from which the project was born. Also, the report structure and its content will be briefly detailed.

1.1 Context

In modern society, digital media is a key part of our daily lives. It is present in lots of ways, some of them hard to imagine few years ago, and have improved our lives in equally unexpected ways: entertainment, medicine, security, industry, productivity in general... But in order to create, manage, improve and distribute these multimedia resources, a wide variety of specialised hardware and software components have to interoperate forming a complex chain from the content source to the user's different senses. Image processing is one of the angular stones around which all this technology is built, for the visual part. It has become an essential technology in the present day, and its importance is still growing.

One of the most usual operations in image processing is contrast enhancement. Contrast enhancement algorithms are powerful tool to reveal details on a low-contrast image hidden in a very small range of grey/colour levels. There are various ways to enhance the contrast of an image. One of the most popular algorithms is Histogram Equalization (HE), which has several variants that add some improvements like Adaptive Histogram Equalization (AHE) or Contrast Limited Adaptive Histogram Equalization (CLAHE) [1]. However that procedure, in any of its variations, also reveals noise hidden in the picture, as it can not distinguish between it and the picture detail by itself.

These kinds of algorithms are usually implemented [2] using standard programming languages like C, C++, Java or Matlab to give just some examples, and are executed on top of regular general purpose processors. It is the easier way and enough for certain cases, but this limits in a severe way the achievable performance and efficiency of the design. This is important since certain image processing operations are computationally very intensive. Computers have increased dramatically their power, making them suitable for certain isolated operations in small environments that are not very time-sensitive. A specialized hardware design, however, can optimize much more the performance per watt and get better results with a fraction of the processing power thanks to parallelization, reduction of overhead and pipelining amongst others. However, it comes at a cost. The lack of flexibility of the final design and longer design and manufacturing process (especially if it is an ASIC instead of an FPGA) are drawbacks that prevent a wider use of these solutions. Despite that, they are still a preferable, very interesting choice for power-sensitive applications such as embedded systems, specialised devices or even just as a co-processing module able to assist a general purpose processor in order to increase the overall processing speed, eliminating bottlenecks.
1.2 Motivation and objectives

Given the boost image processing is receiving and how it will likely still be given a very important role in the near future, it is a very active field that is seeing a great number of technical advances that were unimaginable few years ago. To bring this new technology to the masses, new waves of hardware able to keep up with the advancements and fulfill those visions is necessary, and designing that hardware with the tools available nowadays in the market is a very motivating challenge in itself. However, it is not the only motivating factor. Being able to explore new fields that I had barely studied before, basic courses aside, and connect them to my degree’s speciality was a very good opportunity to have a new point of view and learn new things, in this case image processing algorithms.

Last, but not least, the kind of image enhancement studied in the project was an intriguing field as it tries to make an image look better without having any other information available than the image itself. This gave me a question whose answer I was willing to check by myself, which is: how far can one go while trying to make something look better without manipulating or distorting the source material to the point to make the “enhancement” pointless?

The main objective of this project is the implementation in an FPGA of an advanced contrast enhancement algorithm with selective noise filtering, as described in Badrun Nahar’s Master Thesis (Concordia University) [3], using VHDL as the hardware description language. This is done in order to achieve a high performance while efficient execution of that algorithm and evaluate its viability in time-sensitive applications such as real-time video processing, where a hardware implementation can have a clear advantage over software alternatives. The design is targeted to a real board, made entirely with synthesizable code. Also, it is expected to acquire a good knowledge in image processing and implementation of this kind of algorithms in hardware.

In order to accomplish those goals, the plan to face this project consisted of two clearly differentiated long phases. The first one involved a deep study of the algorithm and the image processing basics related to it. The focus was put mainly in histogram equalization, low-pass filtering and classification algorithms. The second phase was the implementation of the proposed advanced contrast enhancement system. It involved looking for ways to implement each component of the design and implementing and testing each individual part and the whole system with Modelsim, using a top-down bottom-up design approach.
1.3 Report structure

The report is structured in 5 different chapters. The first one, the introduction, gives a brief explanation of the contrast enhancement field in particular and image processing in general, as well as other factors that lead to the creation of this thesis. Also, it provides basic information about the objectives and the contents of the rest of the report. The remaining part of the report is structured in a similar way as the evolution of the project itself. Linking to the first phase of algorithm study, the second chapter provides a description of the implemented mathematical algorithm, detailing its separable parts step by step and briefly talking the image processing concepts associated with them. Chapter 3 is associated with the second phase, the VHDL hardware description. With a structure similar to the one in chapter 2, the hardware implementation of each block and separable component mentioned in the previous chapter is discussed in depth. The next step, in chapter 4, consists in debating the results of the design with different tests and images. Finally, to close the report, a fifth chapter with the conclusions makes balance of the work and results and gives some ideas on how it could be improved and/or expanded.

As additional information, 3 annexes with the description code, Matlab scripts and Modelsim simulations are included.
2. Background in the contrast enhancement algorithm

Good contrast is an essential property in most image processing tasks. However, the conditions in which images are taken are not always optimal. Various factors such as the environment, the sensor limitations, lighting or the photographed object itself influence the final result in ways that are not always desirable, leading to a lack of visible detail and limited color range. In these situations, contrast enhancement becomes a good preprocessing tool for a wide range of image processing cases.

There are many different image contrast enhancement techniques but one of the most popular is histogram equalization (HE), and the algorithm implemented in this project is built around it. There are various versions/variations including the basic histogram equalization, but also improved variants like Adaptive Histogram Equalization (AHE) or Clip Limited Adaptive Histogram Equalization (CLAHE) [1].

However, HE and its variants not only increase the contrast of the real detail, but also the imperfections introduced during the acquisition of the picture, as they have no way to differentiate between both cases. This can be troublesome in some contexts such as when the image's contrast is extremely low or when the relevant data can be easily confused with the noise.

For this reason, some designing efforts in that field are now concentrated on reducing the apparition of that undesired data. There are mainly two points where the problem can be faced: right before or right after the equalization. If the noise reduction treatment is done before or during the equalization, relevant information can be lost together with the removed noise, and thus it cannot be detected and enhanced during the equalization. On the other hand, if the noise reduction takes place after the equalization, it is harder to remove because the enhancement makes it more visible and relevant.

In this project, an algorithm [3] based on CLAHE is evaluated and implemented that, following the trend indicated in the previous paragraphs, intends to improve its treatment of the image noise. The reason why this algorithm was chosen to work in its FPGA implementation is that it can be interesting to see how well it can perform in terms of speed and at what cost, as it can be interesting in certain real-time applications. Those can include examples like an image or a video stream of a medical image, like an echography, where it would be valuable not just as an aesthetical improvement, but also as a way to make diagnose easier for a doctor, who could adjust the parameters on the fly, see the improved image in real time, etcetera. CLAHE is already being widely used for this kind of purposes. [1] [4] [5]

In this chapter, the original mathematical algorithm and its strategy to face the noise problem will be described and the main theoretical concepts behind its parts and blocks will be introduced as well.
2.1 General overview

As it has already been said, the enhancement algorithm implemented is based on CLAHE, with some extra processing to improve the end results, focusing specifically on the noise reduction.

In order to partially overcome the noise problems, the route followed by the implemented algorithm has been to selectively filter key areas more likely to have noise in the enhanced picture, which are detected by the algorithm itself according to parameters entered by the user. This is done after the equalization, according to the following scheme:

![Block diagram of the algorithm. [3]](image)

\(I\) is the original source low-contrast input image. The HE-based enhancement block is what contains specifically the CLAHE algorithm, where the contrast enhancement itself takes place. \(I\) is also used to generate some binary masks that will indicate in which areas of the enhanced image the selective filtering must take place and which part must be left untouched.

After the CLAHE step the “pre-filtering” block applies a soft low pass filter to eliminate some high frequency noise in the whole enhanced bitmap. A common Gaussian filter is enough for our needs, and also allows some calibration of the filtering strength thanks to the \(\sigma\) parameter. Because the noise is more visible in homogeneous regions, the idea is to keep filtering low in non homogeneous regions, which will be affected only by this pre-filtering stage. The most obviously homogeneous parts will be the ones affected also by the next steps.

Finally, the different layers of selective filtering (LP\(_n\) blocks) are applied to the image to get the final result. Depending on the binary masks generated with the classification of the pixels on the original image, it is decided whether the output pixel has to be filtered or unfiltered in each stage. The idea of having different stages is to have different levels of filtering strength: the pixels more likely to have noise (homogeneous) will be classified as such in more masks and thus have more filtering applied than those that are more likely to be misclassified but still included in a single filtering step. For these steps, the filter of choice has been a bidirectional multi-stage median filter, thanks to its capabilities when it comes to preserving the edges after the filtering process.
2.2 Block descriptions

Histogram equalization

Histogram equalization is a widely spread technique to enhance contrast. Ideally, it adjusts the gray scale of the image so that the histogram of the original image is mapped onto a uniform histogram using a transformation function. It is used as the base of the contrast enhancement algorithm implemented in this project.

The transformation produces an output value $s$ for each input pixel value. It must be noted, the function must be single valued, monotonically increasing and lying between 0 and 1. For discrete values, if we translate the equations of the continuous domain from probability density functions and integrals to probabilities and summations [6]:

$$p(r_k) = \frac{n_k}{n} \quad (2.1)$$

Where $k$ is a value between 0 and the total image gray levels minus one. $n$ is the total number of pixels in the image and $n_k$ is the number of pixels that have gray level $r_k$. The transformation function is expressed as the following summation:

$$s_k = T(r_k) = \sum_{j=0}^{k} p(r_j) = \sum_{j=0}^{k} \frac{n_j}{n} \quad (2.2)$$

The equalized image, then, can be obtained by mapping each pixel’s level $r_k$ with its corresponding new level $s_k$, which represents a cumulative distribution function (cdf).

Unlike the continuous version, it cannot be demonstrated that it will produce the discrete equivalent of a uniform probability density function. However, it does tend to spread the histogram of the input image in a way it uses a wider range of the gray spectrum.

Adaptive histogram equalization and contrast limited adaptive histogram equalization: improving the original

Adaptive histogram equalization (AHE) and contrast limited histogram equalization (CLAHE) are more complex, improved versions of the standard histogram equalization.

The standard histogram equalization algorithm has the problem that the contrast enhancement is based on the statistics of the entire image. Because of that, some levels will be used to depict parts of the image of low interest.

Adaptive histogram equalization tries to minimize this problem by using a different histogram for each pixel in the image, calculated using a window with the intensity values immediately surrounding that pixel called contextual region. This produces an image in which the objects with different intensity values which lie in
different intensity subranges are simultaneously visible. However, it must be noted that it does not guarantee that in case pixel a value is greater than pixel b value this relationship will be preserved after the equalization.

Moreover, in practical terms, computing a histogram for each pixel is not viable because of its computational cost. As a consequence, in most cases this approach is scrapped and instead [2], the image is divided in a limited number of tiles, and for each of them a histogram is computed. In order to prevent the apparition of the boundaries of the tiles when applying the transformation to the different pixels, bilinear interpolation is used to make the transitions in the final picture smoother.

The other variant mentioned, contrast limited adaptive histogram equalization (CLAHE), adds another layer to the AHE in order to limit the amount of contrast enhancement in areas of the image with low variability. This is done by clipping the highest bins of the histogram and redistributing the clipped excess across the rest of bins.

CLAHE is useful to limit the appearance of certain noise content in zones of low gray level variability by limiting its enhancement. However, the reduced contrast enhancement in certain zones of this alternative could hide the presence of some significant data in the image.

The reason why the variant chosen is CLAHE is its ability to control the degree of enhancement, which can be useful as a tweaking parameter, while maintaining all the improvements present in AHE regarding better contrast enhancement. [3]

**Low-pass filtering**

Low pass filters are useful to eliminate high frequency noise present in an image, as the equalization itself cannot discriminate the noise.

There are various types of low-pass filters, depending on the main purpose of their application. Some are better at removing noise at the expense of blurring the image. Others are better at preserving the edges. Also, the mathematical complexity is another important characteristic to take into account.

In the following lines, the different low-pass filters used in different stages of the noise removal will be described, and their strengths and weaknesses will be discussed as well.
**Gaussian filtering**

Gaussian filters have always been very popular and relevant because of their simplicity: they are easily specified and both the forward and inverse Fourier transforms are real Gaussian functions.

This kind of filter is effective at eliminating the high frequency noises. However, it also blurs the edges and the general detail of the image. The standard deviation of the Gaussian ($\sigma$) will allow a control of the blurriness level applied in the pre-filtering stage:

$$F(x, y) = \frac{1}{2\pi \sigma^2} e^{-\frac{x^2 + y^2}{2\sigma^2}}$$

(2.3)

The higher the sigma is, the stronger the blurring effect will become.

To compute a filtering operation with a Gaussian filter, a good hardware-friendly approach is to use a Gaussian kernel and a discrete convolution operation:

$$y[m, n] = \sum_{m'} \sum_{n'} x[m', n'] h[m - m', n - n']$$

(2.4)

Where $x$ is the input image and $h$ the Gaussian kernel, which for our needs can be just the sampled version of the continuous Gaussian kernel, obtained by sampling $F(x, y)$.

**Median filtering**

Median filters are non-linear filters with good signal variation preservation qualities while smoothing the noise in an image. This is the reason why they were chosen to perform the selective filtering in the homogeneous zones as the important information in the revealed detail is not as likely to be lost compared to alternatives like the Gaussian filter. As the name suggests, the principle behind median filtering is sorting the pixels inside a window and picking the middle value to substitute the central pixel in the window. The fact it uses one existing value in the neighborhood in contrast with average filters is what makes it more suitable to preserve edges. However, it is worse at preserving thin lines and corners. To overcome these problems, various advanced versions of the median filtering have been developed [7].

![Figure 5. BMM filtering operation diagram [3].](image)

![Figure 6. Kernels used in the first stage of the BMM filter. From left to right: center pixel, $W_o$ and $W_0$[3].](image)
In our case, the variant chosen is a kind of multi-stage median filter. Specifically, the bidirectional multi-stage median filter (BMM) [3]. Multi-stage median filters use several stages of median filters instead of a single median for the entire window. BMM filters operate in two steps: first, they find a median of the diagonal pixels and another of the orthogonal pixels, except the central pixel. Then, they take the median of the subset formed by the values calculated in the previous stage and the central pixel.

Put in proper mathematical terms:

\[
y[m, n] = \text{median}[\text{median}[W_D], \text{median}[W_O], x[m, n]]
\]  

(2.5)

**Classification: binarization of images**

Classification allows separation of an image into different regions, based on some measure of similarity. There are various classification methods, but in this contrast enhancement algorithm, specifically, a classification based on clustering is employed: histogram thresholding. Two threshold gray values are selected around a peak in the histogram, which indicates the presence of a large amount of similar pixels. Then, the image is divided in two regions: one containing all the values around the peak delimited by the selected thresholds, which should include a large portion of the image with similar gray values. The other one contains the rest of the image, outside the threshold. With this division, we can generate a binary mask.

![Graphical representation of the thresholding classification process](image)

**Figure 7.** Graphical representation of the thresholding classification process. The dashed lines represent the boundaries of the classification region [3].

This method is useful in low contrast images because they have important peaks in their histograms, as a consequence of this lack of contrast, so big areas of homogeneous pixels can be defined with this classification method.

\[
l'_\text{mask}(i, j) = \begin{cases} 1 & G_1 \leq l(i, j) \leq G_2 \\ 0 & \text{otherwise} \end{cases}
\]  

(2.6)

However, some pixels are likely to be misclassified. Corrective measures can be applied to the masks in order to partially solve that, as we will see in the next sub-chapter.
**Mask correction**

The correction applied to the generated binary masks consists in checking the similarity of the central pixel with its neighbors. If a certain pattern likely to be a misclassification is detected, the pixels in the mask can be corrected (their value can be changed).

Using 3x3 windows, the patterns considered as indicators of non-homogeneous pixels misclassified as homogeneous pixels have been divided into different categories according to their detection method [3]:

Group 1: one pixel wide regions. Very likely to be misclassified pixels. They will have applied a very strong low pass filtering (this means, corrected in all the masks).

![Figure 8](image)

**Figure 8.** Group one pattern examples. The rest of the patterns can be obtained shifting or rotating them.

Group 2: they can also be seen as one-pixel wide, but have more singular forms and more variations than group 1. Compared to group 1, they are not as likely to be misclassified, and probably are located near non-homogeneous regions. As a consequence, they will not be included in the masks oriented to strong filtering.

![Figure 9](image)

**Figure 9.** Group two pattern examples. The rest of the patterns can be obtained shifting or rotating them.

Group 3: one pixel wide patterns drawing a “cross”, as shown in the figure below.

![Figure 10](image)

**Figure 10.** Group 3 patterns.

When any of these patterns or their shifted/rotated variants is detected, the value of the central pixel should be changed. If the pattern belongs to the “very likely misclassified” category, the correction will be applied to all the masks. Otherwise, just the mask for broader filtering will have the change applied.
3. Hardware description

Now that the different parts of the original algorithm have been exposed, it is time to see how it has been ported to a VHDL description.

Programming algorithms in a general purpose or embedded device using standard programming languages has proven to be a good enough solution for quick, flexible and easy deployment. However, there is a serious amount of overhead caused by different reasons. Due to the fact that processors have to be able to do any task, they are not proficient at any of them. Also, because the algorithm is likely implemented using a high-level programming language to speed up development, the translation process to the executable binary will add more overhead as well. Also, the presence of other layers such as an operating system can make things even more redundant.

Dedicated hardware implementations have a much lower degree of flexibility, but require less power to run and can execute the operations much quicker thanks to the specificity of their design. Also, it provides possibilities related to parallel, customized design that are not possible with a traditional programming language executed on top of a processor, which helps increasing the algorithm execution speed even more.

In order to implement the algorithm detailed in chapter 2, a top-down and bottom-up strategy was chosen to organize the development process. The first step consisted in dividing the algorithm in few big separable blocks that can work autonomously. Then, inside each big block, all the smallest separable parts were identified and studied in order to find a good way to translate them to hardware with the available resources, which were studied as well. Each identified part was implemented and tested separately with a test bench and next, the tested parts were used to “assemble” bigger VHDL entities and recreate the big blocks. Each block was tested separately using Modelsim and finally, the top level entity was designed in order to connect the big blocks.

In this chapter it will be described how the whole algorithm has been redesigned in VHDL code, taking advantage of the possibilities it gives to define the level of concurrency: parallel segments, sequential parts, etc. The first step will be defining the requirements and specifications of the implementation, and next proceed with the high level structure of the hardware design and the detailed implementation of each block of the system. In all the cases, the reasons behind each design decision will be addressed as well.

3.1 Requirements and specifications

Before going into depth about how the design has been made, it is important to have an idea of what the different requirements and specifications of the design are, and also how they have been targeted in the design. The main requirements of the design include:

- The ability to process gray scale (8 bit) images with arbitrary precision/resolution. The resolution of the image is specified thanks to different inputs that include width, length and number of pixels.
• Tweaking options or easy modification of the main parameters of the design. At the end, those include:
  o CLAHE clip limit, as a percentage.
  o Definition of 2 gray zones (by entering an upper and lower limit for each one) for the generation of binary masks, as they will specify the thresholds of the classification process.
  o Ease to modify the Gaussian pre-filter VHDL code to change its strength.
• Performance has been a priority over area and power consumption.
• After some tests with the Matlab code, the chosen tile grid for the CLAHE generation is 8x8 tiles. The benefit of making it bigger than that does not seem to be worth the extra resources.

The design has been targeted to a Virtex 6 based board: specifically, the ML605 [8] development board (listed with a $1795 price at time of writing this report). The specifications that are more relevant to this project include 600 MHz maximum clock frequency, 14976 Kbit of block RAM distributed across the board and 768 DSP slices to make mathematical operations. Also, if needed, it has 512 MB of regular DDR3 RAM. The reason to choose ML605 board is that, given the lack of memory or area constraints for this first design, it did provide a very comfortable environment that does not set very strict physical limits in those regards. Moreover, it is easier to focus in just trying to get the maximum performance by trying to parallelize as much as possible, which comes at the expense of more area and memory slices.

Regarding libraries, the IEEE standard library numeric_std will handle all the needed arithmetic variable types (like signed and unsigned) and operations (addition, subtraction, multiplication, division, comparisons). However, some division operations cause problems in the synthesis step using Precision RTL and hence, the design in its current state cannot be synthesized yet and would need some modifications to build properly.

3.2 System structure

In order to design the FPGA implementation, the first step is defining what parts of the algorithm have to be executed sequentially and which ones can be calculated in parallel as they do not depend on each other. Taking a look at the algorithm diagram in chapter 2 [Figure 2] we can clearly divide it in 3 big blocks, represented in Figure 11: CLAHE computation (1), masks generation (2) and noise filtering (3).

The CLAHE and masks blocks are clearly independent as both depend only on the source image to operate. Consequently, both blocks can be run in parallel, concurrently, during what in Figure 11 is identified as sequence 1. However, block 3 needs both the enhanced CLAHE image and the binary masks to select the zones that need to be filtered, so it must not become active until blocks 1 and 2 finish their work, during sequence 2.

Another aspect to be taken into account is the latency added by each block and how this affects the execution of the other steps. The main bottleneck in this regard is the histogram equalization step. This is caused by the need of computing the histograms for all the image pixels before applying the transformation to them, which is the step that can output a pixel per clock. For the transformation itself, as well as the different filter and mask
FPGA Implementation of a Contrast Enhancement Algorithm

generation steps, there is also an introduction of certain latency. However, it is negligible compared to the part caused by the histogram generation.

**Figure 11.** Block diagram of the system and the execution of its blocks. It shows which steps have been pipelined or set up to run in parallel.

Also, the filtering and mask generation steps in general output one pixel per clock, except when jumping to the next image line, because of the implementation of zero padding. The generation and discard process of those extra pixels in the boundaries of the image adds some small incremental delays directly tied to the image’s vertical resolution. However, the zero padding operations are only necessary at the beginning and at the end of the whole filtering/mask generation process; it is not necessary to repeat it before and after each filter (pre-filtering, discriminative 1, discriminative 2). For more details on this process, read the section corresponding to the noise filtering block.

