

EFFECTIVENESS OF EDGE DETECTION OF COLOR IMAGES

By

PALLABI KUNDU

POST GRADUATE DIPLOMA IN INFORMATION AND COMMUNICATION
TECHNOLOGY



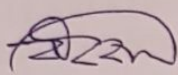
Institute of Information and Communication Technology (IICT)

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY (BUET)

December 2020

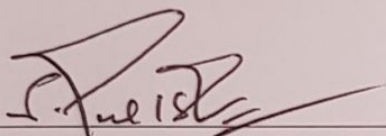
This project titled "EFFECTIVENESS OF EDGE DETECTION OF COLOR IMAGES" submitted by PALLABI KUNDU, Roll No: 1017311003, Session: October,2017, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Post Graduate Diploma in Information and Communication Technology on 19 December 2020.

BOARDS OF EXAMINERS



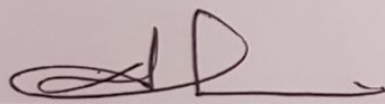
1. Dr. Md. Rubaiyat Hossain Mondal
Professor
IICT, BUET, Dhaka

Chairman
(Supervisor)



2. Dr. Md. Saiful Islam
Professor
IICT, BUET, Dhaka

Member

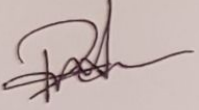


3. Dr. Md. Liakot Ali
Professor
IICT, BUET, Dhaka

Member

Candidate's Declaration

It is hereby declared that this report or any part of it has not been submitted elsewhere for the award of any degree and diploma.



PALLABI KUNDU
ID: 1017311003

Table of Contents

Title	Page No.
Board of Examiners	ii
Candidate's Declaration	iii
Dedication	iv
Table of Contents	v
List of Figures	vii
Abbreviation & Key Terms	ix
Acknowledgement	x
Abstract	xi
 Chapter 1: Introduction	
1.1 Introduction	01
1.2 Objective with specific aims and possible outcome	03
1.3 Outline of this Report	03
 Chapter 2: Theory of Edge Detection	
2.1 Image File Formats	04
2.2 Bitmap	04
2.3 Edge Detection	04
2.4 Related work of Edge Detection	06
 Chapter 3: Edge Detection Methods	
3.1 Methods of Edge Detection	08
3.1.1 First-Order Derivative	08
3.1.2 Second-Order Derivative	09
3.2 Types of Edge Detection	10
3.2.1 Roberts Edge Detection	10
3.2.2 Sobel Edge Detection	10
3.2.3 Canny Edge Detection	11
3.2.4 Prewitt Edge Detection	11
3.2.5 Kirsch Edge detection	12
3.2.6 Robinson Edge detection	12
3.2.7 LoG edge detection	13
3.3 Sobel Operator	13
 Chapter 4: Edge Detection System Design	
4.1 System Architecture	15
4.2 Software and Simulation Tools	15
4.3 VHDL Implementation	16

4.3.1	(3x3) Sliding Window Architecture	18
4.3.2	Sobel Operator Module	19
4.3.3	Cache System Module	20
	A. Fifo Linebuffer Module	21
	B. Rows and Columns Counter	24
	C. Sync_signals_delayer	25
4.3.4	Sobel Kernel Module	25
4.4	Python Implementation	26
4.4.1	(3 x 3) Matrix Kernel Convolution Method	27