The top level entity is called main, and is stored inside the file main3.vhd. According to all the mentioned super-blocks and algorithm parts, its structure is shown in Figure 12.
As the diagram shows, the inputs are:

1. Image number of pixels (numpixels): necessary to know at what point must the system stop reading the ROM because it has reached the end of the image. It is also necessary to compute certain parameters used internally by certain blocks, such as image tiling, equalization, etc...
2. Image width (x_size) and height (y_size): needed in certain steps where knowing when a line or row ends and the aspect ratio of the image is critical, mainly the tiling and equalization steps.
3. Clip limit (clip_limit): introduced as a percentage. It defines the amount of clipping applied by CLAHE.
4. Top and bottom limits 1 and 2 (limit1_t, limit1_b, limit2_t, limit2_b): used to manually define the homogeneous gray ranges that will have the noise filtering applied. Used in the classification step that generates the binary masks employed to determine what is filtered and what not.
5. Start signal to trigger the process (start_cntr).

Ideally, there should be another input to stream the input image and put it into a RAM instead of the ROM of the diagram. However, due to time constraints and because it is enough for testing purposes, a ROM preloaded with an image will be used instead. All the code corresponding to ROM, RAM and FIFO entities instantiated in the various parts of the design has been generated with the Xilinx Core Generator tool using the faster integrated block ram (BRAM) instead of the higher capacity and slower DDR3 RAM, as there is enough BRAM capacity for the
project needs. As for the outputs, they are reduced to the end flag (end_flag output) that indicates the end of the execution. Also, an output to stream the output image from the filtering block would be available if the design was adapted to connect to another component, but it was not added due to time constraints.

Note that all the blocks are synchronous and controlled with the same clock signal, and share a global reset signal to restart the whole system. There is a start/enable signal in each block to activate it when the previous sequential blocks finish their tasks. It is activated by the “end flag” signals present in the previous blocks.

3.3 Detailed block structure

The big blocks of the enhancement system are divided into various components that perform different sequential tasks, described in their own VHDL files. Each big block has its own “internal top level entity” that wraps all the different subcomponents and the system’s global top level entity (main) connects them and provides access to the external inputs and shared memory resources.

CLAHE block

The CLAHE computation block is the most complex of the 3 main blocks. This is mainly caused by the need to divide the source image into tiles and compute several histograms. This brought several challenges:

a) The generation of the addresses for each individual tile when reading them from the source picture and decision of where the boundaries for each tile do lay.

b) Each tile has its own histogram equalization function. Hence, each tile needs a memory pool to store histogram data.

c) Reuse of the histogram computation components and ROM input.

d) Related to points b) and c), an important amount of multiplexing is needed to manage the access to storage blocks.

The clahe entity is instantiated as clahe_generator on the main entity and described in clahe_complete4.vhd. It is the local top level entity that connects the different internal blocks of the CLAHE computation. It also acts as a memory controller. Its memory controller functions include:

- Switching the image rom I/O between the histogram generation and image transformation blocks when the processing of the tiles finishes.
- Switching histogram rams’ array read interface between clipping and image transform blocks when the processing of all the tiles finishes.
- Switching histogram rams’ array write interface between histogram computing and image transform blocks when the tiles’ histogram computing finish.
- When both processing the different tiles and generating the CLAHE image, it has to manage also choosing between the different tile RAM pools depending on the one that is being computed/accessed.
The instances of other entities found in the clohe entity are:

- **tiler** (entity *image_tiling*): outputs sequentially the tiles in which the image is divided, from left to right and top to bottom of the source picture.
- **histo_wrapper** (entity *histogram_wrapper*): contains the histogram generation block.
- **histo_clipper** (entity *clipping_wrapper*): clips the histogram bins that exceed the specified limit and replaces the histogram ram content with the cumulative distribution function (cdf) needed by the transformation function.
- **equalizer** (entity *histogram_equalizer*): using the source image and all the computed cdf it generates the CLAHE image while simultaneously applying bilinear interpolation.
- **tiles(0-99)** (entity *histo_ram2*): array with all the needed memory pools to store the histograms of the different tiles. Despite the image being divided in 8x8 tiles, there is a total amount of 100 tiles to make the implementation of interpolation easier by duplicating the tiles in the sides and corners. Consequently, we end up with a 10x10=100 array of memories. More details on this can be found in the interpolation section.

When the CLAHE execution is triggered, *tiler* starts accessing the addresses of the top left tile, and outputs them to the histogram generation block, which will deliver the resulting histogram to the corresponding histogram ram block of the memory array. Once it finishes, the ram data input signals are switched to connect them to the clipper block. When *histo_clipper’s* end flag rises, the ram data input signals are switched back to their original position and the tiler block is activated again to begin the next histogram.

![Figure 13. Simplified vision of the data flow in the clohe entity.](image_url)

When all the cdf are ready, the *tiler’s* and clipper’s *end_flag* outputs trigger the image transformation. In that moment, the output pins of the array of memories are switched to connect with the transformation block. The same applies to the output pins of the source image ROM. While calculating the new image, the histogram RAM
pools accessed simultaneously change dynamically depending on what is requested by the transformation component according to the current pixel.

Now, see the inner structure of the different blocks.

**Tiler**

The behavior of the instance tiler is described in the file tiling_int3.vhd under the entity name image_tiling.

Its operation principle is simple from an algorithmic point of view. Making use of the board DSP blocks, it calculates the addresses corresponding to the current tile and outputs the pixel values corresponding to those addresses row per row, from left to right and top to bottom. There are various steps involved:

Computation of the position in the x/y axis referenced to the original image:

\[
x_{\text{coord}} = x_{\text{pos}} + \text{num}_x \cdot x_{\text{tile}}; y_{\text{coord}} = y_{\text{pos}} + \text{num}_y \cdot y_{\text{tile}}
\]  \hspace{1cm} (3.1)

Where \(x_{\text{pos}}\) and \(y_{\text{pos}}\) are the current coordinates referenced to the top left corner of the tile and \(\text{num}_x, \text{num}_y\) the current tile coordinates. \(x_{\text{tile}}\) is the tile width and \(y_{\text{tile}}\) the tile height.

With the previous results, it is easy to compute the memory address corresponding to that pixel:

\[
\text{address} = x_{\text{size}} \cdot y_{\text{coord}} + x_{\text{coord}}
\]  \hspace{1cm} (3.2)

Where \(x_{\text{size}}\) is the width of the source image.

The block also has a couple of small counters that are only reset when the global circuit reset is employed, but not when just the start signal is used. They are used to know the next tile to be generated by storing the next \(x\) and \(y\) position in the grid of tiles (\(\text{num}_x\) and \(\text{num}_y\)).

The size of the tile is computed in real time by another counter every time, too. This is done because the tiles on the right and bottom corners of the image can be smaller than the rest when processing certain image sizes.

**Histowrapper**

This instance, whose entity (histogram_wrapper) is described in the file histogram_wrapper_int2.vhd, computes the histogram of any input image with the aid of its internal histogram component (instance histogram_generator), described in histogram_int3.vhd and which includes part of the computation functionality.

This component is capable of calculating each histogram in the amount of cycles it takes to read a streamed image, just with a few cycles of initial latency at the beginning of the computing process. [8]

Every time a pixel is read, its corresponding bin in the histogram storage memory is read and overwritten with the same value incremented by 1. This way, the bins are gradually incremented according to the inputs until the image reaches its end and the reads/writes stop.
\( \text{histo}_\text{ram}[\text{device}_\text{data}] = \text{histo}_\text{data}_\text{in} + 1 \) \hspace{1cm} (3.3)

Where \( \text{histo}_\text{data}_\text{in} \) is the value of \( \text{histo}_\text{ram}[\text{device}_\text{data}] \) during the previous clock cycle.

**Figure 14.** Simplified diagram of the histogram_wrapper entity structure.

**Histoclipper**

As told before, despite its name, the instance \( \text{histo}_\text{clipper} \) does not only clip the histogram, but also generates the cdf, which is used to apply the transformation later. Its entity \( \text{clipping}_\text{wrapper} \) is stored in \( \text{clipping}_\text{wrapperc}_\text{int2}.vhd \). However, the real functionality is stored in another component inside \( \text{clipping}_\text{wrapper} \): the instance \( \text{histo}_\text{clipper} \) (do not confuse it with the previous one) of the entity \( \text{histogram}_\text{clipper} \), described in \( \text{clhe}_\text{clipping}_\text{int4}.vhd \).

The block operation can be divided in two parts: the two sweeps it performs to clip (1) and next generate the cdf (2). To make it possible, multiplexing to select between the logic of both sweeps must be created. Also, there has to be a counter to generate the access addresses at each sweep.

During the first sweep, the excess detected in the bins that are too high is accumulated in a register that increases each cycle with the amount of detected excess. Meanwhile, if the detected excess is 0, the original amount is written back to the same position. If not, the written value is the clip limit.

\[
\begin{align*}
\text{excess} & = \begin{cases} 
\text{excess} + x[n] - \text{clip limit} & \text{if } x[n] > \text{clip limit} \\
\text{clip limit} & \text{if } x[n] \leq \text{clip limit}
\end{cases} \\
y[n] & = \begin{cases} 
x[n] & \text{if } x[n] \leq \text{clip limit} \\
\text{clip limit} & \text{if } x[n] > \text{clip limit}
\end{cases}
\end{align*}
\]

\hspace{1cm} (3.4)
Where \( x[n] \) is the input histogram bin (number of pixels with that gray value), \( \text{excess} \) the variable that gradually accumulates the total excess of pixels and \( \text{clip}\_\text{limit} \) the maximum tolerated value in the histogram.

Next, the second sweep reads again the contents of the memory sequentially, but this time accumulates the value read plus a fraction of the excess in another register. Then, the read address is overwritten with the accumulator value, thus generating the cdf:

\[
cdf[n] = cdf[n - 1] + y[n] + \frac{\text{excess}}{\text{numpixels}}
\]  

(3.5)

Where \( y[n] \) is the histogram bin value as calculated in the previous step, \( n \) the RAM position (bin, grey level, between 0 and 255), \( \text{excess} \) represents the total clipped excess of pixels and \( \text{numpixels} \) the number of pixels of the histogram’s input image: in this context, the tile size.

**Equalizer (equalization and interpolation)**

In order to compute the equalized and interpolated image, the equalizer instance, from the \textit{histogram\_equalizer} entity included in the file \textit{transform\_interp17.vhd}, needs to have access to all the histogram RAM pools and the source image.

In this case, the block reads the source image RAM sequentially, one pixel per clock. Simultaneously, a part of the block computes the pixel position relative to the \( x/y \) axis and extracts which the 4 neighboring tiles are, the position of their central pixels, the distance between central pixels and between them and the current pixel and also retrieves the corresponding cdf value, minimum cdf value and the tile size. Doing so requires a few clock cycles of latency in order to compute and retrieve all the involved data, but still manages to output a throughput of one pixel per clock. With this data, adequate timing and the concepts presented in chapter 2.2.1 regarding equalization and interpolation, the equalizer block computes the final value of the pixel.

To reach the full dynamic range, the transformed values are normalized between 0 and 255. For each tile:

\[
y_{\text{tile}}[m, n] = \frac{cdf[x[m,n]] - cdf_{\text{min}}}{numpixels - cdf_{\text{min}}} \cdot 255
\]

(3.6)

Then, interpolate the different results:

\[
y[m, n] = (y_{\text{tile\_left}}[m, n](x_2 - m)(y_2 - n) + y_{\text{tile\_right}}[m, n](m - x_1)(y_2 - n) + y_{\text{tile\_down\_left}}[m, n](x_2 - m)(n - y_1) + y_{\text{tile\_down\_right}}[m, n](m - x_1)(n - y_1)) / \{\text{tile\_width} \cdot \text{tile\_height}\}
\]

(3.7)

Where \( x_1, x_2, y_1, y_2 \) are the horizontal and vertical distances between \([m, n]\) and the central pixels of each tile.

Next, it stores the pixel in the corresponding address of a RAM prepared to contain the image, waiting for the start of the filtering process.
Theoretically, the borders of the image should be just linearly interpolated (in a single direction) and the corners should not have interpolation at all. This could involve more complexity in the equalizer block if it had to differentiate between the bilinear, linear and not interpolated cases. To avoid that problem, duplicated border tiles were introduced. This way, when interpolating a pixel in one direction that uses the same tile twice, the result is like if there was no interpolation at all, avoiding the implementation of special cases.

![Figure 16](image.png)  
**Figure 16.** Representation of the 100 histogram RAMs arranged according to their spatial position in the image. Tiles with the same indexes represent duplicated/quadruplicated tiles (light/dark blue and red respectively). When interpolating, in the places where the tile is duplicated in one direction, there will not be a visible interpolation.

**Binary masks block**

The binary masks generation is handled by the component *binarizer*, an instantiation of the entity *mask_generator* contained in the file *binary_generatorMultiplier2.vhd*. This component sweeps the source image ROM to sequentially read the image and classifies the pixels with different comparators that take as a reference the values entered externally in the main top level entity. The output is 1 if the pixel falls inside the limits (homogeneous) or 0 otherwise (equation 2.6).

The next step is applying the correction that will output 2 different masks as a result, stored in 2 different RAM blocks waiting for the beginning of the filtering process. This is done by *corrector_1*, contained in *binary_correctionMultiplier2.vhd* as the *binary_correctionLess* entity. However, what this component does not handle is the addition of zero padding, needed to process the corners and borders of the image. Adding a zero at the end of each row when streaming the image to the correction block input should be enough for a 3x3 window. Then, it just has to be discarded when receiving the output stream before writing to the RAM.

**Corrector_1**

This block needs to detect the patterns described in chapter 2. In order to do that, it is necessary to have a 3x3 window centered in the examined pixel during the same clock cycle. Then, the component can use its algorithms to detect the patterns susceptible of correction. Depending on the result, it will give a changed output for one mask, both or leave the binary value unchanged in both masks.

To get this 3x3 window, the structure employed [9] is the same that will be seen in the filtering section. Basically it consists in the use of FIFO blocks and some registers. The FIFOs act as buffers. The pixel values enter sequentially the first row of registers and, after that, the first FIFO. When the first pixel of the next line enters the first line of registers, the first pixel in the buffer must be pushed out of the FIFO in order to align both rows. This is done by keeping track of the FIFO’s built-in data counter. When the counter reaches the appropriate value, the read enable is turned to 1. The same procedure is repeated for the other rows.
To detect the different patterns introduced in chapter 2, the followed procedure is:

- Group 1: if a zero is detected in the central pixel and the sum of the other zeros in the window is 2 or less, it is certain that it is a group 1 pattern. Therefore, the pixel is corrected in both final masks.
- Group 2: if a zero is detected in the central pixel and the sum of the other zeros in the window is 3, it is likely to be a group 2 pattern, but there are some specific cases known as group 4 that must be discarded. The detection of these group 4 patterns is done by checking that the distribution of the 0 does not match them. Because group 2 patterns are not as likely to be misclassified as group 1, only the mask oriented at broader filtering is corrected. In the other, the zero is left unchanged.

![Figure 18. Group 4 patterns.](image)

- Group 3: if any of the 2 exact patterns in Figure 10 is detected by checking the values in all positions individually, the central pixel is changed in the mask for broader filtering.

**Filtering block**

Finally, the last step is the filtering block. It is managed by the instance named **filters**. It is implemented in the entity **filter_testbench**, which acts as the local top level entity and is described in the file **filter_system_int2.vhd**.

Similarly to the system employed in the binary correction section, all 3 filtering blocks employ an equivalent system to get the filtering windows, but expanded to a 5x5 window. The zero padding, which needs to be 2 pixels wide for a 5x5 window, is added at the beginning of the first filtering stage and removed right before writing the final image to the output RAM after the last filter. It pipelines the 3 filtering steps without any intermediate buffer, which helps minimizing the latency introduced by this block of the system.

The only extra steps involved between filters are the discriminations between filtered and non-filtered pixels. Those take place after their corresponding median filter. The decision takes place there because it is easier and more efficient to just compute the whole filtered image and replace parts of the output with unfiltered pixels than
selectively choosing the pixels that have to be filtered. It would not save time (the current system can already output one pixel per clock after the initial delay) and would increase a lot the complexity of the design.

To do so, the source image stream is not just put inside the first filter, but also copied to a FIFO used as a buffer. The binary mask is also streamed to a FIFO buffer. When the first useful pixel appears at the output of the median filter and its write enable output rises, the read enable for both FIFOs is changed to 1 as well. This way, the filter output stream is aligned with the binary mask and the unfiltered image. Then, the decision can take place: the next step’s input will be the unfiltered one if the aligned mask bit is 0. Otherwise, the multiplexer will choose the filtered bit.

**Figure 19.** Simplified representation of the data alignment and multiplexing in the filtering stages. All the inputs and outputs are zero padded. Bear in mind that the filter_testbench entity has some logic not represented in this diagram.

But how do the filters operate internally?

**Gaussian filter**

As said before, the windowing process for the Gaussian filter is the same used in the binary correction component, but expanded to a 5x5 window. It is described in the filter_system_int2.vhd file, as the entity smooth_filter.

Using the elements of the window and a pre-generated Gaussian kernel, it computes the output for the central pixel using a discrete convolution as presented in chapter 2 (equation 2.4), which employs all the pixels in the 5x5 window. The kernel present in the current version of the description was generated using the fspecial function in Matlab and a standard deviation $\sigma = 0.5$:

\[
\begin{bmatrix}
0 & 0.0028 & 0.0208 & 0.0028 & 0 \\
0.0028 & 1.1332 & 8.3731 & 1.1332 & 0.0028 \\
0.0208 & 8.3731 & 61.8694 & 8.3731 & 0.0208 \\
0.0028 & 1.1332 & 8.3731 & 1.1332 & 0.0028 \\
0 & 0.0028 & 0.0208 & 0.0028 & 0
\end{bmatrix}
\] (3.8)

The kernel can be changed in the code to compile a new filter with a different blurring. The values are multiplied by 100 and rounded to operate with natural numbers. The output is divided by 100 again to get the gray value between 0 and 255.
**Median filter**

The median filter, implemented as the entity `median_filter` in the file `median_filter2.vhd`, is the same for both stages of discriminative filtering. The code structure is almost identical to the one of the Gaussian filter, just changing the computation method of the output pixel and the alignment of the output signals, as the computation has some cycles of latency.

The cause of this latency is the sorting process of the values in the $W_D$ and $W_O$ masks, as well as the final sorting of the BMM filter. The sorting algorithm of the final stage is easy as it has to deal with just three inputs. However, the sorting of $W_D$ and $W_O$ involves eight different integers for each mask, and consequently the algorithm is not trivial.

Number sorting is, in general, a complex problem that has been subject to a lot of study in order improve the efficiency of the existing algorithms. However, there are very few particular cases where a sorting network that is both comparison-optimal and delay-optimal exists. It is the case for number of inputs $n \leq 9$ so fortunately, there is an optimal solution for this implementation. [11] [12]

**Figure 20.** Diagram of optimal sorting networks for $n=8$ and $n=3$. Bear in mind that, because 8 is not an odd number, the median in that case is the average of the 2 central values.

Using the $n=8$ network for $W_D$ and $W_O$ and delaying the same amount of cycles the write enable signal the implementation of the first stage of the median filter is complete (see Figure 5). The second stage is easier, as it involves just 3 numbers. Three cycles, with one comparison per cycle, are enough to sort the values. The implementation of equation 2.5 is finished.
4. Implementation results

This chapter will expose the results of the implementation described in chapter 3. To do so, some sample images will be shown in their original, Matlab processed and hardware processed forms, as well as their resulting histograms. With that data, some key points will be highlighted.

4.1 Test pictures

Various tests have been made with different images. Part of the sample is taken from the original thesis but also new images have been employed to check the behavior in new situations. Each image has a different number of pixels and aspect ratios in order to test the pursued ability of the system to deal with arbitrary sized pictures. Also, different input settings for each case have been used to check that it performs the operations correctly. Bear in mind that the scale of the Matlab-processed images is different because of how it operates with the images, giving a final result consisting in real values between 0 and 1 instead of an integer value in the 0-255 range.

Picture 1

![Figure 21. Original picture 1 and its histogram.](image)

![Figure 22. Picture 1 CLAHE processed. At left using the hardware design; at right using the Matlab script.](image)
Figure 23. Picture 1 final selectively filtered result. At left, using the hardware design; at right using Matlab.

Figure 24. At left, histogram of the hardware output for picture 1; at right, histogram of the Matlab script’s output obtained with the same image.

The first image is one of the test pictures used while developing the original algorithm. It is included in order to test and compare under already known circumstances. For reference, it has a size of 512x439 pixels. In this test, the standard deviation of the Gaussian pre-filter is $\sigma=0.5$, the clip limit is set at 3% and the gray ranges of the pixel classification are $67 \leq x \leq 79$ and $208 \leq x \leq 224$. The differences between the two processed images are barely noticeable but still visible. The hardware simulation produces a slightly darker image, which can be easily associated with the lack pixels in the brighter bins of its histogram, compared to the Matlab results. It is a deviation already visible on the CLAHE images, prior to the noise removal process, so it can be concluded that the design problem is in the CLAHE block. It is likely due to certain operations where truncations instead of rounding are used to make the synthesis less complex and/or other non-identified design bugs. On the other hand, the hardware implementation is as good as the software one at eliminating noise. Running on top of a multicore desktop processor clocked at more than 1 GHz, Matlab needs 11.47 seconds to get the final image. The hardware simulation estimates that 28.5 milliseconds are needed to finish the operations with a 25 MHz clock, which is well below the maximum 600 MHz clock of the board.
Figure 25. Original picture 2 and its histogram.

Figure 26. Picture 2 CLAHE processed. At left using the hardware design; at right using the Matlab script.

Figure 27. Picture 2 final selectively filtered result. At left, using the hardware design; at right using Matlab.
Figure 28. At left, histogram of the hardware output for picture 2; at right, histogram of the Matlab script’s output obtained with the same image.

In the second example, the resolution of the picture changes to 444xx504 pixels. The rest of the settings are equal to those found in picture 1. In this one it is easier to see how the hardware implementation makes overall darker images, by looking at the top half of the picture. Looking at the histograms it is clear: the equalization does not make use of the brighter levels of gray. Still, the output of the design has a good contrast enhancement compared to the original and the differences in filtering are indistinguishable. The processing times are 28.36 milliseconds for the hardware implementation at 25 MHz and 13.27 seconds for the Matlab script. In both cases it is similar to the previous image, which has a similar amount of pixels.

Picture 3

Figure 29. Original picture 3 and its histogram.

Figure 30. Picture 3 CLAHE processed. At left using the hardware design; at right using the Matlab script.
4.2 Summary

Overall, the results are very similar, consistent across all the sample images. It is clear that the CLAHE block has some minor imperfections that prevent it from being on par with the software results in terms of output quality, mainly the lack of use of part of the gray levels. This is probably due to some truncations in certain operations where rounding should be implemented, or other hidden small implementation mistakes in the CLAHE components, most likely the. On the other hand, the selective filtering seems to work very well in all the pictures. In efficiency terms, the hardware implementation clearly stands out with processing times that are various orders of magnitude shorter and an output hard to distinguish without a comparison side to side and with access to the histograms.
5. Conclusions

In this last chapter some conclusions and last thoughts about the work will be exposed and possible future work paths will be suggested as well. Expectations, important events during development, decisions and the final outcome of the design will be talked amongst other experiences and learned lessons.

5.1 Project results

According to the results exposed in chapter 4, it can be seen that the results of the original algorithm have been almost matched, which was the main goal of the project. The contrast enhancement of the image is sharp and in line with the results seen in the original implementation, with some minor imperfections. Also, the selective filtering matches the original implementation pixel per pixel, giving a good smooth effect if calibrated correctly but also the expected weaker results if the adjustments make it show up in non desired places.

It is capable of processing correctly (without glitches) images with arbitrary resolutions up to 512x512 pixels without modifying or recompiling the description and, if necessary, the design can be easily scaled to accommodate bigger image sizes by making buses wider. The only restriction is the amount of resources available in the board. The main tweak parameters can be easily changed with an input signal too.

In terms of efficiency versus speed, the improvements are also remarkable, in line what one would expect with the shift from a high-level preliminary software implementation in Matlab to a specific FPGA implementation. While the original supplied code needed more than 30 seconds to process a single 512x439 image on top of a multicore processor clocked at several GHz, the hardware design can potentially modify the same picture in less than 30 milliseconds with a theoretical 25 MHz clock speed while getting near identical visual results, which is a very welcome improvement. With a faster clock, which is achievable with the target board, the results can be even better without modifying the design itself at all. Also, these numbers definitely situate it as a viable tool for real time video processing, and with more stable processing times than the original Matlab script. Also, the RAM use has been quite low, being necessary only part of the integrated block RAM slices. The slower and much higher capacity DDR3 RAM was left unused. And there is still room for improvement in the implementation, as will be noted later in this chapter.

However, due to time constraints, the design could not be tested in the physical target board as it was initially forecasted. Probably, the initial time plan was too optimistic on my part. Familiarizing with new image processing concepts and revising old ones, fully understanding all the details of an advanced image processing algorithm and recovering the knowledge and agility acquired one year before with VHDL coding were long tasks. While learning those was very valuable and will be very useful in future projects, they also took much more time than expected (almost two of the four available months of work), which left a tight schedule for the coding itself. At the end, due to some implementation difficulties found while working on the CLAHE logic (mainly the tile generation and interpolation steps), this schedule rapidly became too tight to fully realize the initial plan in just four months.
5.2 Future work

Anyway, the final outcome of the project is overall satisfactory. There is a fully working simulation and the code could potentially be synthesized and tried in real hardware spending just few weeks tweaking the code for the problematic divisions (probably with suitable IP cores) and optimizing the delays and latencies. Perhaps this would be the most obvious step to expand the work in the future. Also, to make the implementation suitable for integration in more complex systems, it needs interfaces to acquire the input picture and deliver the equalized output.

Another welcome addition would be a straightforward method to change the Gaussian pre-filter without recompiling the component. It would involve making a component able to generate a kernel according to a certain standard deviation and changing the filtering block to retrieve the kernel generated by the new block.

Additions aside, the architecture of the current parts is still easily improvable. There are some critical points that can be changed to get better performance, decrease the amount of area used or improve the output image. First, as noted in chapter 4, the output images are a bit darker in the hardware implementation than the Matlab code. Analyzing the histograms, it can be easily seen that it is because the histogram equalization does not relocate pixels in the highest gray levels, the ones closest to white, whereas the Matlab implementation does distribute them better. This is probably related to certain truncations during the CLAHE step, mainly in the equalization and interpolation block. Rounding was not initially implemented to simplify both code and logic in those sections of the system. There might be other factors related to potential differences between the mathematical algorithms used by Matlab and the hardware design as well.

Another point in the CLAHE block that can be improved is the storage and access of the block RAMs that store the histogram data of the different tiles. With some work, the duplicated tile RAMs could be scrapped and implement a way to skip their access and use the data provided by the original tile RAM instead. This way, the total RAM usage of the system can be lowered.

Also, in order to improve the latency and reduce even more the amount of used RAM, the memory that stores the CLAHE processed image could be replaced by a small FIFO that could act as a buffer and begin the filtering process when the first CLAHE processed pixels appear (pipelining). The same is possible with the binary masks, but using leghtier buffers as the mask generation process is much shorter. This way, not just the RAM usage would be lower but also the latency would be considerably reduced and the area corresponding to certain address counters would be cut too, as the FIFO blocks would no longer need them. It was not implemented directly this way because the address counters and the RAMs were inherited from the initial test benches for the isolated components of the implementation.

All the mentioned improvements are oriented to reducing both the area and the processing time. However, depending on the final application, it might be desirable to modify other parts of the algorithm in order to prioritize either area or performance in places where they can conflict.
6. Annexes

A. Matlab codes

Algorithm Matlab Implementation (Author: Badrun Nahar)

tic;
input_image= imread('C:\Users\Roger\Documents\UPC\TFG\imatges\microscopic_merge2.jpg');
%input_image= imread('/home/roger/Documents/UPC/TFG/imatges/444x504.jpg');
A1=rgb2gray(input_image);

cliplimit_cla=0.08;
grid_size=[8 8];
figure, imshow(A1),title('input image');
figure, imhist(A1),title('input image histogram');

% CLAHE Enhanced Image
enhanced_A=adapthisteq(A1,'ClipLimit',cliplimit_cla,'NumTiles',grid_size);
figure,imshow(enhanced_A),title('After CLAHE');

enhanced_A1=double(enhanced_A)/255;
filt_gaussian=fspecial('gaussian', [5 5], 0.5);
enhanced_A12=imfilter(enhanced_A1,filt_gaussian,'conv','replicate');

figure, imshow(enhanced_A12),title('pre-filtered output');

% size adjustment with zero padding
[m n]=size(A1);
x1=zeros(m+4,n+4);
x1(3:m+2,3:n+2)=enhanced_A12(1:n);
x1(2,3:n+2)=enhanced_A12(1,1:n);
x1(m+3,3:n+2)=enhanced_A12(m,1:n);
x1(m+4,3:n+2)=enhanced_A12(m,1:n);
x1(3:m+2,1)=enhanced_A12(1:m,1);
x1(3:m+2,2)=enhanced_A12(1:m,1);
x1(3:m+2,n+3)=enhanced_A12(1:m,n);
x1(3:m+2,n+4)=enhanced_A12(1:m,n);
x1(1,1)=enhanced_A12(1,1);
x1(1,2)=enhanced_A12(1,1);
x1(2,1)=enhanced_A12(1,1);
x1(2,2)=enhanced_A12(1,1);
x1(1,1)=enhanced_A12(1,1);
x1(2,1)=enhanced_A12(1,1);
x1(2,2)=enhanced_A12(1,1);
x1(1,n+3)=enhanced_A12(1,n);
x1(1,n+4)=enhanced_A12(1,n);
x1(2,n+3)=enhanced_A12(1,n);
x1(2,n+4)=enhanced_A12(1,n);
x1(m+3,1)=enhanced_A12(m,1);
x1(m+3,2)=enhanced_A12(m,1);
x1(m+4,1)=enhanced_A12(m,1);
x1(m+4,2)=enhanced_A12(m,1);
x1(m+3,1)=enhanced_A12(m,1);
x1(m+3,2)=enhanced_A12(m,1);
x1(m+4,1)=enhanced_A12(m,1);
x1(m+4,2)=enhanced_A12(m,1);

% Clustering original image
d=zeros(size(A1));
for i=1:m*n
    if (A1(i)>=0 && A1(i)<=20) || (A1(i)>=210 && A1(i)<=250)
        d(i)=1;
    else
        d(i)=0;
    end
end

figure, imshow(d), title('gray level thresholding');

% Region Correction for low-pass2
count1=8*ones(m+4,n+4);
count2=8*ones(m+4,n+4);
d1=zeros(m+4,n+4);
d2= zeros(m+4,n+4);
d2(3:m+2,3:n+2)=d(:,:);
d3=d1;
for i=3:m+2
    for j=3:n+2
        if d1(i,j)==0
            count1(i,j)=count1(i,j)-(d1(i-1,j-1)+d1(i-1,j)+d1(i-1,j+1)+d1(i,j-1)+d1(i,j+1)+d1(i+1,j-1)+d1(i+1,j)+d1(i+1,j+1));
            if count1(i,j)<=1
                d2(i,j)=1;
            elseif (d1(i-1,j-1)==d1(i-1,j)==d1(i+1,j)==d1(i-1,j+1)==d1(i,j+1)) && (d1(i-1,j)==d1(i+1,j)==d1(i,j-1))
                % if count2(i,j)<=3
                if (d1(i-1,j)==d1(i,j+1)==d1(i,j-1)==0) || (d1(i,j+1)==d1(i,j-1)==d1(i-1,j)==0) || (d1(i,j-1)==d1(i,j)==d1(i+1,j))
                    % if count2(i,j)<2
                    if (d1(i-1,j-1)==d1(i-1,j+1)==d1(i+1,j)==d1(i+1,j-1)) && (d1(i-1,j)==d1(i,j+1)==d1(i+1,j-1))
                        % if count2(i,j)<2
                        if (d1(i-1,j-1)==d1(i+1,j-1)==d1(i+1,j)==d1(i,j))
                            % if count2(i,j)<2
                            if (d1(i-1,j)==d1(i+1,j))
                                % if count2(i,j)<2
                                if (d1(i-1,j)==d1(i,j))
                                    d3(i,j)=1;
                                else
                                    count2(i,j)=1;
                                end
                            end
                        end
                    end
                end
                % if count2(i,j)<2
            else
                d3(i,j)=1;
            end
        end
    end
end

figure, imshow(d2(3:m+2,3:n+2)), title('Group-1 & Group-3 corrected for low-pass2');

%%% Region correction for low-pass1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Group-1, group-3 & Group-2 with some preservation correction
for i=3:m+2
    for j=3:n+2
        if d1(i,j)==0
            count2(i,j)=count2(i,j)-(d1(i-1,j-1)+d1(i-1,j)+d1(i-1,j+1)+d1(i,j-1)+d1(i,j+1)+d1(i+1,j-1)+d1(i+1,j)+d1(i+1,j+1));
            if count2(i,j)<3
                if (d1(i-1,j)==d1(i+1,j+1)==d1(i,j+1)==0) || (d1(i,j+1)==d1(i+1,j)==d1(i-1,j+1))
                    % if count2(i,j)<2
                    if (d1(i-1,j-1)==d1(i-1,j+1)==d1(i+1,j)==d1(i+1,j-1)) && (d1(i-1,j)==d1(i,j+1)==d1(i+1,j-1))
                        % if count2(i,j)<2
                        if (d1(i-1,j-1)==d1(i+1,j-1)==d1(i+1,j)==d1(i,j))
                            % if count2(i,j)<2
                            if (d1(i-1,j)==d1(i,j))
                                % if count2(i,j)<2
                                if (d1(i-1,j)==d1(i,j))
                                    d3(i,j)=1;
                                else
                                    count2(i,j)=1;
                                end
                            end
                        end
                    end
                end
            else
                d3(i,j)=1;
            end
        end
    end
end

figure, imshow(d3(3:m+2,3:n+2)), title('Group-1, Group-3 & Group-2 with some preservation corrected: R. C. for low-pass1');

% Discriminative filtering
filtered_A123=zeros(m+4,n+4);
%low-pass 1, binary mask \(d_3(3:m+2,3:n+2)\)
for \(i=3:m+2\)
for \(j=3:n+2\)
if \(d_3(i,j)==1\)
    hor_ver_data = \([x_1(i,j-2) \ x_1(i,j-1) \ x_1(i,j+1) \ x_1(i,j+2) \ x_1(i-2,j) \ x_1(i-1,j) \ x_1(i+1,j) \ x_1(i+2,j) \ x_1(i,j)]\); Mr=median(hor_ver_data);
    diag_data = \([x_1(i-2, j-2) \ x_1(i-1, j-1) \ x_1(i+1,j+1) \ x_1(i+2,j+2) \ x_1(i+1,j-1) \ x_1(i+2,j-2) \ x_1(i-1,j+1) \ x_1(i-2,j+2) \ x_1(i,j)]\); Md=median(diag_data);
    vect1=[Mr Md x_1(i,j)];
    filtered_{A123}(i,j)= median(vect1);
else filtered_{A123}(i,j)=x_1(i,j);
end
end
end

filtered_{A1234}=zeros(m+4,n+4);
%low-pass 2, binary mask \(d_2(3:m+2,3:n+2)\)
for \(i=3:m+2\)
for \(j=3:n+2\)
if \(d_2(i,j)==1\)
    hor_ver_data1 = \([filtered_{A123}(i,j-2) \ filtered_{A123}(i,j-1) \ filtered_{A123}(i,j) \ filtered_{A123}(i,j+1) \ filtered_{A123}(i,j+2) \ filtered_{A123}(i-2,j) \ filtered_{A123}(i-1,j) \ filtered_{A123}(i+1,j) \ filtered_{A123}(i+2,j) \ filtered_{A123}(i,j)]\); Mr1=median(hor_ver_data1);
    diag_data1 = \([filtered_{A123}(i-2, j-2) \ filtered_{A123}(i-1, j-1) \ filtered_{A123}(i+1,j+1) \ filtered_{A123}(i+2,j+2) \ filtered_{A123}(i+1,j-1) \ filtered_{A123}(i+2,j-2) \ filtered_{A123}(i-1,j+1) \ filtered_{A123}(i-2,j+2) \ filtered_{A123}(i,j)]\); Md1=median(diag_data1);
    vect2=[Mr1 Md1 filtered_{A123}(i,j)];
    filtered_{A1234}(i,j)= median(vect2);
else filtered_{A1234}(i,j)= filtered_{A123}(i,j);
end
end
end

filtered_{A12345}=zeros(m+4,n+4);
%low-pass 3, binary mask \(d_2(3:m+2,3:n+2)\)
for \(i=3:m+2\)
for \(j=3:n+2\)
if \(d_2(i,j)==1\)
    hor_ver_data2 = \([filtered_{A1234}(i,j-2) \ filtered_{A1234}(i,j-1) \ filtered_{A1234}(i,j) \ filtered_{A1234}(i,j+1) \ filtered_{A1234}(i,j+2) \ filtered_{A1234}(i-2,j) \ filtered_{A1234}(i-1,j) \ filtered_{A1234}(i+1,j) \ filtered_{A1234}(i+2,j) \ filtered_{A1234}(i,j)]\); Mr2=median(hor_ver_data2);
    diag_data2 = \([filtered_{A1234}(i-2, j-2) \ filtered_{A1234}(i-1, j-1) \ filtered_{A1234}(i+1,j+1) \ filtered_{A1234}(i+2,j+2) \ filtered_{A1234}(i+1,j-1) \ filtered_{A1234}(i+2,j-2) \ filtered_{A1234}(i-1,j+1) \ filtered_{A1234}(i-2,j+2) \ filtered_{A1234}(i,j)]\); Md2=median(diag_data2);
    vect3=[Mr2 Md2 filtered_{A1234}(i,j)];
    filtered_{A12345}(i,j)= median(vect3);
else filtered_{A12345}(i,j)= filtered_{A1234}(i,j);
end
end
end

figure,imshow(d3(3:m+2,3:n+2)),title('region corrected mask for low-pass1');
figure,imshow(d2(3:m+2,3:n+2)),title('region corrected mask for low-pass2');
figure,imshow(filtered_{A123}(3:m+2,3:n+2)),title('output of 1st stage of discriminative filtering by 5x5 BMM');
figure,imshow(filtered_{A1234}(3:m+2,3:n+2)),title('output of 2nd stage of discriminative filtering by 5X5 mult med');
toc;
figure,imshow(filtered_{A12345}(3:m+2,3:n+2)),title('output of 3rd stage by 5X5 mult med');
toc;
Script to convert bitmaps to .coe format (suitable for recording into ROM with Xilinx Coregen)

Adapted from [14].

```matlab
%Al=rgb2gray(input_image);

%read bmp data in and display it to the screen
%image_name='C:\Users\roger\Documents\UPC\imatges\im_grisa.bmp';
image_name='/home/roger/Documents/UPC/imatges/444x504.jpg';
input_image = imread(image_name);
imdata = rgb2gray(input_image);
%imdata=input_image;
image(imdata);
numpixels=numel(imdata);

%create .COE file
COE_file=image_name;
COE_file(end-2:end)='coe';
fid=fopen(COE_file,'w');

%write header information
fprintf(fid,';******************************************************************
');
fprintf(fid,';****                 BMP file in .COE Format                 *****
');
fprintf(fid,';******************************************************************
');
fprintf(fid,'; This .COE file specifies initialization values for a
');
fprintf(fid,'; block memory of depth= %d, and width=8. In this case,\n',numpixels);
fprintf(fid,'; values are specified in hexadecimal format.\n');

%start writing data to the file
fprintf(fid,'memory_initialization_radix=16;
');
fprintf(fid,'memory_initialization_vector=
');
%convert image data to row major
newimdata=transpose(double(imdata));
%write image data to file
for j=1:(numpixels-1)
    fprintf(fid,'%s,\n',dec2hex(newimdata{j},2));
end
%last data value supposed to have a semicolon instead of a comma
fprintf(fid,'%s;\n',dec2hex(newimdata{numpixels}));
%clean shutdown
fclose(fid)
```
Script to read and show image from RAM dump (.mem Modelsim file)

clear
clc
A=fopen('def415.mem');
B =fgetl(A);
B =fgetl(A);
B =fgetl(A);

width=415;
height=265;

%width=512;
%height=512;

empty=512*512-width*height;

rubbish = 0;
for i=1:(empty)
    rubbish = rubbish + fscanf(A, '%u\n', 1);
end

for i=1:height
    i2=height+1-i;
    for j=1:width
        j2=width+1-j;
        C(i2,j2)=uint8(fscanf(A, '%u\n', 1));
    end
end

figure;
imshow(C);

fclose(A)
B. Project VHDL code

Binary_correction_int.vhd

------------------------------------------------------------------------
-- Original smooth_filter: Núria Orduña
-- Modified by: Roger Olivé
-- Concordia University
-- 2012-2013
------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity binary_correction_less is
port ( clk, clearn : in std_logic;
   su_flag : in std_logic;
   binary_in : in std_logic_vector(0 downto 0);
   binary_out1 : out std_logic_vector(0 downto 0);
   binary_out2 : out std_logic_vector(0 downto 0);
   set_up_flag : out std_logic;
   im_width : in unsigned(10 downto 0);
   numpixels : in unsigned(18 downto 0)
);
end binary_correction_less;

architecture behavior of binary_correction_less is

component binary_fifo
port ( clk: IN std_logic;
   rst: IN std_logic;
   din: IN std_logic_VECTOR(0 downto 0);
   wr_en: IN std_logic;
   rd_en: IN std_logic;
   dout: OUT std_logic_VECTOR(0 downto 0);
   full: OUT std_logic;
   empty: OUT std_logic;
   data_count: OUT std_logic_VECTOR(8 downto 0)
);
end component;

------------------------------------------------------------------------
-- Signal Declarations
------------------------------------------------------------------------
type data_win is array (0 to 4) of unsigned (0 downto 0);
signal row1 : data_win;
signal row2 : data_win;
signal row3 : data_win;
signal row4 : data_win;
signal row5 : data_win;
signal sbinary_in, sbinary_out1, sbinary_out2 : unsigned(0 downto 0);
signal data_in1 : unsigned (0 downto 0);
signal data_in2 : unsigned (0 downto 0);
signal data_in3 : unsigned (0 downto 0);
signal data_in4 : unsigned (0 downto 0);
signal t_setup : unsigned (18 downto 0);
signal activated : std_logic;

signal data_count1, data_count2, data_count3, data_count4 : std_logic_vector(8 downto 0);
signal fifo_size : unsigned(10 downto 0);
signal wr_en, rd_en1, rd_en2, rd_en3, rd_en4, rst : std_logic;

begin
  fifo1 : binary_fifo
  port map (  
    clk => clk,  
    rst => rst,  
    din => std_logic_vector(data_in1),  
    wr_en => wr_en,  
    rd_en => rd_en1,  
    dout => data_out1,  
    full => open,  
    empty => open,  
    data_count => data_count1);

  fifo2 : binary_fifo
  port map (  
    clk => clk,  
    rst => rst,  
    din => std_logic_vector(data_in2),  
    wr_en => wr_en,  
    rd_en => rd_en2,  
    dout => data_out2,  
    full => open,  
    empty => open,  
    data_count => data_count2);

  fifo3 : binary_fifo
  port map (  
    clk => clk,  
    rst => rst,  
    din => std_logic_vector(data_in3),  
    wr_en => wr_en,  
    rd_en => rd_en3,  
    dout => data_out3,  
    full => open,  
    empty => open,  
    data_count => data_count3);

  fifo4 : binary_fifo
  port map (  
    clk => clk,  
    rst => rst,  
    din => std_logic_vector(data_in4),  
    wr_en => wr_en,  
    rd_en => rd_en4,  
    dout => data_out4,  
    full => open,  
    empty => open,  
    data_count => data_count4);

------------------------------------------------------------------------
-- Module Implementation
------------------------------------------------------------------------

process (clk)
begin
  if (clk'event and clk = '1') then
    if (clearn = '0') then  
      t_setup <= (others => '0');  
    else
      if su_flag = '1' and t_setup < (numpixels+im_width*2+3) then  
        t_setup <= t_setup + 1;  
      end if;
    end if;
  end if;
end process;

process (clk)
variable sync_cnt : integer range 0 to 6 := 0;
begin
  if (clk'event and clk = '1') then  
    --initialization

UFC
if (clearn = '0') then
  set_up_flag <= '0';
  activated <= '0';
  for j in 0 to 4 loop -- fifo reset
    row1(j) <= (others => '0');
    row2(j) <= (others => '0');
    row3(j) <= (others => '0');
    row4(j) <= (others => '0');
    row5(j) <= (others => '0');
  end loop;

  sbinary_out1 <= (others => '0');
  sbinary_out2 <= (others => '0');
  data_in1 <= (others => '0');
  data_in2 <= (others => '0');
  data_in3 <= (others => '0');
  data_in4 <= (others => '0');
  rd_en1 <= '0';
  rd_en2 <= '0';
  rd_en3 <= '0';
  rd_en4 <= '0';
elsif su_flag = '1' then --Shifts all the registers and fifo's data one position
  row1(0) <= sbinary_in;
  row1(1 to 4) <= row1(0 to 3);
  data_in1 <= row1(4);
  if (unsigned(data_count1) >= fifo_size) then --Maintain a constant amount of data in the fifo
    rd_en1 <= '1';
    row2(0) <= unsigned(data_out1);
  else
    rd_en1 <= '0';
    row2(0) <= (others=>'0');
  end if;
  row2(1 to 4) <= row2(0 to 3);
  data_in2 <= row2(4);
  if (unsigned(data_count2) >= fifo_size) then
    rd_en2 <= '1';
    row3(0) <= unsigned(data_out2);
  else
    rd_en2 <= '0';
    row3(0) <= (others=>'0');
  end if;
  row3(1 to 4) <= row3(0 to 3);
  data_in3 <= row3(4);
  if (unsigned(data_count3) >= fifo_size) then
    rd_en3 <= '1';
    row4(0) <= unsigned(data_out3);
  else
    rd_en3 <= '0';
    row4(0) <= (others=>'0');
  end if;
  row4(1 to 4) <= row4(0 to 3);
  data_in4 <= row4(4);
  if (unsigned(data_count4) >= fifo_size) then
    rd_en4 <= '1';
    row5(0) <= unsigned(data_out4);
  else
    rd_en4 <= '0';
    row5(0) <= (others=>'0');
  end if;
  row5(1 to 4) <= row5(0 to 3);
  if t_setup >= (im_width*2+3) and t_setup < (numpixels+im_width*2+2) then -- +3-1+9 en realitat
    set_up_flag <= '1';
    activated <= '1'; --Apply the convolution operation for the current pixel and its window.
    sync_cnt := 0;
  --Mask with broader filtering--
  if (row3(2)= 0 and (((resize(not(row2(1)),4)+not(row2(2))+not(row2(3)))+
                     not(row3(1))+not(row3(3))+
                     not(row4(1))+not(row4(2))+not(row4(3))) < 4) and
FPGA Implementation of a Contrast Enhancement Algorithm

not((row2(1)=0 and row2(2)=0 and row2(3)=1 and row3(1)=0 and row3(3)=1) and row4(1)=1 and row4(2)=1 and row4(3)=1) or (row2(1)=1 and row2(2)=0 and row2(3)=0 and row3(1)=1 and row3(3)=0 and row4(1)=1 and row4(2)=1 and row4(3)=1) or (row2(1)=1 and row2(2)=1 and row2(3)=1 and row3(1)=1 and row3(3)=0 and row4(1)=1 and row4(2)=0 and row4(3)=0) or (row2(1)=1 and row2(2)=1 and row2(3)=1 and row3(1)=0 and row3(3)=1 and row4(1)=0 and row4(2)=0 and row4(3)=1))) then

sbinary_out1 <= "1";
else

sbinary_out1 <= row3(2);
end if;

--Mask with more restricted filtering--
if (row3(2)= 0 and ((resize(not(row2(1)),4)+not(row2(2))+not(row2(3))+not(row3(1))+not(row3(3))+not(row4(1))+not(row4(2))+not(row4(3))) < 3)) then

sbinary_out2 <= "1";
else

sbinary_out2 <= row3(2);
end if;
else

if activated <= '1' then

--if sync_cnt < 6 then
-- sync_cnt := sync_cnt + 1;
--else
set_up_flag <= '0'; --Finish
--end if;
end if;
end if;
end if;
end if;
end if;
end if;
end process;

wr_en <= su_flag;
fifo_size <= im_width - 8; -- Added a -1 initially not forecasted
rst <= not(clearn);

sbinary_in <= unsigned(binary_in);
binary_out1 <= std_logic_vector(sbinary_out1);
binary_out2 <= std_logic_vector(sbinary_out2);

end behavior;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mask_generator is
  port(
  --global clock signal, active with its rising edge
  clk: in std_logic;
  --Reset signal, synchronous and active high
  reset: in std_logic;
  --Trigger to start the generation
  pulse_start_input: in std_logic;
  limit1_t : in unsigned(7 downto 0);
  limit1_b : in unsigned(7 downto 0);
  limit2_t : in unsigned(7 downto 0);
  limit2_b : in unsigned(7 downto 0);
  end_flag: out std_logic;
  rom_addrb : out std_logic_vector(17 downto 0);
  rom_doutb : in std_logic_vector(7 downto 0);
  im_width: in unsigned(9 downto 0);
  numpixels: in unsigned(18 downto 0);
  binary_wea: out std_logic_vector(0 downto 0);
  binary_addra: out std_logic_vector(17 downto 0);
  binary1_dina: out std_logic_vector(0 downto 0);
  binary2_dina: out std_logic_vector(0 downto 0)
  );
end mask_generator;

architecture bench of mask_generator is
  signal device_data, data_in, filter_out1, filter_out2 : std_logic_vector(0 downto 0); --current pixel value
  signal ram_wr_addr, ram_wr_addr2 : unsigned(17 downto 0); --address to be accessed in the RAM containing the histogram
  signal pulse_out, pulse_out2, pulse_out3, end_flag_signal : std_logic;
  signal new_width, width_counter, width_counter2 : unsigned(10 downto 0);
  signal image_addr: unsigned(18 downto 0);
  signal pre_binary : std_logic_vector(7 downto 0);
  signal nrst : std_logic;
  signal wren : std_logic_vector(0 downto 0);
  signal new_numpixels : unsigned(18 downto 0);

  component binary_correction_less is
    port ( clk, clearn : in std_logic;
      su_flag : in std_logic;
      binary_in : in std_logic_vector(0 downto 0);
      binary_out1 : out std_logic_vector(0 downto 0);
      binary_out2 : out std_logic_vector(0 downto 0);
      set_up_flag : out std_logic;
      im_width : in unsigned(10 downto 0);
      numpixels : in unsigned(18 downto 0)
    );
  end component;

begin
  corrector_1 : binary_correction_less
    port map( 
      clk => clk,
      clearn => nrst,
      su_flag => su_flag,
      binary_in => binary_in,
      binary_out1 => binary_out1,
      binary_out2 => binary_out2,
      set_up_flag => set_up_flag,
      im_width => im_width,
      numpixels => numpixels);
su_flag => pulse_out3,
binary_in => device_data,
binary_out1 => filter_out1,
binary_out2 => filter_out2,
set_up_flag => wren(0),
im_width => new_width,
numpixels => new_numpixels);

process(clk) --Process containing an address counter to read the image in the
--ROM memory sequentially and compute its histogram or transformed version.
begin
  if (CLK'EVENT AND CLK = '1') then
    if reset = '1' or pulse_out='0' then
      image_addr <= to_unsigned(0, 19);
      width_counter <= (others=>'0');
      device_data <= (others=>'0');
    elsif unsigned(image_addr) >= (unsigned(numpixels)+1) then
      image_addr <= image_addr;
      width_counter <= width_counter;
      device_data <= (others=>'0');
    elsif width_counter = (im_width) then
      image_addr <= image_addr;
      width_counter <= width_counter + 1;
      device_data <= data_in;
    elsif width_counter = (im_width+1) then
      image_addr <= image_addr;
      width_counter <= width_counter + 1;
      device_data <= (others=>'0');
    elsif width_counter = (new_width) then
      image_addr <= image_addr+1;
      width_counter <= width_counter + 1;
      width_counter <= "00000000001";
      device_data <= (others=>'0');
    else
      image_addr <= image_addr + 1;
      width_counter <= width_counter + 1;
      device_data <= data_in;
    end if;
    if reset='1' then
      pulse_out <= '0';
      pulse_out2 <= '0';
      pulse_out3 <= '0';
    else
      pulse_out<=pulse_start_input or pulse_out;
      pulse_out2 <= pulse_out;
      pulse_out3 <= pulse_out2;
    end if;
  end if;
  if (CLK'EVENT AND CLK = '1') then
    if reset = '1' or wren="0" then
      ram_wr_addr <= to_unsigned(0, 18);
      width_counter2 <= (others=>'0');
      binary1_dina <= (others=>'0');
    elsif unsigned(ram_wr_addr) >= (unsigned(numpixels) - 1) then
      ram_wr_addr <= ram_wr_addr;
      width_counter2 <= width_counter2;
      binary1_dina <= filter_out1;
    elsif width_counter2 = (im_width) or width_counter2 = (im_width + 1) then
      ram_wr_addr <= ram_wr_addr;
      width_counter2 <= width_counter2 + 1;
      binary1_dina <= filter_out1;
    else
      width_counter2 = (im_width + 2) then
FPGA Implementation of a Contrast Enhancement Algorithm

```vhdl
ram_wr_addr <= ram_wr_addr + 1;
width_counter2 <= "000000001";
binary1_dina <= filter_out1;
binary2_dina <= filter_out2;
elsec
ram_wr_addr <= ram_wr_addr + 1;
width_counter2 <= width_counter2 + 1;
binary1_dina <= filter_out1;
binary2_dina <= filter_out2;
end if;
if reset = '1' then
  end_flag_signal <= '0';
elsif unsigned(ram_wr_addr) >= (unsigned(numpixels) - 1) then
  end_flag_signal <= '1';
else
  end_flag_signal <= end_flag_signal;
end if;
ram_wr_addr2 <= ram_wr_addr;
end if;
end process;

nrst <= not(reset);
new_width <= resize(im_width, 11) + 2;
nw_numpixels <= numpixels + resize(2*numpixels/im_width, 19);
data_in(0) <= '1' when ((unsigned(pre_binary) >= limit1_b) and (limit1_t >= unsigned(pre_binary))) or ((unsigned(pre_binary) >= limit2_b) and (limit2_t >= unsigned(pre_binary))) else '0';
pre_binary <= rom_doutb;
rom_addrb <= std_logic_vector(image_addr(17 downto 0));
binary_wea <= wren;
binary_addra <= std_logic_vector(ram_wr_addr2);
end_flag <= end_flag_signal;
end bench;
```
FPGA Implementation of a Contrast Enhancement Algorithm

clahe_complete4.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

dentity clahe is
  port (  
    clk : in std_logic;
    rst : in std_logic;
    start_cntr : in std_logic; --marks the end of the histogram calculation
    end_flag : out std_logic;  
    numpixels : in unsigned(18 downto 0);
    x_size : in unsigned(9 downto 0);
    y_size : in unsigned(9 downto 0);
    clip_limit : in unsigned(6 downto 0);
    rom_addra : out std_logic_vector(17 downto 0);
    rom_douta : in std_logic_vector(7 downto 0);
    ram_wea : out std_logic_vector(0 downto 0);
    ram_addra : out std_logic_vector(17 downto 0);
    ram_dina : out std_logic_vector(7 downto 0);
  );
end clahe;

architecture wrapper of clahe is

component histo_ram2 --Memory to store the resulting histogram
  port (  
    clka: IN std_logic;
    wea: IN std_logic_VECTOR(0 downto 0);
    addra: IN std_logic_VECTOR(7 downto 0);
    dina: IN std_logic_VECTOR(18 downto 0);
    clkb: IN std_logic;
    rstb: IN std_logic;
    addrb: IN std_logic_VECTOR(7 downto 0);
    doutb: OUT std_logic_VECTOR(18 downto 0));
end component;

compoment clipping_wrapper

  port (  
    clk : in std_logic;
    rst : in std_logic;
    start_cntr : in std_logic; --triggers the beginning of the operation
    numpixels : in unsigned(18 downto 0); --Total number of pixels in the image
    clip_limit : in unsigned(6 downto 0); --Tolerated bin limit
    histo_wea : out std_logic_vector(0 downto 0);
    histo_addra : out std_logic_vector(7 downto 0);
    histo_dina : out std_logic_vector(18 downto 0);
    histo_addrb : out std_logic_vector(7 downto 0);
    histo_doutb : in std_logic_vector(18 downto 0);
    cdf_min : out unsigned(18 downto 0)
  );
end component;

compoment histogram_wrapper

  port (  
    clk: in std_logic;
    reset: in std_logic;
    cntr_value: in std_logic_vector(18 downto 0);
    pulse_start_input: in std_logic;
    im_douta : in std_logic_vector(7 downto 0);
    histo_dina : out std_logic_vector(18 downto 0);
    histo_addra : out std_logic_vector(7 downto 0);
  );
end component;
component image_tiling
port (  romraddr : out std_logic_vector(17 downto 0) ; -- device data as address for RAM
datain : in std_logic_vector (7 downto 0); -- RAM data out
clk : in std_logic;
ramwaddr : out std_logic_vector(17 downto 0); -- RAM data in
numx : out unsigned(2 downto 0);
numy : out unsigned(2 downto 0);
rst : in std_logic;
start_cntr : in std_logic;
end_flag : out std_logic; --marks the end of the distribution
numpixels : in unsigned(18 downto 0); --number of pixels of the image
tile_numpixels : out unsigned(18 downto 0); --number of pixels of the tile
x_size : in unsigned(9 downto 0);
y_size : in unsigned(9 downto 0);
xtile : out unsigned(9 downto 0);
ytile : out unsigned(9 downto 0)
);
end component;

component histogram_equalizer is
port ( histo_raddr : out std_logic_vector(7 downto 0) ; -- device data as address for RAM
histo_in_ul : in std_logic_vector (18 downto 0); -- histogram CDF value
histo_in_ur : in std_logic_vector (18 downto 0); -- histogram CDF value
histo_in_ll : in std_logic_vector (18 downto 0); -- histogram CDF value
histo_in_lr : in std_logic_vector (18 downto 0); -- histogram CDF value
rom_raddr : out std_logic_vector(17 downto 0); --image pixel address
rom_in : std_logic_vector(7 downto 0);--image pixel value
clk : in std_logic;
clhe_wraddr : out std_logic_vector( 17 downto 0); --Address for the transformed pixel to write
rst : in std_logic;
start_cntr : in std_logic; --Triggers the transformation operations
wren : out std_logic;
clhe_out : out std_logic_vector(7 downto 0); -- Output for transformed pixel value
end_flag : out std_logic; --marks the end of the histogram calculation
numpixels : in unsigned(18 downto 0);
im_width : in unsigned(9 downto 0);
im_height : in unsigned(9 downto 0);
x_size : in unsigned(9 downto 0);--Subimage
y_size : in unsigned(9 downto 0);--Subimage
ul_id : out unsigned(7 downto 0);
ur_id : out unsigned(7 downto 0);
l1_id : out unsigned(7 downto 0);
lr_id : out unsigned(7 downto 0);
numpixels_ul : in unsigned(18 downto 0); --number of pixels of the image
histo_min_ul : in unsigned(18 downto 0); --Lowest CDF value of the histogram
numpixels_ur : in unsigned(18 downto 0); --number of pixels of the image
histo_min_ur : in unsigned(18 downto 0); --Lowest CDF value of the histogram
numpixels_ll : in unsigned(18 downto 0); --number of pixels of the image
histo_min_ll : in unsigned(18 downto 0); --Lowest CDF value of the histogram
numpixels_lr : in unsigned(18 downto 0); --number of pixels of the image
histo_min_lr : in unsigned(18 downto 0) --Lowest CDF value of the histogram
);
end component;

type HISTO_IO is array (0 to 99) of std_logic_vector(18 downto 0);
type HISTO_ADDR is array (0 to 99) of std_logic_vector(7 downto 0);
type HISTO_WEA_T is array (0 to 99) of std_logic_vector(0 downto 0);
type TILESIZE is array (0 to 99) of unsigned(18 downto 0);
signal histo_wea1, histo_wea2, wren_trans : std_logic_vector(0 downto 0);
signal histo_wea : HISTO_WEA_T;
signal histo_addr1, histo_addr2, histo_addr3, histo_addr4, tiler_datain, tiler_dataout_a,
histo_raddr_trans, rom_in_trans, clhe_out_trans : std_logic_vector (7 downto 0);
signal histo_addr, histo_addrb : HISTO_ADDR;
signal tiler_ramaddr, rom_raddr_trans, clhe_wraddr_trans : std_logic_vector (17 downto 0);
signal histo_dina1, histo_dina2, histo_doutb1, histo_doutb2, histo_in_ul_trans, histo_in_ur_trans, histo_in_ll_trans, histo_in_lr_trans : std_logic_vector (18 downto 0);
signal histo_dina, histo_doutb : HISTO_IO;
signal start_clipping, start_clipping_pre, start_clipping_pre2, rstclip, start_cntr_trans : std_logic;
signal histo_start_cntr_pre, histo_start_cntr, end_flag_clipper, tiler_start_cntr, tiler_wren, tiler_wren_pre, tiler_end_flag, end_flag_trans, transform, rst_trans : std_logic;
signal histo_min, numx : unsigned(2 downto 0);
signal numtile_int : integer range 0 to 99;
signal numpixels_tile, numpixels_tile_pre, cdfmin, numpixels_ul_trans, numpixels_ur_trans, numpixels_ll_trans, numpixels_lr_trans, histo_min_ul_trans, histo_min_ur_trans, histo_min_ll_trans, histo_min_lr_trans : unsigned(18 downto 0);
signal tile_numpixels, cdf_min : TILESIZE;
signal x_size_sub, y_size_sub : unsigned(9 downto 0);
signal ul_id_trans, ur_id_trans, ll_id_trans, lr_id_trans : unsigned(7 downto 0);

begin
--Connections between all the memory blocks and the computation block

tiles : for I in 0 to 99 generate
  histogram_ram : histo_ram2
  port map (  
    clka => clk,  
    dina => histo_dina(I),  
    addra => histo_addra(I),  
    wea => histo_wea(I),  
    clkb => clk,  
    rstb => '0',  
    addrb => histo_addrb(I),  
    doutb => histo_doutb(I)),
end generate;

histo_wrapper : histogram_wrapper
  port map(  
    --global clock signal, active with its rising edge  
    clk => clk,  
    --reset signal, synchronous and active high  
    reset => rst,  
    --number of pixels of the image  
    cntr_value => std_logic_vector(numpixels_tile_pre),  
    --Trigger to start the histogram generation  
    pulse_start_input => histo_start_cntr,--tiler_wren... IMPORTANT! TODO!  
    --Output of the histogram data  
    im_douta => tiler_dataout_a,  
    histo_dina => histo_dina1,  
    histo_addra => histo_addra1,  
    histo_wea => histo_wea1,  
    histo_addrb => histo_addrb1,  
    histo_doutb => histo_doutb1,  
    end_flag => start_clipping_pre2)
);

histo_clipper : clipping_wrapper
  port map (  
    clk => clk,  
    rst => rstclip,  
    start_cntr => start_clipping,  
    end_flag => end_flag_clipper, --marks the end of the histogram clipping  
    numpixels => numpixels_tile_pre,  
    clip_limit => clip_limit, --Tolerated bin limit  
    histo_wea => histo_wea2,  
    histo_addra => histo_addra2,  
    histo_dina => histo_dina2,  
    histo_addrb => histo_addrb2,  
    histo_doutb => histo_doutb2,  
    cdf_min => cdfmin
);
Concordia University and Universitat Politècnica de Catalunya
FPGA Implementation of a Contrast Enhancement Algorithm

tiler : image_tiling
port map (  
romraddr => tiler_ramraddr,  
datain => tiler_datain,  
clk => clk,  
ramwraddr => open,  
dataout => tiler_dataout_a,  
umx => numx,  
umy => numy,  
rst => rst,  
start_cntr => tiler_start_cntr,  
wren => tiler_wren,  
end_flag => tiler_end_flag,  
numpixels => numpixels,  
tile_numpixels => numpixels_tile_pre,  
x_size => x_size,  
y_size => y_size,  
xtile => x_size_sub,  
ytile => y_size_sub
);

equalizer : histogram_equalizer
port map (  
histo_raddr => histo_raddr_trans, -- device data as address for RAM  
histo_in_ul => histo_in_ul_trans, -- histogram CDF value  
histo_in_ur => histo_in_ur_trans, -- histogram CDF value  
histo_in_ll => histo_in_ll_trans, -- histogram CDF value  
histo_in_lr => histo_in_lr_trans, -- histogram CDF value  
rom_raddr => rom_raddr_trans, --image pixel address  
rom_in => rom_in_trans,--image pixel value  
clk => clk,  
clhe_wraddr => clhe_wraddr_trans, --Address for the transformed pixel to  
write  
rst => rst_trans,  
start_cntr => start_cntr_trans, --Triggers the transformation operations  
wren => wren_trans(0),  
clhe_out => clhe_out_trans, -- Output for transformed pixel value  
end_flag => end_flag_trans, --marks the end of the histogram calculation  
numpixels => numpixels,  
im_width => x_size,  
im_height => y_size,  
x_size => x_size_sub,--Subimage  
y_size => y_size_sub,--Subimage  
ul_id => ul_id_trans,  
ur_id => ur_id_trans,  
ll_id => ll_id_trans,  
lr_id => lr_id_trans,  
numpixels_ul => numpixels_ul_trans,--number of pixels of the image  
histo_min_ul => histo_min_ul_trans, --Lowest CDF value of the histogram  
numpixels_ur => numpixels_ur_trans,--number of pixels of the image  
histo_min_ur => histo_min_ur_trans, --Lowest CDF value of the histogram  
numpixels_ll => numpixels_ll_trans,--number of pixels of the image  
histo_min_ll => histo_min_ll_trans, --Lowest CDF value of the histogram  
numpixels_lr => numpixels_lr_trans,--number of pixels of the image  
histo_min_lr => histo_min_lr_trans); --Lowest CDF value of the histogram

process(clk)
begin
if (CLK'EVENT AND CLK = '1') then
if rst='1' then
start_clipping_pre <= '0';
start_clipping <= '0';
else
start_clipping_pre <= start_clipping_pre2;
start_clipping <= not(start_clipping_pre) and start_clipping_pre2;
end if;
end if;
end process;

process(rst, start_clipping_pre, histo_dina1, histo_dina2, histo_addr1, histo_addra2,  
histo_wea1, histo_wea2, histo_addrb1, histo_addrb2, histo_doutb, cdfmin, numpixels_tile,
FPGA Implementation of a Contrast Enhancement Algorithm

\[
\begin{align*}
\text{ul_id_trans, ur_id_trans, ll_id_trans, lr_id_trans, tiler_ramaddr, rom_raddr_trans, numtile_int, histo_raddr_trans, rom_douta} \\
\text{begin} \\
\text{if rst = '1' then} \\
\text{for j in 0 to 99 loop} \\
\text{histo_dina(j) <= (others=>'0');} \\
\text{histo_addra(j) <= (others=>'0');} \\
\text{histo_wea(j) <= (others=>'0');} \\
\text{histo_addrb(j) <= (others=>'0');} \\
\text{tile_numpixels(j) <= (others=>'0');} \\
\text{cdf_min(j) <= (others=>'0');} \\
\text{--histo_rstb <= histo_rstb1;} \\
\text{end loop;} \\
\text{histo_doutb1 <= (others=>'0');} \\
\text{histo_doutb2 <= (others=>'0');} \\
\text{numpixels_ul_trans <= (others=>'0');} \\
\text{numpixels_ur_trans <= (others=>'0');} \\
\text{numpixels_ll_trans <= (others=>'0');} \\
\text{numpixels_lr_trans <= (others=>'0');} \\
\text{histo_min_ul_trans <= (others=>'0');} \\
\text{histo_min_ur_trans <= (others=>'0');} \\
\text{histo_min_ll_trans <= (others=>'0');} \\
\text{histo_min_lr_trans <= (others=>'0');} \\
\text{histo_in_ul_trans <= (others=>'0');} \\
\text{histo_in_ur_trans <= (others=>'0');} \\
\text{histo_in_ll_trans <= (others=>'0');} \\
\text{histo_in_lr_trans <= (others=>'0');} \\
\text{rom_addra <= (others=>'0');} \\
\text{elsif transform = '0' then} \\
\text{for k in 0 to 9 loop --TODO: haig de fer que s'escriguin a l'hora que la resta (o no) } \\
\text{histo_dina(k) <= histo_dina(k+10);} \\
\text{histo_addra(k) <= histo_addra(k+10);} \\
\text{histo_wea(k) <= histo_wea(k+10);} \\
\text{histo_addrb(k) <= histo_addrb(k+10);} \\
\text{tile_numpixels(k) <= tile_numpixels(k+10);} \\
\text{cdf_min(k) <= cdf_min(k+10);} \\
\text{end loop;} \\
\text{for l in 90 to 99 loop} \\
\text{histo_dina(l) <= histo_dina(l-10);} \\
\text{histo_addra(l) <= histo_addra(l-10);} \\
\text{histo_wea(l) <= histo_wea(l-10);} \\
\text{histo_addrb(l) <= histo_addrb(l-10);} \\
\text{tile_numpixels(l) <= tile_numpixels(l-10);} \\
\text{cdf_min(l) <= cdf_min(l-10);} \\
\text{end loop;} \\
\text{for m in 1 to 8 loop} \\
\text{histo_dina(10*m) <= histo_dina(10*m+1);} \\
\text{histo_addra(10*m) <= histo_addra(10*m+1);} \\
\text{histo_wea(10*m) <= histo_wea(10*m+1);} \\
\text{histo_addrb(10*m) <= histo_addrb(10*m+1);} \\
\text{tile_numpixels(10*m) <= tile_numpixels(10*m+1);} \\
\text{cdf_min(10*m) <= cdf_min(10*m+1);} \\
\text{end loop;} \\
\text{for n in 2 to 9 loop} \\
\text{histo_dina(10*n-1) <= histo_dina(10*n-2);} \\
\text{histo_addra(10*n-1) <= histo_addra(10*n-2);} \\
\text{histo_wea(10*n-1) <= histo_wea(10*n-2);} \\
\text{histo_addrb(10*n-1) <= histo_addrb(10*n-2);} \\
\text{tile_numpixels(10*n-1) <= tile_numpixels(10*n-2);} \\
\text{cdf_min(10*n-1) <= cdf_min(10*n-2);} \\
\text{end loop;} \\
\end{align*}
\]
for p in 1 to 8 loop
    for q in 1 to 8 loop
        if (p*10+q)=numtile_int then
            if start_clipping_pre = '1' then
                histo_dina(p*10+q) <= histo_dina2;
                histo_addra(p*10+q) <= histo_addra2;
                histo_wea(p*10+q) <= histo_wea2;
                histo_addrb(p*10+q) <= histo_addrb2;
                tile_numpixels(p*10+q) <= tile_numpixels(p*10+q);
                cdf_min(p*10+q) <= cdfmin;
            else
                histo_dina(p*10+q) <= histo_dina1;
                histo_addra(p*10+q) <= histo_addra1;
                histo_wea(p*10+q) <= histo_wea1;
                histo_addrb(p*10+q) <= histo_addrb1;
                tile_numpixels(p*10+q) <= numpixels_tile;
                cdf_min(p*10+q) <= cdfmin;
            end if;
        else
            histo_dina(p*10+q) <= (others=>'0');
            histo_addra(p*10+q) <= (others=>'0');
            histo_wea(p*10+q) <= (others=>'0');
            histo_addrb(p*10+q) <= (others=>'0');
            tile_numpixels(p*10+q) <= tile_numpixels(p*10+q);
            cdf_min(p*10+q) <= cdf_min(p*10+q);
        end if;
    end loop;
end loop;

if start_clipping_pre = '1' then
    histo_doutb1 <= (others=>'0');
    histo_doutb2 <= histo_doutb(numtile_int);
else
    histo_doutb1 <= histo_doutb(numtile_int);
    histo_doutb2 <= (others=>'0');
end if;

numpixels_ul_trans <= (others=>'0');
numpixels_ur_trans <= (others=>'0');
numpixels_ll_trans <= (others=>'0');
numpixels_lr_trans <= (others=>'0');
histo_min_ul_trans <= (others=>'0');
histo_min_ur_trans <= (others=>'0');
histo_min_ll_trans <= (others=>'0');
histo_min_lr_trans <= (others=>'0');
histo_in_ul_trans <= (others=>'0');
histo_in_ur_trans <= (others=>'0');
histo_in_ll_trans <= (others=>'0');
histo_in_lr_trans <= (others=>'0');

rom_in_trans <= (others=>'0');

tiler_datain <= rom_douta;
rom_addra <= tiler_ramraddr;
else
for p in 0 to 9 loop
    for q in 0 to 9 loop
        histo_dina(p*10+q) <= (others=>'0');
        histo_addra(p*10+q) <= (others=>'0');
        histo_wea(p*10+q) <= (others=>'0');
        histo_addrb(p*10+q) <= histo_raddr_trans;
        tile_numpixels(p*10+q) <= tile_numpixels(p*10+q);
        cdf_min(p*10+q) <= cdf_min(p*10+q);
    end loop;
end loop;

numpixels_ul_trans <= tile_numpixels(to_integer(ul_id_trans));
numpixels_ur_trans <= tile_numpixels(to_integer(ur_id_trans));
numpixels_ll_trans <= tile_numpixels(to_integer(ll_id_trans));
numpixels_lr_trans <= tile_numpixels(to_integer(lr_id_trans));

histo_min_ul_trans <= cdf_min(to_integer(ul_id_trans));
histo_min_ur_trans <= cdf_min(to_integer(ur_id_trans));
histo_min_ll_trans <= cdf_min(to_integer(ll_id_trans));
histo_min_lr_trans <= cdf_min(to_integer(lr_id_trans));

histo_in_ul_trans <= histo_doutb(to_integer(ul_id_trans));
histo_in_ur_trans <= histo_doutb(to_integer(ur_id_trans));
histo_in_ll_trans <= histo_doutb(to_integer(ll_id_trans));
histo_in_lr_trans <= histo_doutb(to_integer(lr_id_trans));

histo_doutb1 <= (others=>'0');
histo_doutb2 <= (others=>'0');

rom_in_trans <= rom_douta;

tiler_datain <= (others=>'0');

rom_addra <= rom_raddr_trans;
end if;
end process;

----Processes for tiler
process(clk)
begin
    if (CLK'EVENT AND CLK = '1') then
        if rst='1' then
            tiler_start_cntr <= '0';
            histo_start_cntr_pre <= '0';
            histo_start_cntr <= '0';
            tiler_wren_pre <= '0';
        elsif tiler_end_flag='1' then
            tiler_start_cntr <= '0';
            histo_start_cntr_pre <= tiler_start_cntr;
            histo_start_cntr <= histo_start_cntr_pre;
        else
            tiler_start_cntr <= start_cntr or (end_flag_clipper and start_clipping_pre);
            histo_start_cntr_pre <= tiler_start_cntr;
            histo_start_cntr <= histo_start_cntr_pre;
        end if;
    end if;
end process;
if rst='1' then
    numtile_int <= 11;
elsif histo_start_cntr = '1' then
    numtile_int <= to_integer(numx)+11+to_integer(numy)*10;
else
    numtile_int <= numtile_int;
end if;

if rst='1' then
    numpixels_tile <= (others=>'0');
    transform <= '0';
    start_cntr_trans <= '0';
elsif histo_start_cntr = '1' then
    numpixels_tile <= numpixels_tile_pre +1;
    transform <= start_cntr_trans or transform;
    start_cntr_trans <= (end_flag_clipper and tiler_end_flag);
end if;

end if;
end process;

end if;

---End of tiler processes

rstclip <= rst;
rst_trans <= rst or not(transform);
end_flag <= end_flag_trans;

ram_wea <= wren_trans;
ram_addra <= clhe_wraddr_trans;
ram_dina <= clhe_out_trans;

end wrapper;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity histogram_clipper is
  port (  ramraddr : out std_logic_vector(7 downto 0); -- device data as address for RAM
datain : in std_logic_vector (18 downto 0); -- RAM data out
clk : in std_logic;
ramwraddr : out std_logic_vector( 7 downto 0); -- written histogram bin as address
for RAM
rst : in std_logic;
start_cntr : in std_logic;--triggers the beginning of the operation
wren : out std_logic;--write enable output for the ram
dataout : out std_logic;--write enable output for the ram
end_flag : out std_logic; --marks the end of the histogram calculation
numpixels : in unsigned(18 downto 0); --number of pixels of the histogram
clip_limit : in unsigned(6 downto 0); -- Clip limit in %
cdf_min : out unsigned(18 downto 0));
end histogram_clipper;

architecture clipper of histogram_clipper is
  --Signal list
  signal start_add, pre_start_add, pre_wren, pre_wren2 : std_logic;
  signal excess, abs_limit, data_out, cdf_min_signal : unsigned(18 downto 0);
  signal binIncr : unsigned(26 downto 0); --Potser es podria reduir la mida?
  signal ramraddru, ramwraddru, ramwraddru2 : unsigned(7 downto 0);
  signal difference : unsigned(19 downto 0);
  signal incr_trigger, start, start2, operation, wren1, end_flag_signal : std_logic;

begin
  --Process to read all the histogram, calculate the amount of absolute clipping and
generate the CDF for the transformation.
  -- This is done in 2 sweeps:
  --The first sweep calculates the amount of pixels that exceed the clip limit and clips
  them from the corresponding bins.
  --The second sweep adds to each bin the increase calculated from those counted excess
  pixels and substitutes the bin value
  --with the corresponding CDF value.
  process(clk)
  begin
    if (CLK'EVENT AND CLK = '1') then
      if rst = '1' then
        start_add <= '0';
        pre_start_add <= '0';
      else
        start_add <= start2;
        pre_start_add <= start_add; --Delay the beginning of the operation start
      end if;
    signal_until
    end if;
  --it is aligned with input data
  ---------------------------------------------beginning of reading block, shared amongst the 2 sweeps--------
---------------------------------------------

  if rst = '1' or start2='0' then --Initialization of variables
    ramraddru <= to_unsigned(0, 8);
    ramwraddru <= to_unsigned(0, 8);
    ramwraddru2 <= to_unsigned(0, 8);
    pre_wren <= '0';
    pre_wren2 <= '0';
    wren1 <= '0';
    incr_trigger <= '0';
    end_flag_signal <= '0';
elsif ramraddr2 = 255 then --Don't let it keep writing if the whole memory has
been swept.
    ramraddr2 <= to_unsigned(255, 8);
    ramwaddr2 <= ramraddr;
    ramwaddr <= ramwaddr2;--align the write addresses and write enable with
the output data
    pre_wren2 <= '0';
    pre_wren <= pre_wren2;
    wren1 <= pre_wren;
    incr_trigger <= operation; --will turn to 1 only the first time we do the
memory sweep
    end_flag_signal <= not(operation);
else
    ramraddr <= ramraddr + 1; --Sweep the addresses
    ramwaddr2 <= ramwaddr;--align the write addresses and write enable with
the output data
    ramwaddr <= ramwaddr2;
    pre_wren2 <= '1';
    pre_wren <= '1';
    wren1 <= pre_wren;
    incr_trigger <= '0';
    end_flag_signal <= '0';
end if;
---------------------------------end of reading block, beginning of writing block--------------
if (start_add='0' and operation='1') or rst='1' then --Clipping block: excess
variable counter management
    excess <= to_unsigned(0, 19);
elsif operation = '1' then
    if difference >= to_signed(1,19) and pre_wren = '1' then --wren condition
here is to avoid increases in
    excess <= excess + resize(unsigned(difference), 18); --the excess
variable after finishing the sweep.
else
    excess <= excess;
end if;
else
    excess<=excess;
end if;
if start_add='0' or rst='1' then --Clipping block: data output management
(clipped output or CDF value, according to
    data_out <= (others=>'0');--the sweep number.
estif operation = '1' then
    if difference >= to_signed(1,19) and pre_wren = '1' then --wren condition
here is to avoid increases in
    data_out <= abs_limit;      --the excess variable after
finishing the sweep.
else
    data_out <= unsigned(datain);
end if;
else
    data_out <= (data_out+resize((unsigned(datain) + binIncr),19));
end if;
if rst='1' then --Management of the start signal, used to trigger the shared
logic of the 2 sweeps
    start <= '0';
elsif (start_cntr = '1' or incr_trigger = '1') then
    start <= '1';
elsif (ramraddr = 255) then
    start <= '0';
else
    start<=start;
end if;
if rst='1' then
FPGA Implementation of a Contrast Enhancement Algorithm

start2 <= '0';
else
start2 <= start;
end if;

if rst='1' or start_cntr = '1' then --Management of the operation flag. Selects between the 2 sweeps
operation <= '1';
elsif incr_trigger = '1' then
operation <= '0';
else
operation <= operation;
end if;

if rst='1' then
end_flag <= '0';
else
end_flag <= end_flag_signal;
end if;

if rst='1' then
cdf_min_signal <= (others=>'0');
elsif operation = '0' and ramwraddru = 0 and wren1='1' then
cdf_min_signal <= unsigned(data_out);
else
cdf_min_signal <= cdf_min_signal;
end if;
end if;
end process;

wren <= wren1;
ramraddr <= std_logic_vector(ramraddru);
ramwraddr <= std_logic_vector(ramwraddru);

---debug---
op1 <= (numpixels*clip_limit);
op2 <= op1/to_unsigned(100, 7);
abs_limit <= resize(op2, 19);
---fi debug---
difference <= signed('0' & datain) - signed('0'&abs_limit); --Computes the difference between bin value and clipping limit
binIncr <= resize(excess, 27)/256;
dataout <= std_logic_vector(data_out);
cdf_min <= cdf_min_signal;
end clipper;
clipping_wrapper_int2.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity clipping_wrapper is
  port (   clk : in std_logic;
            rst : in std_logic;
            start_cntr : in std_logic; -- triggers the beginning of the operation
            dataout : out std_logic_vector(17 downto 0); -- RAM data in
            end_flag : out std_logic; -- marks the end of the histogram clipping
            numpixels : in unsigned(18 downto 0); -- Total number of pixels in the image
            clip_limit : in unsigned(6 downto 0); -- Tolerated bin limit
            histo_wea : out std_logic_vector(0 downto 0);
            histo_addra : out std_logic_vector(7 downto 0);
            histo_dina : out std_logic_vector(18 downto 0);
            histo_addrb : out std_logic_vector(7 downto 0);
            histo_doutb : in std_logic_vector(18 downto 0);
            cdf_min : out unsigned(18 downto 0)
            );
end clipping_wrapper;

architecture wrapper of clipping_wrapper is

component histogram_clipper
  port (    ramraddr : out std_logic_vector(7 downto 0); -- accessed histogram bin as address for RAM
            datain : in std_logic_vector (18 downto 0); -- RAM data out
            clk : in std_logic;
            ramwraddr : out std_logic_vector( 7 downto 0); -- written histogram bin as address for RAM
            rst : in std_logic;
            start_cntr : in std_logic;-- triggers the beginning of the operation
            wren : out std_logic;-- write enable output for the ram
            dataout : out std_logic_vector(18 downto 0); -- RAM data in
            end_flag : out std_logic; -- marks the end of the histogram calculation
            numpixels : in unsigned(18 downto 0); -- number of pixels of the histogram
            clip_limit : in unsigned(6 downto 0); -- clip limit in %
            cdf_min : out unsigned(18 downto 0)
            );
end component;

signal ramraddr, ramwraddr : std_logic_vector(7 downto 0);
signal wren : std_logic_vector(0 downto 0);
signal datain, dataout_a : std_logic_vector (18 downto 0);

begin

  histo_clipper : histogram_clipper
    port map (   ramraddr => ramraddr,
                datain => datain,
                clk => clk,
                ramwraddr => ramwraddr,
                rst => rst,
                start_cntr => start_cntr,
                wren => wren(0),
                dataout => dataout_a,
                end_flag => end_flag,
                numpixels => numpixels,
                clip_limit => clip_limit,
                cdf_min => cdf_min);

  begin

    -- Process to read all the histogram and calculate the amount of absolute clipping

  histo_clipper : histogram_clipper
    port map (   ramraddr => ramraddr,
                datain => datain,
                clk => clk,
                ramwraddr => ramwraddr,
                rst => rst,
                start_cntr => start_cntr,
                wren => wren(0),
                dataout => dataout_a,
                end_flag => end_flag,
                numpixels => numpixels,
                clip_limit => clip_limit,
                cdf_min => cdf_min);

  begin

    -- Process to read all the histogram and calculate the amount of absolute clipping

  histo_clipper : histogram_clipper
    port map (   ramraddr => ramraddr,
                datain => datain,
                clk => clk,
histo_dina <= dataout_a;
histo_addra <= ramwraddr;
histo_wea <= wren;
histo_addrb <= ramraddr;
datain <= histo_doutb;

--dataout <= dataout_a;
end wrapper;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity filter_testbench is
port(
  -- global clock signal, active with its rising edge
  clk: in std_logic;
  -- reset signal, synchronous and active high
  reset: in std_logic;
  -- Trigger to start the generation
  pulse_start_input: in std_logic;
  -- Output of the data
  dataout : out std_logic_vector(7 downto 0);
  memout : out std_logic_vector(7 downto 0);
  end_flag : out std_logic;
  im_width : in unsigned(10 downto 0);
  numpixels : in unsigned(18 downto 0);
  b1_doutb : in std_logic_vector(0 downto 0);
  b2_doutb : in std_logic_vector(0 downto 0);
  input_ram_addr : out std_logic_vector(17 downto 0);
  clahe_ram_doutb : in std_logic_vector(7 downto 0)
);
end filter_testbench;

architecture bench of filter_testbench is

signal device_data, device_data_2, filter_out_s, filter_out_sf, filter_out_sf1,
    filter_out_sf2, filter_out_sf3, filter_out_sf4, filter_out_sf5, filter_out_sf6,
    filter_out_sf7, filter_out_sf8, filter_out_sf9, filter_out_l, filter_out_l2, filter_out_lf,
    filter_out_2 : std_logic_vector(7 downto 0); -- current pixel value
signal ram_wr_addr, ram_wr_addr2 : unsigned(17 downto 0); -- address to be accessed in the RAM
containing the histogram
signal pulse_out, pulse_out2, pulse_out3, end_flag_signal: std_logic;

begin
component smooth_filter is
  port ( clk, clearn : in std_logic;
       su_flag : in std_logic;
       smooth_filter_in : in std_logic_vector(7 downto 0);
       smooth_filter_out : out std_logic_vector(7 downto 0);
       set_up_flag : out std_logic;
       im_width : in unsigned(10 downto 0);
       numpixels : in unsigned(18 downto 0)
  );
end component;

component clhe_ram2
port ( clka: IN std_logic;
       wea: IN std_logic_VECTOR(0 downto 0);
       addra: IN std_logic_VECTOR(17 downto 0);
       dina: IN std_logic_VECTOR(7 downto 0);
       douta: OUT std_logic_VECTOR(7 downto 0));
end component;
component median_filter is
    port ( clk, clearn : in std_logic;
        su_flag : in std_logic;
        smooth_filter_in : in std_logic_vector(7 downto 0);
        smooth_filter_out : out std_logic_vector (7 downto 0);
        set_up_flag : out std_logic;
        im_width : in unsigned(10 downto 0);
        numpixels : in unsigned(18 downto 0);
    );
end component;

component wait_fifo
    port ( clk: IN std_logic;
            rst: IN std_logic;
            din: IN std_logic_VECTOR(7 downto 0);
            wr_en: IN std_logic;
            rd_en: IN std_logic;
            dout: OUT std_logic_VECTOR(7 downto 0);
            full: OUT std_logic;
            empty: OUT std_logic;
            data_count: OUT std_logic_VECTOR(11 downto 0);
        );
end component;

component binary1_wait_fifo
    port ( clk: IN std_logic;
            rst: IN std_logic;
            din: IN std_logic_VECTOR(0 downto 0);
            wr_en: IN std_logic;
            rd_en: IN std_logic;
            dout: OUT std_logic_VECTOR(0 downto 0);
            full: OUT std_logic;
            empty: OUT std_logic;
            data_count: OUT std_logic_VECTOR(11 downto 0);
        );
end component;

component binary2_wait_fifo
    port ( clk: IN std_logic;
            rst: IN std_logic;
            din: IN std_logic_VECTOR(0 downto 0);
            wr_en: IN std_logic;
            rd_en: IN std_logic;
            dout: OUT std_logic_VECTOR(0 downto 0);
            full: OUT std_logic;
            empty: OUT std_logic;
            data_count: OUT std_logic_VECTOR(11 downto 0);
        );
end component;

begin

  smoother : smooth_filter
    port map(
      clk => clk,
      clearn => nrst,
      su_flag => pulse_out3,
      smooth_filter_in => device_data,
      smooth_filter_out => filter_out_s,
      set_up_flag => wren_s(0),
      im_width => new_width,
      numpixels => new_numpixels);

  discriminative1 : median_filter
    port map(
      clk => clk,
      clearn => nrst,
      su_flag => wren_s(0),
      smooth_filter_in => filter_out_s,
      smooth_filter_out => filter_out_1,
      set_up_flag => wren_l(0),
      im_width => new_width,
      numpixels => new_numpixels);

  discriminative2 : median_filter
port map(
    clk => clk,
    clearn => nrst,
    su_flag => wren_1_2(0),
    smooth_filter_in => device_data_2,
    smooth_filter_out => filter_out_2,
    set_up_flag => wren_2(0),
    im_width => new_width,
    numpixels => new_numpixels);

waitf_1 : wait_fifo
  port map(
    clk => clk,
    rst => reset,
    din => filter_out_s,
    wr_en => wren_s(0),
    rd_en => wren_1(0),
    dout => filter_out_sf,
    full => open,
    empty => open,
    data_count => open);

waitf_2 : wait_fifo
  port map(
    clk => clk,
    rst => reset,
    din => device_data_2,
    wr_en => wren_1_2(0),
    rd_en => wren_2(0),
    dout => filter_out_1f,
    full => open,
    empty => open,
    data_count => open);

binary_wait1 : binary1_wait_fifo
  port map(
    clk => clk,
    rst => reset,
    din => b1_doutb,
    wr_en => pulse_out3,
    rd_en => wren_1(0),
    dout => binary1_data,
    full => open,
    empty => open,
    data_count => open);

binary_wait2 : binary2_wait_fifo
  port map(
    clk => clk,
    rst => reset,
    din => b2_doutb,
    wr_en => pulse_out3,
    rd_en => wren_2(0),
    dout => binary2_data,
    full => open,
    empty => open,
    data_count => open);

ram : clhe_ram2
  port map(
    clka => clk,
    wea => wren_2,
    addra => std_logic_vector(ram_wr_addr2),
    dina => data_ram,
    douta => memout);

process(clk) -- Process containing an address counter to read the image in the
-- ROM memory sequentially and compute its histogram or transformed version.
begin
  if (CLK'EVENT AND CLK = '1') then -- Part to read the source image. Adds zero padding
    in the process.
    if reset = '1' or pulse_out='0' then
      image_addr <= to_unsigned(0, 19);
      width_counter <= (others=>'0');
device_data <= (others=>'0'); --zeros for zero padding
elsif unsigned(image_addr) >= (unsigned(numpixels)+1) then --if end of the image
    image_addr <= image_addr;
    width_counter <= width_counter + 1;
    device_data <= data_in;
elsif width_counter = (im_width) then --if end of the row, stop the address
    image_addr <= image_addr;
    width_counter <= width_counter + 1;
    device_data <= data_in;
elsif width_counter = (im_width+1) then
    image_addr <= image_addr;
    width_counter <= width_counter + 1;
    device_data <= (others=>'0'); --zero padding
elsif width_counter = (new_width) then
    image_addr <= image_addr + 1;
    width_counter <= width_counter + 1;
    device_data <= data_in;
else
    image_addr <= image_addr + 1;
    width_counter <= width_counter + 1;
    device_data <= data_in;
end if;

pulse_out <= pulse_start_input;
pulse_out2 <= pulse_out;
pulse_out3 <= pulse_out2;

end if;

if (CLK'EVENT AND CLK = '1') then --Part to write the filtered image. It also
    if reset = '1' or wren_2="0" then--edges for zero padding, as they contain no
    ram_wr_addr <= to_unsigned(0, 18);
    width_counter2 <= (others=>'0');
    end_flag_signal <= '0';
    end_flag <= '0';
    elsif unsigned(ram_wr_addr) >= (unsigned(numpixels) - 1) then
        ram_wr_addr <= ram_wr_addr;
        width_counter2 <= width_counter2 + 1;
        data_out <= filter_out_2;
        end_flag_signal <= '1';
        end_flag <= end_flag_signal;
    elsif width_counter2 = (im_width) or width_counter2 = (im_width + 1) then
        ram_wr_addr <= ram_wr_addr + 1;
        width_counter2 <= width_counter2 + 1;
        data_out <= filter_out_2;
        end_flag <= end_flag_signal;
    elsif width_counter2 = (im_width + 2) then
        ram_wr_addr <= ram_wr_addr + 1;
        width_counter2 <= width_counter2 + 1;
        data_out <= filter_out_2;
        end_flag <= end_flag_signal;
    else
        ram_wr_addr <= ram_wr_addr + 1;
        width_counter2 <= width_counter2 + 1;
        data_out <= filter_out_2;
        end_flag <= end_flag_signal;
    end if;

    ram_wr_addr2 <= ram_wr_addr;

    if reset = '1' then
        filter_out_sf1 <= (others=>'0');
        filter_out_sf2 <= (others=>'0');
        filter_out_sf3 <= (others=>'0');
        filter_out_sf4 <= (others=>'0');
filter_out_sf5 <= (others=>'0');
filter_out_sf6 <= (others=>'0');
filter_out_sf7 <= (others=>'0');
filter_out_sf8 <= (others=>'0');
filter_out_sf9 <= (others=>'0');
filter_out_1_2 <= (others=>'0');
wren_1_2 <= (others=>'0');
else
  filter_out_sf1 <= filter_out_sf;
  filter_out_sf2 <= filter_out_sf1;
  filter_out_sf3 <= filter_out_sf2;
  filter_out_sf4 <= filter_out_sf3;
  filter_out_sf5 <= filter_out_sf4;
  filter_out_sf6 <= filter_out_sf5;
  filter_out_sf7 <= filter_out_sf6;
  filter_out_sf8 <= filter_out_sf7;
  filter_out_sf9 <= filter_out_sf8;
end if;
end if;
end process;
data_ram <= data_out when (binary2_data = "1") else filter_out_1f;
device_data_2 <= filter_out_1_2 when (binary1_data = "1") else filter_out_sf;
nrst <= not(reset);
new_width <= im_width+2;
new_numpixels <= numpixels+resize(2*numpixels/im_width, 19);

input_ram_addr <= std_logic_vector(image_addr(17 downto 0));
data_in <= clahe_ram_doutb;
dataout <= data_ram;

end bench;
FPGA Implementation of a Contrast Enhancement Algorithm

filter4c.vhd

-- Original smooth_filter: Núria Orduña
-- Modified by: Roger Olivé
-- Concordia University
-- 2012-2013

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity smooth_filter is
  port ( clk, clearn : in std_logic;
        su_flag : in std_logic;
        smooth_filter_in : in std_logic_vector(7 downto 0);
        smooth_filter_out : out std_logic_vector (7 downto 0);
        set_up_flag : out std_logic;
        im_width : in unsigned(10 downto 0);
        num_pixels : in unsigned(18 downto 0)
    );
end smooth_filter;

architecture behavior of smooth_filter is

component filter_fifo
  port ( clk: IN std_logic;
        rst: IN std_logic;
        din: IN std_logic_VECTOR(7 downto 0);
        wr_en: IN std_logic;
        rd_en: IN std_logic;
        dout: OUT std_logic_VECTOR(7 downto 0);
        full: OUT std_logic;
        empty: OUT std_logic;
        data_count: OUT std_logic_VECTOR(8 downto 0)
    );
end component;

-- Signal Declarations

type data_win is array (0 to 4) of unsigned (7 downto 0);
signal row1 : data_win;
signal row2 : data_win;
signal row3 : data_win;
signal row4 : data_win;
signal row5 : data_win;
signal data_in1 : unsigned (7 downto 0);
signal data_out1 : std_logic_vector (7 downto 0);
signal data_in2 : unsigned (7 downto 0);
signal data_out2 : std_logic_vector (7 downto 0);
signal data_in3 : unsigned (7 downto 0);
signal data_out3 : std_logic_vector (7 downto 0);
signal data_in4 : unsigned (7 downto 0);
signal data_out4 : std_logic_vector (7 downto 0);
signal t_setup : unsigned (18 downto 0);
signal activated : std_logic;
signal data_count1, data_count2, data_count3, data_count4 : std_logic_vector(8 downto 0);
signal fifo_size : unsigned(10 downto 0);
signal wr_en, rd_en1, rd_en2, rd_en3, rd_en4, rst : std_logic;

begin

fifol : filter_fifo
  port map (
clk => clk,
rst => rst,
din => std_logic_vector(data_in1),
wr_en => wr_en,
rd_en => rd_en1,
dout => data_out1,
full => open,
empty => open,
data_count => data_count1);
fifo2 : filter_fifo
port map (  
    clk => clk,
    rst => rst,
    din => std_logic_vector(data_in2),
    wr_en => wr_en2,
    rd_en => rd_en2,
    dout => data_out2,
    full => open,
    empty => open,
data_count => data_count2);
fifo3 : filter_fifo
port map (  
    clk => clk,
    rst => rst,
    din => std_logic_vector(data_in3),
    wr_en => wr_en3,
    rd_en => rd_en3,
    dout => data_out3,
    full => open,
    empty => open,
data_count => data_count3);
fifo4 : filter_fifo
port map (  
    clk => clk,
    rst => rst,
    din => std_logic_vector(data_in4),
    wr_en => wr_en4,
    rd_en => rd_en4,
    dout => data_out4,
    full => open,
    empty => open,
data_count => data_count4);

-- Module Implementation

process (clk)
begin
    if (clk’event and clk = ‘1’) then
        if (clearn = ‘0’) then
            t_setup <= (others => ‘0’);
        else
            if su_flag = ‘1’ and t_setup < (numpixels+im_width*2+3) then
                t_setup <= t_setup + 1;
            end if;
        end if;
    end if;
end process;

process (clk)
variable sync_cnt : integer range 0 to 6 := 0;
begin
    if (clk’event and clk = ‘1’) then
        if (clearn = ‘0’) then
            set_up_flag <= ‘0’;
            activated <= ‘0’;
            for j in 0 to 4 loop -- fifo reset
                row1(j) <= (others => ’0’);
                row2(j) <= (others => ’0’);
                row3(j) <= (others => ’0’);
                row4(j) <= (others => ’0’);
            end loop;
        end if;
    end if;
end process;
row5(j) <= (others => '0');
end loop;
smooth_filter_out <= (others => '0');
data_in1 <= (others => '0');
data_in2 <= (others => '0');
data_in3 <= (others => '0');
data_in4 <= (others => '0');
rd_en1 <= '0';
rd_en2 <= '0';
rd_en3 <= '0';
rd_en4 <= '0';

elsif su_flag = '1' then --Shifts all the registers and fifo's data one position
row1(0) <= unsigned(smooth_filter_in);
row1(1 to 4) <= row1(0 to 3);
data_in1 <= row1(4);
if (unsigned(data_count1) >= fifo_size) then --Maintain a constant amount of data
in the fifo
    rd_en1 <= '1';
    --components depending on the image size
    row2(0) <= unsigned(data_out1);
else
    rd_en1 <= '0';
    row2(0) <= (others=>'0');
end if;
row2(1 to 4) <= row2(0 to 3);
data_in2 <= row2(4);
if (unsigned(data_count2) >= fifo_size) then
    rd_en2 <= '1';
    row3(0) <= unsigned(data_out2);
else
    rd_en2 <= '0';
    row3(0) <= (others=>'0');
end if;
row3(1 to 4) <= row3(0 to 3);
data_in3 <= row3(4);
if (unsigned(data_count3) >= fifo_size) then
    rd_en3 <= '1';
    row4(0) <= unsigned(data_out3);
else
    rd_en3 <= '0';
    row4(0) <= (others=>'0');
end if;
row4(1 to 4) <= row4(0 to 3);
data_in4 <= row4(4);
if (unsigned(data_count4) >= fifo_size) then
    rd_en4 <= '1';
    row5(0) <= unsigned(data_out4);
else
    rd_en4 <= '0';
    row5(0) <= (others=>'0');
end if;
row5(1 to 4) <= row5(0 to 3);
if t_setup >= (im_width*2+3) and t_setup < (numpixels+im_width*2+2) then -- +3 to 1
    set_up_flag <= '1'; --Apply the convolution operation for the current pixel and
its window.
    sync_cnt := 0;
    --smooth_filter_out <= std_logic_vector(resize((("00000000"&row1(1)) +
        ("00000000"&row1(2)) & '0') + ("00000000"&row1(3)) + ("00000000"&row2(0)) +
        ("00000000"&row2(1)) & '0' + ("00000000"&row2(2)) & '0' +
        ("00000000"&row2(3)) & '0' + ("00000000"&row3(0)) & '0' +
        ("00000000"&row3(1)) & '0' + ("00000000"&row3(2)) & '0' +
        ("00000000"&row3(3)) & '0' + ("00000000"&row4(0)) & '0' +
        ("00000000"&row4(1)) & '0' + ("00000000"&row4(2)) & '0' +
        ("00000000"&row4(3)) & '0' + ("00000000"&row5(0)) & '0' +
        ("00000000"&row5(1)) & '0' + ("00000000"&row5(2)) & '0' +
        ("00000000"&row5(3)))/100),8));
smooth_filter_out <= std_logic_vector(resize((("00000000"&row2(1)) + ("00000000"&row2(2)) + ("00000000"&row2(3)) + ("00000000"&row3(1)) + ("00000000"&row3(2)) + ("00000000"&row3(3)) + ("00000000"&row4(1)) + ("00000000"&row4(2)) + ("00000000"&row4(3)))/100),8));
else
    if activated <= '1' then
        -- if sync_cnt < 6 then
            -- sync_cnt := sync_cnt + 1;
        -- else
            -- set_up_flag <= '0'; -- Finish
        -- end if;
        end if;
    end if;
end if;
end process;

wr_en <= su_flag;
fifo_size <= im_width - 8; --Added a -1 not initially forecasted
rst <= not(clearn);
end behavior;
FPGA Implementation of a Contrast Enhancement Algorithm

histogram_int3.vhd

--2013/04/20-- Forked from the description available at:
--http://www.edn.com/design/integrated-circuit-design/4363979/Compute-a-histogram-in-an-FPGA-
with-one-clock
--------------------------------------------------------------------------------------------
--2013/05/01-- The code has been simplified to remove unneeded functionality and make
interfacing easier.
--------------------------------------------------------------------------------------------
--2013/05/03-- Fixed a bug which caused histogram count increasing to not work properly when
--two or more consecutive pixels with the same exact value appeared.
--2013/05/04-- Comments added for clarity and future reference.
--2013/05/07-- More bugfixes, related to component reset.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity histogram is
port ( addrin : in std_logic_vector(7 downto 0) ; -- device data as address for RAM
datain : in std_logic_vector (18 downto 0); -- RAM data out
clk : in std_logic; --Synchronous rising edge clock
cntr_value : in std_logic_vector (18 downto 0); --Number of pixels of the input image
ramwraddr : out std_logic_vector(7 downto 0); --Address where the updated histogram
value must be
--written.Equal to the grey value (from 0 to 255)
rat : in std_logic; --global reset
start_cntr : in std_logic; --triggers the start of the histogram calculation
wren : out std_logic; --write enable output for the ram containing the histogram
data_out : out std_logic_vector(18 downto 0); -- RAM data in
end_flag : out std_logic =
);
end histogram;

architecture hlsm of histogram is

begin
addr <= addrin;

process(clk,rst)
begin
if(clk'event and clk = '1') then
if(rat = '1' or start_cntr='1') then --restart all the procedure
pre_cntr <= (others => '0');
wren_next1 <= '0';
wren_next <= '0';
wren <= '0';
pre_dout <= dout; --counter for use if next pixel's gray value is equal
end_flag_signal <= '0';
else
pre_cntr <= next_cntr;
wren_next1 <= wren_next2;
wren_next <= wren_next1;--delay write enable changes to sync it
wren <= wren_next; --with the output of valid values
wren_signal <= wren_next;
end_flag_signal <= '0';
end if;
if wren_signal='1' and wren_next='0' then
end_flag_signal <= '1';
else
end if;
end if;
end process;
end


end_flag <= end_flag_signal;
end if;
if (addr=pre_addr) then --if the gray value is the same of the
--previous clock, load pre_dout instead of RAM in
addrpreaddr <= '1';
else
addrpreaddr <= '0';
end if;
end if;

wr_addr <= addr;
if (addr=pre_addr) then -- delay write address by 2 clock
wr_addr <= wr_addr1;
else
wr_addr <= wr_addr;
end if;
end process;

process(datain, addrpreaddr, rst, pre_cntr, cntr_value, pre_dout, wr_addr)
begin
if((pre_cntr >= cntr_value)) then --finish if the internal counter reaches the total
next_cntr <= pre_cntr; --amount of pixels in the photo
wren_next2 <= '0';
else
wren_next2 <= '1'; --else: keep calculating and writing.
next_cntr <= pre_cntr + '1';
end if;
if(rst = '1') then
dout <= (others => '0');
else
if(datain = "111111111111111110") then -- prevent overflow
    dout <= datain;
elsif addrpreaddr='1' then --See previous process for addrpreaddr's "if/else"
    dout <= datain + 1; --to avoid not having yet the updated value in the RAM
else
    dout <= datain + '1';
end if;
end if;
ramwraddr <= wr_addr;
data_out <= dout;
end process;
end_flag <= end_flag_signal;
end hls;
---Adds transformation function to complete CLHE functionality.
---2013/05/04--Comments added for clarity and future reference.

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity histogram_wrapper is
  port(
    --global clock signal, active with its rising edge
    clk: in std_logic;
    --reset signal, synchronous and active high
    reset: in std_logic;
    --number of pixels of the image
    cntr_value: in std_logic_vector(18 downto 0);
    --Trigger to start the histogram generation
    pulse_start_input: in std_logic;
    --Output of the histogram data
    histo_dina : out std_logic_vector(17 downto 0);
    --enable : in std_logic;
    histo_douta : in std_logic_vector(7 downto 0);
    histo_addra : out std_logic_vector(7 downto 0);
    histo_wea : out std_logic_vector(0 downto 0);
    histo_addrb : out std_logic_vector(7 downto 0);
    histo_doutb : in std_logic_vector(18 downto 0);
    --histo_rstb : out std_logic;
    end_flag : out std_logic
  );
end histogram_wrapper;

architecture wrapper of histogram_wrapper is

component histogram --computes the histogram
  port ( addrin : in std_logic_vector(7 downto 0) ; -- device data as address for RAM
datain : in std_logic_vector(18 downto 0) ; -- RAM data out
clk : in std_logic;
cntr_value : in std_logic_vector(18 downto 0);
ramwraddr : out std_logic_vector( 7 downto 0);
--rstcntr : in std_logic;
rst : in std_logic;
start_cntr : in std_logic;
wren : out std_logic;
data_out : out std_logic_vector(18 downto 0); -- RAM data in
end_flag : out std_logic
);
end component;
begin
--image_rom, histogram_generator and histogram_ram are interconnected according to the design
--principle proposed in:
--http://www.edn.com/design/integrated-circuit-design/4363979/Compute-a-histogram-in-an-FPGA-
--with-one-clock
--However, some functionality was simplified or removed because it was not needed.

histogram_generator : histogram
  port map(
    addrin => device_data,
    datain => histo_out,
    clk => clk,
    cntr_value => cntr_value,
    ramwraddr => ram_wr_addr,
    rst => reset,
    start_cntr => pulse_out,
    wren => wren(0),
    data_out => dataout,
    end_flag => end_flag_signal
  );

process(clk) --Process containing an address counter to read the image in the
--ROM memory sequentially and compute its histogram
begin
  if (CLK'EVENT AND CLK = '1') then
    if reset = '1' or pulse_out='1' then
      --image_addr <= to_unsigned(0, 18);
    --else
    --image_addr <= image_addr + 1;
    --end if;
    if reset = '1' then
      pulse_out <= '0';
    else
      pulse_out<=pulse_start_input;
    end if;
  end if;
end process;

--histogram_out<=histo_out;
--rstb <= reset or pulse_out;
--im_addra <= std_logic_vector(image_addr);
device_data <= im_douta;

histo_dina <= dataout;
histo_addra <= ram_wr_addr;
histo_wea <= wren;
--histo_rstb <= rstb;
histo_addrb <= device_data;
histo_out <= histo_doutb;
end_flag <= end_flag_signal;
--end_flag <= '0';
end wrapper;
main3.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity main is
    port (  
        clk : in std_logic;
        rst : in std_logic;
        start_cntr : in std_logic; --marks the end of the histogram calculation
        numpixels : in unsigned(18 downto 0);
        x_size : in unsigned(9 downto 0);
        y_size : in unsigned(9 downto 0);
        clip_limit : in unsigned(6 downto 0);
        limit1_t : in unsigned(7 downto 0);
        limit1_b : in unsigned(7 downto 0);
        limit2_t : in unsigned(7 downto 0);
        limit2_b : in unsigned(7 downto 0);
    end
);
end main;

architecture system of main is

signal rom_addrb, binary_addra, clahe_addra, binary_addrb : std_logic_vector(17 downto 0);
signal binary1_dina, binary2_dina, binary_wea, clahe_wea, binary1_doutb, binary2_doutb : std_logic_vector(0 downto 0);
signal rom_doutb, rom_douta, clahe_dina, clahe_doutb : std_logic_vector(7 downto 0);
signal start_filters, end_flag_masks, end_flag_clahe : std_logic;
signal x_size2 : unsigned(10 downto 0);

component clahe_ram_dual  
    port (  
        clka: IN std_logic;
        wea: IN std_logic_VECTOR(0 downto 0);
        addra: IN std_logic_VECTOR(17 downto 0);
        dina: IN std_logic_VECTOR(7 downto 0);
        clkb: IN std_logic;
        addrb: IN std_logic_VECTOR(17 downto 0);
        doutb: OUT std_logic_VECTOR(7 downto 0));
end component;

component prova_grisa_rom3 --contains the source image  
    port (  
        clka: IN std_logic;
        addra: IN std_logic_VECTOR(17 downto 0);
        douta: OUT std_logic_VECTOR(7 downto 0);
        clkb: IN std_logic;
        addrb: IN std_logic_VECTOR(17 downto 0);
        doutb: OUT std_logic_VECTOR(7 downto 0));
end component;

component binary_ram_dual  
    port (  
        clka: IN std_logic;
        wea: IN std_logic_VECTOR(0 downto 0);
        addra: IN std_logic_VECTOR(17 downto 0);
        dina: IN std_logic_VECTOR(0 downto 0);
        clkb: IN std_logic;
        addrb: IN std_logic_VECTOR(17 downto 0);
        doutb: OUT std_logic_VECTOR(0 downto 0));
end component;

component clahe is
    port (  

        clka: IN std_logic;
        wea: IN std_logic_VECTOR(0 downto 0);
        addra: IN std_logic_VECTOR(17 downto 0);
        dina: IN std_logic_VECTOR(0 downto 0);
        clkb: IN std_logic;
        addrb: IN std_logic_VECTOR(17 downto 0);
        doutb: OUT std_logic_VECTOR(0 downto 0));
end component;
FPGA Implementation of a Contrast Enhancement Algorithm

begin
  clahe_generator : clahe

end
FPGA Implementation of a Contrast Enhancement Algorithm

```vhdl
port map (  
    clk => clk,  
rst => rst,  
    start_cntr => start_cntr,  
    end_flag => end_flag_clahe,  
numpixels => numpixels,  
x_size => x_size,  
y_size => y_size,  
clip_limit => clip_limit,  
rom_addra => rom_addra,  
rom_douta => rom_douta,  
ram_wea => clahe_wea,  
ram_addra => clahe_addra,  
ram_dina => clahe_dina  
);  

image_rom : prova_grisa_rom3  
port map (  
    clka => clk,  
    addra => rom_addra,  
    douta => rom_douta,  
    clkb => clk,  
    addrb => rom_addrb,  
    doutb => rom_doutb);  

bram1 : binary_ram_dual  
port map (  
    clka => clk,  
    wea => binary_wea,  
    addra => binary_addra,  
    dina => binary1_dina,  
    clkb => clk,  
    addrb => binary_addrb,  
    doutb => binary1_doutb);  

bram2 : binary_ram_dual  
port map (  
    clka => clk,  
    wea => binary_wea,  
    addra => binary_addra,  
    dina => binary2_dina,  
    clkb => clk,  
    addrb => binary_addrb,  
    doutb => binary2_doutb);  

binarizer : mask_generator  
port map(  
    clk => clk,  
    reset => rst,  
    pulse_start_input => start_cntr,  
    limit1_t => limit1_t,  
    limit1_b => limit1_b,  
    limit2_t => limit2_t,  
    limit2_b => limit2_b,  
    end_flag => end_flag_masks,  
    rom_addrb => rom_addrb,  
    rom_doubt => rom_doubt,  
    im_width => x_size,  
    numpixels => numpixels,  
    binary_wea => binary_wea,  
    binary_addra => binary_addra,  
    binary1_dina => binary1_dina,  
    binary2_dina => binary2_dina  
);  

output_clahe : clahe_ram_dual  
port map (  
    clka => clk,  
    wea => clahe_wea,  
    addra => clahe_addra,  
    dina => clahe_dina,  
    clkb => clk,  
    addrb => binary_addrb,  
    doutb => clahe_doutb);  
```
filters : filter_testbench
  port map(
    --global clock signal, active with its rising edge
    clk => clk,
    --reset signal, synchronous and active high
    reset => rst,
    --Trigger to start the generation
    pulse_start_input => start_filters,
    --Output of the data
    dataout => open,
    memout => open,
    end_flag => end_flag,
    --addrb : in std_logic_vector(17 downto 0);
    --doutb : out std_logic_vector(7 downto 0);
    --ram_wea : out std_logic_vector(0 downto 0);
    --ram_addra : out std_logic_vector(17 downto 0);
    --ram_dina : out std_logic_vector(7 downto 0);
    b1_doutb => binary1_doutb,
    b2_doutb => binary2_doutb,
    input_ram_addr => binary_addrb,
    clahe_ram_doutb => clahe_doutb
  );

start_filters <= end_flag_clahe and end_flag_masks;

x_size2 <= resize(x_size,11);

end system;
median_filter2.vhd

-----------------------------------------------
-- Contrast enhancement algorithm with noise removal
-- 2nd level of hierarchy - smooth_filter
-- Original: Núria Orduña
-- Modified by: Roger Olivé
-- Concordia University
-- 2012-2013
-----------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity median_filter is
  port ( clk, clearn : in std_logic;
  su_flag : in std_logic;
  smooth_filter_in : in std_logic_vector(7 downto 0);
  smooth_filter_out : out std_logic_vector (7 downto 0);
  set_up_flag : out std_logic;
  im_width : in unsigned(10 downto 0);
  numpixels : in unsigned(18 downto 0)
  );
end median_filter;

architecture behavior of median_filter is

component filter_fifo
  port ( clk: IN std_logic;
  rst: IN std_logic;
  din: IN std_logic_VECTOR(7 downto 0);
  wr_en: IN std_logic;
  rd_en: IN std_logic;
  dout: OUT std_logic_VECTOR(7 downto 0);
  full: OUT std_logic;
  empty: OUT std_logic;
  data_count: OUT std_logic_VECTOR(8 downto 0));
end component;

-- Signal Declarations
-----------------------------------------------
type data_win is array (0 to 4) of unsigned (7 downto 0);
type median_vector is array (0 to 7) of unsigned (7 downto 0);
type final_median_vector is array (0 to 2) of unsigned (7 downto 0);
type center_pixel_wait is array (0 to 5) of unsigned (7 downto 0);
signal wc : center_pixel_wait;
signal wf0, wf1, wf2 : final_median_vector;
signal wd0, wd1, wd2, wd3, wd4, wd5, wd6, wo0, wo1, wo2, wo3, wo4, wo5, wo6 : median_vector;
signal wd, wo : unsigned(7 downto 0);
signal row1 : data_win;
signal row2 : data_win;
signal row3 : data_win;
signal row4 : data_win;
signal row5 : data_win;
signal data_in1 : unsigned (7 downto 0);
signal data_out1 : std_logic_vector (7 downto 0);
signal data_in2 : unsigned (7 downto 0);
signal data_out2 : std_logic_vector (7 downto 0);
signal data_in3 : unsigned (7 downto 0);
signal data_out3 : std_logic_vector (7 downto 0);
signal data_in4 : unsigned (7 downto 0);
signal data_out4 : std_logic_vector (7 downto 0);
signal t_setup : unsigned (18 downto 0);
signal activated : std_logic;
FPGA Implementation of a Contrast Enhancement Algorithm

signal data_count1, data_count2, data_count3, data_count4 : std_logic_vector(8 downto 0);
signal fifo_size : unsigned(10 downto 0);
signal wr_en, rd_en1, rd_en2, rd_en3, rd_en4, rst : std_logic;

begin
fifo1 : filter_fifo
port map (
  clk => clk,
  rst => rst,
  din => std_logic_vector(data_in1),
  wr_en => wr_en,
  rd_en => rd_en1,
  dout => data_out1,
  full => open,
  empty => open,
  data_count => data_count1);

fifo2 : filter_fifo
port map (
  clk => clk,
  rst => rst,
  din => std_logic_vector(data_in2),
  wr_en => wr_en,
  rd_en => rd_en2,
  dout => data_out2,
  full => open,
  empty => open,
  data_count => data_count2);

fifo3 : filter_fifo
port map (
  clk => clk,
  rst => rst,
  din => std_logic_vector(data_in3),
  wr_en => wr_en,
  rd_en => rd_en3,
  dout => data_out3,
  full => open,
  empty => open,
  data_count => data_count3);

fifo4 : filter_fifo
port map (
  clk => clk,
  rst => rst,
  din => std_logic_vector(data_in4),
  wr_en => wr_en,
  rd_en => rd_en4,
  dout => data_out4,
  full => open,
  empty => open,
  data_count => data_count4);

-- Module Implementation

process (clk)
begin
  if (clk'event and clk = '1') then
    if (clearn = '0') then
      t_setup <= (others => '0');
    else
      if su_flag = '1' and t_setup < (numpixels+im_width*2+3+9) then
        t_setup <= t_setup + 1;
      end if;
    end if;
  end if;
end process;

process (clk)
variable sync_cnt : integer range 0 to 6 := 0;
begin
FPGA Implementation of a Contrast Enhancement Algorithm

if \( \text{clk}'\text{event and clk = '1'} \) then --initialization
  if \( \text{clearn = '0'} \) then
    set_up_flag <= '0';
    activated <= '0';
    for \( j \) in 0 to 4 loop -- fifo reset
      row1(j) <= (others => '0');
      row2(j) <= (others => '0');
      row3(j) <= (others => '0');
      row4(j) <= (others => '0');
      row5(j) <= (others => '0');
    end loop;
    --wd <= (others => '0');
    --wo <= (others => '0');
    for \( m \) in 0 to 7 loop
      wd6(m) <= (others => '0');
      wd1(m) <= (others => '0');
      wd2(m) <= (others => '0');
      wd3(m) <= (others => '0');
      wd4(m) <= (others => '0');
      wd5(m) <= (others => '0');
      wo6(m) <= (others => '0');
      wo1(m) <= (others => '0');
      wo2(m) <= (others => '0');
      wo3(m) <= (others => '0');
      wo4(m) <= (others => '0');
      wo5(m) <= (others => '0');
    end loop;
    for \( n \) in 0 to 2 loop
      wf0(n) <= (others => '0');
      wf1(n) <= (others => '0');
      wf2(n) <= (others => '0');
    end loop;
    for \( o \) in 0 to 5 loop
      wc(o) <= (others => '0');
    end loop;
    smooth_filter_out <= (others => '0');
    data_in1 <= (others => '0');
    data_in2 <= (others => '0');
    data_in3 <= (others => '0');
    data_in4 <= (others => '0');
    rd_en1 <= '0';
    rd_en2 <= '0';
    rd_en3 <= '0';
    rd_en4 <= '0';
  elsif su_flag = '1' then --Shifts all the registers and fifo's data one position
    row1(0) <= unsigned(smooth_filter_in);
    row1(1 to 4) <= row1(0 to 3);
    data_in1 <= row1(4);
    if (unsigned(data_count1) >= fifo_size) then --Maintain a constant amount of data in the fifo
      rd_en1 <= '1'; --components depending on the image size
      row2(0) <= unsigned(data_out1);
    else
      rd_en1 <= '0';
      row2(0) <= (others=>'0');
    end if;
    row2(1 to 4) <= row2(0 to 3);
    data_in2 <= row2(4);
    if (unsigned(data_count2) >= fifo_size) then
      rd_en2 <= '1';
      row3(0) <= unsigned(data_out2);
    else
      rd_en2 <= '0';
      row3(0) <= (others=>'0');
    end if;
    row3(1 to 4) <= row3(0 to 3);
    data_in3 <= row3(4);
    if (unsigned(data_count3) >= fifo_size) then
      rd_en3 <= '1';
      row4(0) <= unsigned(data_out3);
    else
      rd_en3 <= '0';
      row4(0) <= (others=>'0');
    end if;
  end if;
else
  if \( \text{su_flag = '1'} \) then --Shifts all the registers and fifo's data one position
    row1(0) <= unsigned(smooth_filter_in);
    row1(1 to 4) <= row1(0 to 3);
    data_in1 <= row1(4);
    if (unsigned(data_count1) >= fifo_size) then --Maintain a constant amount of data in the fifo
      rd_en1 <= '1'; --components depending on the image size
      row2(0) <= unsigned(data_out1);
    else
      rd_en1 <= '0';
      row2(0) <= (others=>'0');
    end if;
    row2(1 to 4) <= row2(0 to 3);
    data_in2 <= row2(4);
    if (unsigned(data_count2) >= fifo_size) then
      rd_en2 <= '1';
      row3(0) <= unsigned(data_out2);
    else
      rd_en2 <= '0';
      row3(0) <= (others=>'0');
    end if;
    row3(1 to 4) <= row3(0 to 3);
    data_in3 <= row3(4);
    if (unsigned(data_count3) >= fifo_size) then
      rd_en3 <= '1';
      row4(0) <= unsigned(data_out3);
    else
      rd_en3 <= '0';
      row4(0) <= (others=>'0');
    end if;
  end if;
end if;
FPGA Implementation of a Contrast Enhancement Algorithm

--Stage 1

rd_en3 <= '0';
row4(0) <= (others=>'0');
end if;
row4(1 to 4) <= row4(0 to 3);
data_in4 <= row4(4);
if (unsigned(data_count4) >= fifo_size) then
  rd_en4 <= '1';
  row5(0) <= unsigned(data_out4);
else
  rd_en4 <= '0';
  row5(0) <= (others=>'0');
end if;
row5(1 to 4) <= row5(0 to 3);
if t_setup >= (im_width*2+3+9) and t_setup < (numpixels+im_width*2+2+9) then --

set_up_flag <= '1';
activated <= '1'; --Apply the convolution operation for the current pixel and
its window.
sync_cnt := 0;
smooth_filter_out <= std_logic_vector(wf2(1));
else
  if activated <= '1' then
    --if sync_cnt < 6 then
    -- sync_cnt := sync_cnt + 1;
    --else
    --set_up_flag <= '0'; --Finish
    --end if;
  end if;
end if;

wc(0) <= row3(2);
for o in 0 to 4 loop
  wc(o+1) <= wc(o);
end loop;

--Calculation of the wd kernel median
if (wd0(0) >= wd0(7)) then
  wd1(0) <= wd0(7);
  wd1(7) <= wd0(0);
else
  wd1(0) <= wd0(0);
  wd1(7) <= wd0(7);
end if;
if (wd0(1) >= wd0(6)) then
  wd1(1) <= wd0(6);
  wd1(6) <= wd0(1);
else
  wd1(1) <= wd0(1);
  wd1(6) <= wd0(6);
end if;
if (wd0(2) >= wd0(5)) then
  wd1(2) <= wd0(5);
  wd1(5) <= wd0(2);
else
  wd1(2) <= wd0(2);
  wd1(5) <= wd0(5);
end if;
if (wd0(3) >= wd0(4)) then
  wd1(3) <= wd0(4);
  wd1(4) <= wd0(3);
else
  wd1(3) <= wd0(3);
  wd1(4) <= wd0(4);
end if;

--Stage 2
if (wd1(0) >= wd1(3)) then
  wd2(0) <= wd1(3);
  wd2(3) <= wd1(0);
else
  wd2(0) <= wd1(0);
  wd2(3) <= wd1(3);
end if;

if (wd1(4) >= wd1(7)) then
  wd2(4) <= wd1(7);
  wd2(7) <= wd1(4);
else
  wd2(4) <= wd1(4);
  wd2(7) <= wd1(7);
end if;

if (wd1(1) >= wd1(2)) then
  wd2(1) <= wd1(2);
  wd2(2) <= wd1(1);
else
  wd2(1) <= wd1(1);
  wd2(2) <= wd1(2);
end if;

if (wd1(5) >= wd1(6)) then
  wd2(5) <= wd1(6);
  wd2(6) <= wd1(5);
else
  wd2(5) <= wd1(5);
  wd2(6) <= wd1(6);
end if;

--Stage 3

if (wd2(0) >= wd2(1)) then
  wd3(0) <= wd2(1);
  wd3(1) <= wd2(0);
else
  wd3(0) <= wd2(0);
  wd3(1) <= wd2(1);
end if;

if (wd2(2) >= wd2(3)) then
  wd3(2) <= wd2(3);
  wd3(3) <= wd2(2);
else
  wd3(2) <= wd2(2);
  wd3(3) <= wd2(3);
end if;

if (wd2(4) >= wd2(5)) then
  wd3(4) <= wd2(5);
  wd3(5) <= wd2(4);
else
  wd3(4) <= wd2(4);
  wd3(5) <= wd2(5);
end if;

if (wd2(6) >= wd2(7)) then
  wd3(6) <= wd2(7);
  wd3(7) <= wd2(6);
else
  wd3(6) <= wd2(6);
  wd3(7) <= wd2(7);
end if;

--Stage 4

wd4(0) <= wd3(0);
wd4(1) <= wd3(1);
wd4(6) <= wd3(6);
wd4(7) <= wd3(7);

if (wd3(2) >= wd3(4)) then
  wd4(2) <= wd3(4);
  wd4(4) <= wd3(2);

end if;
else
    wd4(2) <= wd3(2);
    wd4(4) <= wd3(4);
end if;

if (wd3(3) >= wd3(5)) then
    wd4(3) <= wd3(5);
    wd4(5) <= wd3(3);
else
    wd4(3) <= wd3(3);
    wd4(5) <= wd3(5);
end if;

--Stage 5
wd5(0) <= wd4(0);
wd5(7) <= wd4(7);

if (wd4(1) >= wd4(2)) then
    wd5(1) <= wd4(2);
    wd5(2) <= wd4(1);
else
    wd5(1) <= wd4(1);
    wd5(2) <= wd4(2);
end if;

if (wd4(3) >= wd4(4)) then
    wd5(3) <= wd4(4);
    wd5(4) <= wd4(3);
else
    wd5(3) <= wd4(3);
    wd5(4) <= wd4(4);
end if;

if (wd4(5) >= wd4(6)) then
    wd5(5) <= wd4(6);
    wd5(6) <= wd4(5);
else
    wd5(5) <= wd4(5);
    wd5(6) <= wd4(6);
end if;

--Stage 6
wd6(0) <= wd5(0);
wd6(1) <= wd5(1);
wd6(5) <= wd5(6);
wd6(7) <= wd5(7);

if (wd5(2) >= wd5(3)) then
    wd6(2) <= wd5(3);
    wd6(3) <= wd5(2);
else
    wd6(2) <= wd5(2);
    wd6(3) <= wd5(3);
end if;

if (wd5(4) >= wd5(5)) then
    wd6(4) <= wd5(5);
    wd6(5) <= wd5(4);
else
    wd6(4) <= wd5(4);
    wd6(5) <= wd5(5);
end if;

--Calculation of the wo kernel median
if (wo0(0) >= wo0(7)) then
    wo1(0) <= wo0(7);
    wo1(7) <= wo0(0);
else
    wo1(0) <= wo0(0);
    wo1(7) <= wo0(7);
end if;
if (wo0(1) >= wo0(6)) then
    wo1(1) <= wo0(6);
    wo1(6) <= wo0(1);
else
    wo1(1) <= wo0(1);
    wo1(6) <= wo0(6);
end if;

if (wo0(2) >= wo0(5)) then
    wo1(2) <= wo0(5);
    wo1(5) <= wo0(2);
else
    wo1(2) <= wo0(2);
    wo1(5) <= wo0(5);
end if;

if (wo0(3) >= wo0(4)) then
    wo1(3) <= wo0(4);
    wo1(4) <= wo0(3);
else
    wo1(3) <= wo0(3);
    wo1(4) <= wo0(4);
end if;

--Stage 2

if (wo1(0) >= wo1(3)) then
    wo2(0) <= wo1(3);
    wo2(3) <= wo1(0);
else
    wo2(0) <= wo1(0);
    wo2(3) <= wo1(3);
end if;

if (wo1(4) >= wo1(7)) then
    wo2(4) <= wo1(7);
    wo2(7) <= wo1(4);
else
    wo2(4) <= wo1(4);
    wo2(7) <= wo1(7);
end if;

if (wo1(1) >= wo1(2)) then
    wo2(1) <= wo1(2);
    wo2(2) <= wo1(1);
else
    wo2(1) <= wo1(1);
    wo2(2) <= wo1(2);
end if;

if (wo1(5) >= wo1(6)) then
    wo2(5) <= wo1(6);
    wo2(6) <= wo1(5);
else
    wo2(5) <= wo1(5);
    wo2(6) <= wo1(6);
end if;

--Stage 3

if (wo2(0) >= wo2(1)) then
    wo3(0) <= wo2(1);
    wo3(1) <= wo2(0);
else
    wo3(0) <= wo2(0);
    wo3(1) <= wo2(1);
end if;

if (wo2(2) >= wo2(3)) then
    wo3(2) <= wo2(3);
    wo3(3) <= wo2(2);
else
    wo3(2) <= wo2(2);
    wo3(3) <= wo2(3);
end if;
if (wo2(4) >= wo2(5)) then
  wo3(4) <= wo2(5);
  wo3(5) <= wo2(4);
else
  wo3(4) <= wo2(4);
  wo3(5) <= wo2(5);
end if;

if (wo2(6) >= wo2(7)) then
  wo3(6) <= wo2(7);
  wo3(7) <= wo2(6);
else
  wo3(6) <= wo2(6);
  wo3(7) <= wo2(7);
end if;

-- Stage 4
wo4(0) <= wo3(0);
wo4(1) <= wo3(1);
wo4(6) <= wo3(6);
wo4(7) <= wo3(7);

if (wo3(2) >= wo3(4)) then
  wo4(2) <= wo3(4);
  wo4(4) <= wo3(2);
else
  wo4(2) <= wo3(2);
  wo4(4) <= wo3(4);
end if;

if (wo3(3) >= wo3(5)) then
  wo4(3) <= wo3(5);
  wo4(5) <= wo3(3);
else
  wo4(3) <= wo3(3);
  wo4(5) <= wo3(5);
end if;

-- Stage 5
wo5(0) <= wo4(0);
wo5(7) <= wo4(7);

if (wo4(1) >= wo4(2)) then
  wo5(1) <= wo4(2);
  wo5(2) <= wo4(1);
else
  wo5(1) <= wo4(1);
  wo5(2) <= wo4(2);
end if;

if (wo4(3) >= wo4(4)) then
  wo5(3) <= wo4(4);
  wo5(4) <= wo4(3);
else
  wo5(3) <= wo4(3);
  wo5(4) <= wo4(4);
end if;

if (wo4(5) >= wo4(6)) then
  wo5(5) <= wo4(6);
  wo5(6) <= wo4(5);
else
  wo5(5) <= wo4(5);
  wo5(6) <= wo4(6);
end if;

-- Stage 6
wo6(0) <= wo5(0);
wo6(1) <= wo5(1);
wo6(6) <= wo5(6);
wo6(7) <= wo5(7);
FPGA Implementation of a Contrast Enhancement Algorithm

```plaintext
if (wo5(2) >= wo5(3)) then
    wo6(2) <= wo5(3);
    wo6(3) <= wo5(2);
else
    wo6(2) <= wo5(2);
    wo6(3) <= wo5(3);
end if;

if (wo5(4) >= wo5(5)) then
    wo6(4) <= wo5(5);
    wo6(5) <= wo5(4);
else
    wo6(4) <= wo5(4);
    wo6(5) <= wo5(5);
end if;

--Final median calculation
wf0(0) <= wc(5);
if (wo >= wd) then
    wf0(1) <= wd;
    wf0(2) <= wo;
else
    wf0(1) <= wo;
    wf0(2) <= wd;
end if;

wf1(2) <= wf0(2);
if (wf0(0) >= wf0(1)) then
    wf1(0) <= wf0(1);
    wf1(1) <= wf0(0);
else
    wf1(0) <= wf0(0);
    wf1(1) <= wf0(1);
end if;

if (wf1(1) >= wf1(2)) then
    wf2(2) <= wf1(1);
    wf2(1) <= wf1(2);
else
    wf2(1) <= wf1(1);
    wf2(2) <= wf1(2);
end if;

wf2(0) <= wf1(0);
end if;
end if;
end process;

wr_en <= su_flag;
fifo_size <= im_width - 8;  --Afegit un -1 no previst inicialment
rst <= not(clearn);

wd0(0) <= row1(0);
wd0(1) <= row1(4);
wd0(2) <= row2(1);
wd0(3) <= row2(3);
wd0(4) <= row4(1);
wd0(5) <= row4(3);
wd0(6) <= row5(0);
wd0(7) <= row5(4);

wo0(0) <= row1(2);
wo0(1) <= row2(2);
wo0(2) <= row3(0);
wo0(3) <= row3(1);
wo0(4) <= row3(3);
wo0(5) <= row3(4);
wo0(6) <= row4(2);
wo0(7) <= row5(2);
```

\texttt{\textbf{wo} <= \texttt{resize((resize(wo6(3),9)+wo6(4))/2),8)};}

\texttt{\textbf{wd} <= \texttt{resize((resize(wd6(3),9)+wd6(4))/2),8)};}

\texttt{end behavior;}
tiling_int3.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity image_tiling is
  port (  romraddr : out std_logic_vector(17 downto 0) ; -- device data as address for RAM
datain : in std_logic_vector (7 downto 0); -- RAM data out
clk : in std_logic;
ramwraddr : out std_logic_vector(17 downto 0);
dataout : out std_logic_vector(7 downto 0); -- RAM data in
numx : out unsigned(2 downto 0);
numy : out unsigned(2 downto 0);
rst : in std_logic;
start_cntr : in std_logic;
active_flag : out std_logic;
end_flag : out std_logic; --marks the end of the distribution
numpixels : in unsigned(18 downto 0); --number of pixels of the histogram
tile_num_pixels : out unsigned(18 downto 0);
x_size : in unsigned(9 downto 0);
y_size : in unsigned(9 downto 0);
xtile : out unsigned(9 downto 0);
ytile : out unsigned(9 downto 0)
);
end image_tiling;

architecture tiler of image_tiling is
  --Llista de senyals
  signal x_pos, y_pos, x_pos_calc, y_pos_calc : unsigned(7 downto 0);
signal x_coord : unsigned(11 downto 0);
signal y_coord : unsigned(10 downto 0);
signal x_tile, x_tile1, y_tile : unsigned(7 downto 0);
signal x_tile2, x_tile3, x_tile4 : unsigned(11 downto 0);
signal start, pre_start, active : std_logic;
signal over_num : std_logic_vector(1 downto 0);
signal num_x : unsigned(3 downto 0);
signal num_y : unsigned(2 downto 0);
signal pixel_count, pixel_count2, pixel_count3 : unsigned(18 downto 0);
signal romraddr_signal : unsigned(17 downto 0);
signal wren1, wren2, finish, stop : std_logic;
signal x_condition, y_condition, y_condition1, y_condition2 : std_logic;
signal numpixels_tile : unsigned(18 downto 0);

  --Llista de components?
begin
--Codi codi codi (processos i no processos)
  --Process to read all the histogram and calculate the amount of absolute clipping
process(clk)
begin
  if (CLK'EVENT AND CLK = '1') then --calculate tile size
    if rst = '1' then
      x_tile <= (others=>'0');
y_tile <= (others=>'0');
start <= '0';
pre_start <= '0';
    else
      x_tile <= resize(shift_right((x_size),3)+1,8);
      y_tile <= resize(shift_right((y_size),3)+1,8);
start <= start_cntr and not(finish);
pre_start <= start;
end if;
  end if;
    if (CLK'EVENT AND CLK = '1') then
end process;
if rst = '1' then
  active <= '0';
  num_x <= (others => '0');
  num_y <= (others => '0');
elsif start='1' and pre_start = '0' then
  active <= '1';
  if num_x = 8 then
    num_x <= to_unsigned(1,4);
    if num_y = 7 then
      num_y <= to_unsigned(7,3);
    else
      num_y <= num_y+1;
    end if;
  else
    num_x <= num_x+1;
  end if;
elsif stop='1' then
  active <= '0';
  num_x <= num_x;
  num_y <= num_y;
else
  active <= active;
  num_x <= num_x;
  num_y <= num_y;
end if;
end if;

if (CLK'EVENT AND CLK = '1') then --romraddr section
  if rst = '1' then
    romraddr_signal <= (others=>'0');
    dataout <= (others=>'0');
    ramwraddr <= (others=>'0');
    wren <= '0';
    numpixels_tile <= (others=>'0');
    --pre_active <= '0';
  else
    --romraddr <= x_size*(y_pos wren+y_tile*num_y) + x_pos + num_x*x_tile;
    romraddr_signal <= (resize((x_size*y_coord + x_coord),18));
    dataout <= datain;
    ramwraddr <= std_logic_vector(pixel_count3(17 downto 0));
    wren <= wren1;
    if active='1' then
      numpixels_tile <= pixel_count;
    else
      numpixels_tile <= numpixels_tile;
    end if;
    --pre_active <= active;
  end if;
end if;

if (CLK'EVENT AND CLK = '1') then --x_pos and y_pos section
  if rst = '1' or active = '0' then
    pixel_count <= (others => '0');
    pixel_count2 <= (others => '0');
    --pixel_count3 <= (others => '0');
    x_pos <= (others => '0');
    y_pos <= (others => '0');
    finish <= '0';
    --wren1 <= wren2; --S'hauria d'arreglar per evitar glitxos. Que amb el reset
    wren2 <= '0';
    if (x_condition='1') then
      x_pos <= (others=>'0');
    else
      x_pos <= x_pos_calc;
    end if;
  else
    pixel_count <= pixel_count + 1;
    pixel_count2 <= pixel_count;
    --wren1 <= wren2;
    wren2 <= '1';
    if (x_condition='1') then
      x_pos <= (others=>'0');
    else
      x_pos <= x_pos_calc;
    end if;
  end if;
end if;
end if;

--if (y_condition1 = '1') then
  --finish <= '1';
  --else
  --finish <= '0';
  --end if;
else
  --finish <= '0';
  --stop <= '0';
  x_pos <= x_pos_calc;
  y_pos <= y_pos;
end if;
end if;
if rst='1' then
  finish<='0';
else
  if romraddr_signal>= (numpixels-1) then
    finish<='1';
  else
    finish<=finish;
  end if;
end if;
if rst = '1' then
  wren1 <= '0'; --S'hauria d'arreglar per evitar glitzos. Que amb el reset si
  que es posi a zero.
else
  wren1 <= wren2;
end if;

pixel_count3 <= pixel_count2;
end if;
end process;
x_pos_calc <= x_pos + 1;
y_pos_calc <= y_pos + 1;
x Coord <= x_pos + (num_x-1)*x_tile;
y Coord <= y_pos + num_y*y_tile;
x-condition <= '1' when (x_pos_calc >= x_tile or x_coord >= (x_size-1))
else '0';
y-condition1 <= '1' when (y_coord >= (y_size-1))
else '0';
y-condition2 <= '1' when (y_pos_calc >= y_tile)
else '0';
stop <= '1' when (x_condition = '1' and (y_condition1 = '1' or y_condition2='1'))
else '0';
active_flag <= active;
numx <= resize((num_x-1),3);
numy <= num_y;
tile_numpixels <= numpixels_tile;
romraddr <= std_logic_vector(romraddr_signal);
end_flag <= finish;

xtile <= resize(x_tile,10);
ytile <= resize(y_tile,10);

--debug

--x_tile1 <= resize(x_tile2,8);
--x_tile2 <= shift_right(x_tile3,2);
--x_tile3 <= shift_left((x_tile4+4),2)/8;
--x_tile4 <= resize(x_size,12);
end tiler;
entity histogram_equalizer is
   port (  histo_raddr : out std_logic_vector(7 downto 0); -- device data as address for RAM
           histo_in_ul : in std_logic_vector (18 downto 0); -- histogram CDF value
           histo_in_ur : in std_logic_vector (18 downto 0); -- histogram CDF value
           histo_in_ll : in std_logic_vector (18 downto 0); -- histogram CDF value
           histo_in_lr : in std_logic_vector (18 downto 0); -- histogram CDF value
           rom_raddr : out std_logic_vector(17 downto 0); --image pixel address
           rom_in : std_logic_vector(7 downto 0);--image pixel value
           clk : in std_logic;
           clhe_wraddr : out std_logic_vector( 17 downto 0); --Address for the transformed pixel to write
           rst : in std_logic;
           start_cntr : in std_logic; --Triggers the transformation operations
           wren : out std_logic;
           clhe_out : out std_logic_vector(7 downto 0); -- Output for transformed pixel value
           end_flag : out std_logic; --marks the end of the histogram calculation
           numpixels : in unsigned(18 downto 0);
           numpixels_ul : in unsigned(18 downto 0); --number of pixels of the image
           histo_min_ul : in unsigned(18 downto 0); --Lowest CDF value of the histogram
           numpixels_ur : in unsigned(18 downto 0); --number of pixels of the image
           histo_min_ur : in unsigned(18 downto 0); --Lowest CDF value of the histogram
           numpixels_ll : in unsigned(18 downto 0); --number of pixels of the image
           histo_min_ll : in unsigned(18 downto 0) --Lowest CDF value of the histogram
           numpixels_lr : in unsigned(18 downto 0); --number of pixels of the image
           histo_min_lr : in unsigned(18 downto 0) --Lowest CDF value of the histogram
          );
end histogram_equalizer;

architecture transformation of histogram_equalizer is

begin
  process(clk)
  begin
    if (CLK'EVENT AND CLK = '1') then
      if rst = '1' then
        start_add <= '0';
      else
        start_add <= start;
      end if;
      --------------------------------beginning of reading block--------------------------------------
      if rst = '1' or start='1' then --Initialization of variables

      signal ramwraddr1, ramwraddr2, ramwraddr3, ramwraddr4, im_raddr : unsigned(17 downto 0);
      signal pixel, pixel_pre, transformed_u1, transformed_u2, transformed_l1, transformed_l2,
      ul_id_pre : unsigned(7 downto 0);
      signal start_add, start, pre_wren, pre_wren2, pre_wren3, pre_wren4, wren1 : std_logic;
      signal x_pos, x_pos_pre1, x_pos_pre2, x_pos_pre3, x_pos_pre4, x_pos_pre5, y_pos, y_pos_pre1,
      y_pos_pre2, y_pos_pre3, y_pos_pre4, y_pos_pre5 : signed(13 downto 0);
      signal x_ref, x1, x1_pre1, x1_pre2, x2, x2_pre1, x2_pre2, y_ref, y1, y1_pre1, y1_pre2, y2,
      y2_pre1, y2_pre2 : signed(13 downto 0);
      signal num_x, num_x1, num_x2, num_y, num_y1, num_y2 : unsigned(3 downto 0);
      signal numpixels2 : unsigned(18 downto 0);
      signal numpixels_ul1, numpixels_ul2, histo_min_ul1, histo_min_ul2, numpixels_ur1,
      numpixels_ur2, histo_min_ur1, histo_min_ur2, numpixels_ll1, numpixels_ll2, histo_min_ll1,
      histo_min_ll2, numpixels_lr1, numpixels_lr2, histo_min_lr1, histo_min_lr2 : unsigned(18
downto 0));
      begin
        process(clk)
        begin
          if (CLK'EVENT AND CLK = '1') then
            if rst = '1' then
              start_add <= '0';
            else
              start_add <= start;
            end if;
            begin
              if rst = '1' or start='1' then --Initialization of variables
              end if;
            end begin
          end if;
        end begin
      end process(clk)
    end if;
  end process(clk)
end architecture;
FPGA Implementation of a Contrast Enhancement Algorithm

```vhdl
im_raddr <= to_unsigned(0, 18);
clhe_wraddr <= (others => '0');
ramwraddr1 <= to_unsigned(0, 18);
ramwraddr2 <= to_unsigned(0, 18);
ramwraddr3 <= to_unsigned(0, 18);
ramwraddr4 <= to_unsigned(0, 18);
pixel <= to_unsigned(0, 8);
pixel_pre <= to_unsigned(0, 8);
pre_wren <= '0';
pre_wren2 <= '0';
pre_wren3 <= '0';
pixel <= unsigned(rom_in);
pixel_pre <= pixel;
clhe_wraddr <= std_logic_vector(ramwraddr1);
pre_wren4 <= '0';
pre_wren3 <= pre_wren4;
pre_wren2 <= pre_wren3;
pre_wren <= pre_wren2;
x_pos <= x_pos;
y_pos <= y_pos;
num_x <= num_x;
num_y <= num_y;
enflag <= '1';
elsif unsigned(im_raddr) >= (numpixels2) then -- Don't let it keep writing if the whole memory has been swept.
    im_raddr <= numpixels2(17 downto 0);
    ramwraddr4 <= im_raddr;
    ramwraddr3 <= ramwraddr4;
    ramwraddr2 <= ramwraddr3;
    --align the write addresses and write enable with the output data
    ramwraddr1 <= ramwraddr2;
    clhe_wraddr <= std_logic_vector(ramwraddr1);
    pixel <= unsigned(rom_in);
pixel_pre <= pixel;
    clhe_wraddr <= std_logic_vector(ramwraddr1);
pre_wren <= pre_wren2;
if unsigned(x_pos) < (im_width-1) then
    x_pos <= x_pos + 1;
y_pos <= y_pos;
else
    x_pos <= (others=>'0');
y_pos <= y_pos + 1;
end if;
if unsigned(y_pos_pre1) >=
    (shift_right(shift_left(((resize(y_size,11)/2)+1),1),1) + y_size*num_y-1) then
    num_y <= num_y + 1;
else
    num_y <= num_y;
end if;
if unsigned(x_pos_pre2) >= (im_width-1) then
    num_x <= (others=>'0');
else
    num_x <= num_x + 1;
end if;
```
else
    num_x <= num_x;
end if;
end_flag <= '0';
end if;

if rst='1' then
    y_pos_pre1 <= (others=>'0');
    y_pos_pre2 <= (others=>'0');
    y_pos_pre3 <= (others=>'0');
    y_pos_pre4 <= (others=>'0');
    y_pos_pre5 <= (others=>'0');
    x_pos_pre2 <= (others=>'0');
    x_pos_pre3 <= (others=>'0');
    x_pos_pre4 <= (others=>'0');
    x_pos_pre5 <= (others=>'0');
    
x1_pre1 <= (others=>'0');
    x1_pre2 <= (others=>'0');
    x2_pre1 <= (others=>'0');
    x2_pre2 <= (others=>'0');
    y1_pre1 <= (others=>'0');
    y1_pre2 <= (others=>'0');
    y2_pre1 <= (others=>'0');
    y2_pre2 <= (others=>'0');
    num_x1<= (others=>'0');
    num_x2<= (others=>'0');
    num_y1<= (others=>'0');
    num_y2<= (others=>'0');
    transformed_ul <= (others=>'0');
    transformed_ur <= (others=>'0');
    transformed_ll <= (others=>'0');
    transformed_lr <= (others=>'0');
else
    y_pos_pre1 <= y_pos;
    y_pos_pre2 <= y_pos_pre1;
    y_pos_pre3 <= y_pos_pre2;
    y_pos_pre4 <= y_pos_pre3;
    y_pos_pre5 <= y_pos_pre4;
    x_pos_pre1 <= x_pos;
    x_pos_pre2 <= x_pos_pre1;
    x_pos_pre3 <= x_pos_pre2;
    x_pos_pre4 <= x_pos_pre3;
    x_pos_pre5 <= x_pos_pre4;
    
x1_pre1 <= x1;
    x1_pre2 <= x1_pre1;
    x2_pre1 <= x2;
    x2_pre2 <= x2_pre1;
    y1_pre1 <= y1;
    y1_pre2 <= y1_pre1;
    y2_pre1 <= y2;
    y2_pre2 <= y2_pre1;
    num_x1<= num_x;
    num_x2<= num_x1;
    num_y1<= num_y;
    num_y2<= num_y1;
    transformed_ul <= resize(((unsigned(histo_in_ul)-unsigned(histo_min_ul))*(unsigned(255,8))/(unsigned(numpixels_ul)-unsigned(histo_min_ul)), 8));
    transformed_ur <= resize(((unsigned(histo_in_ur)-unsigned(histo_min_ur))*(unsigned(255,8))/(unsigned(numpixels_ur)-unsigned(histo_min_ur)), 8));
    transformed_ll <= resize(((unsigned(histo_in_ll)-unsigned(histo_min_ll))*(unsigned(255,8))/(unsigned(numpixels_ll)-unsigned(histo_min_ll)), 8));
transformed_lr <= resize(((unsigned(histo_in_lr) - unsigned(histo_min_lr)) * to_unsigned(255, 8)) / (unsigned(numpixels_lr) - unsigned(histo_min_lr)), 8);
end if;

--------------------end of reading block, beginning of writing block--------------------
if (start_add='1' or rst='1') then
clh_out <= (others=>'0');
else -- Transforming routine

clh_out <= std_logic_vector(resize((transformed_ul*(unsigned(x2_pre1-x_pos_pre3))*(unsigned(y2_pre1-y_pos_pre3))+transformed_ur*(unsigned(x_pos_pre3-x1_pre1))*(unsigned(y2_pre1-y_pos_pre3))+transformed_ll*(unsigned(x2_pre1-x_pos_pre3))*(unsigned(y_pos_pre3-y1_pre1)) + transformed_lr*(unsigned(x_pos_pre3-x1_pre1))*(unsigned(y_pos_pre3-y1_pre1)))/((x_size*y_size), 8));
end if;

if rst='1' or start_cntr='1' then
start <= '1';
else
start <= '0';
end if;

end if;
end process;
x1 <= x_ref-signed(x_size)-1;
x2 <= x_ref;
x_ref <= signed(shift_right(((shift_left((resize(x_size,11)),1)/2)+1),1))+signed(x_size*num_x);
y_ref <= signed(shift_right(((shift_left((resize(y_size,11)),1)/2)+1),1))+signed(y_size*num_y);
y1 <= y_ref-signed(y_size)-1;
y2 <= y_ref;
u1_id_pre <= num_x+num_y*10;
u1_id <= ul_id_pre;
u1_id <= ul_id_pre + 1;
1l_id <= ul_id_pre + 10;
1l_id <= ul_id_pre + 11;
histo_raddr <= std_logic_vector(pixel);
-- Computation of the equalized pixel
rom_raddr <= std_logic_vector(im_raddr);
wren <= wren1;
numpixels2 <= numpixels-1;
end transformation;
C. Modelsim simulations

Global timeline: main entity view
In this figure the signals corresponding to the top level entity are shown: the input parameters, which are constant, and also the signals of access to the different memories. The signals corresponding to each block’s access are distinguishable: the sorter ones at the beginning are accesses by the binary mask generation block, the ones that are longer are from the CLAHE block. The signals that have activity during the second part are controlled by the filtering block.
Global timeline: CLAHE block
FPGA Implementation of a Contrast Enhancement Algorithm
In the first capture the signals are mostly memory accesses/writes. 2 main phases of the CLAHE computation are visible here: all the computation needed to have ready a transformation function and the transformation process itself. In the second capture, the control signals of the tile computation are visible as well: one can appreciate the start impulses of each tile and CLAHE sub-block, as well as the end flags and the signals that store which is the tile that is being computed at the moment.
Global timeline view: binary mask generation
This is clearly the shortest part. It takes almost only the time necessary to read all the image. The different inputs and outputs of the mask correction block can be seen, both memories and parameters, as well as recalculated image sizes for the addition of zero padding.
Global time-line: filter block view
This part, the filtering block, is more or less like the previous one because it consists basically in the same kind of windowing implementation, with different operations. The difference is that there are more filters, which are pipelined, hence the presence of more signals with high activity that transport the image before and after each stage, with some delay. Note how, as said in main, it starts when the other steps have finished. The signal that is red (unknown status) during most of the time is the output of the first address of the memory that stores the CLAHE image. It is not problematic because when the CLAHE is computed the image is overwritten with the desired known value and the unspecified value does not impact any part of the filtering system.
7. References