Chapter 5: Experimental Result

5.1	VHDL Test Bench Simulation	29
5.2	Simulation with Python	30

Chapter 6: Conclusion

6.1	Conclusion	33
6.2	Future Work	33

References	38
-------------------	-----------

List of Figures

Figure No.	Figure Caption	Page No.
Figure 2.1	256×256 image with 3×3 neighborhood of pixels	05
Figure 2.2	Contents of convolution table to detect edge at coordinate [i, j]	05
Figure 2.3	Nested loops to move convolution table over image	06
Figure 2.4	Coordinates of 3×3 convolution table	06
Figure 4.1	Block diagram of proposed system	15
Figure 4.2	Structural block diagram of Sobel Edge Detection using VHDL	16
Figure 4.3	Color to Gray Conversion in VHDL	17
Figure 4.4	Store header data to a file	18
Figure 4.5	Architecture of 3x3 Sliding Window Module	19
Figure 4.6	Schematic Diagram of 3x3 Sliding Window	19
Figure 4.7	Edge-sobel-wrapper Module	19
Figure 4.8	Edge-sobel-wrapper Internal Connection	20
Figure 4.9	Convolving a 3x3 mask with an image. With numbers from 1 to 9 are marked the relevant positions of the mask	21
Figure 4.10	Memory Architecture	21
Figure 4.11	Cache-system Internal Structure	22
Figure 4.12	FIFO-linebuffer Module	23
Figure 4.13	Read an image from filesystem in VHDL	24
Figure 4.14	Write edges into memory in VHDL	24
Figure 4.15	Double-fifo-linebuffer Internal Structure	24
Figure 4.16	Logical Division of Pixel Elements Variables	25
Figure 4.17	Sliding Window Process in VHDL	25
Figure 4.18	Rows Counter Connections	26
Figure 4.19	Columns Counter Connections	26
Figure 4.20	Counting row and columns in VHDL	27
Figure 4.21	Sobel-kernel Module	28
Figure 4.22	Sobel Kernel in VHDL	28
Figure 4.23	Using threshold value in VHDL	29
Figure 4.24	Libraries used in Python	29
Figure 4.25	Structural block diagram of Sobel Edge Detection using Python	30
Figure 4.26	Image read and gray scale conversion in Python	30
Figure 4.27	Gradient Calculation in Python	30
Figure 4.28	Thresholding and edge calculation in Python	31
Figure 5.1(a)	Image 1-Color test	32
Figure 5.1(b)	Edges of Image-1	32
Figure 5.2(a)	Image 2-Car	32
Figure 5.2(b)	Edges of Image-2	32
Figure 5.3	Simulation of Image 1	33
Figure 5.4	Simulation of Image 2	33
Figure 5.5(a)	Image 1-Color test	34
Figure 5.5(b)	Edges of Image-1 (x-axis)	34
Figure 5.5(c)	Edges of Image-1 (y-axis)	34
Figure 5.5(d)	Edges of Image-1 (weighted sum)	34
Figure 5.6(a)	Image 2-Car	35

Figure 5.6(b)	Edges of Image 2 (x-axis)	35
Figure 5.6(c)	Edges of Image 2 (y-axis)	35
Figure 5.6(d)	Edges of Image 2 (weighted sum)	35

Abbreviation & Key Terms

RGB	Red, Green, and Blue
YUV	"Y" means luminance "U" and "V" means chrominance
CMY	Cyan Magenta Yellow
CMYK	Cyan, Magenta, Yellow and Black
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
OPEN CV	Open Source Computer Vision Library
PIL	Python Imaging Library
PNG	Portable Network Graphics
JPEG	Joint Photographic Experts Group
GIF	Graphics Interchange Format
BMP	Bitmap
GIMP	GNU Image Manipulation Program
FPGA	Field Programmable Gate Array
ASCII	American Standard Code for Information Interchange
LoG	Logarithm
FIFO	First In, First Out
HLS	High-Level Synthesis
UUT	Unit Under Test
VGA	Video Graphics Array
PYNQ	Open-source project from Xilinx® using Python Language

Acknowledgement

At first, I would like to convey my gratitude to Almighty for providing me the opportunity to accomplish the project work. After that, I want to express the deepest appreciation and gratefulness to my supervisor **Dr. Md. Rubaiyat Hossain Mondal**, Director and Professor of IICT, BUET for giving me a chance to explore such an interesting field of research. I have successfully completed the goal of the project due to his tireless and patient monitoring throughout the project whenever I needed it. Without his proper advice and active involvement in this process of work, it would not be possible to complete the work.

In addition, I am extremely thankful to all the faculty members of IICT, BUET who provided me the proper ideas and perception to explore different IT related subjects and future possibilities. I would also like to thank all the officers and staffs of IICT, BUET for providing their kind support and information for the successive completion of my Post Graduation Diploma degree.

Moreover, I would like to thank to all my classmates for giving me proper support and any type of information about the degree program whenever it is needed.

Finally, I am so grateful to my parents and my beloved husband who were always besides me with their continuous supports and encourages.

Abstract

Digital image processing is an ever-expanding and dynamic field, and its applications have involved our daily lives, such as medicine, space exploration, surveillance, identity verification, automatic industrial inspection, and intelligent transportation systems, self-driving cars and self-driving cars, Guided weapons, etc. Applications such as these involve different processes, such as image enhancement and object detection. The edge detection process can simplify image analysis by greatly reducing the amount of data to be processed, while retaining useful structural information about object boundaries. Efficient and accurate edge detection will ensure the performance of the further image processing stage. Edge detection uses different methods, such as Sobel, Prewitt, Canny and other operators. Edge detection based on Sobel operator has been widely used in edge detection, becoming a real-time gray image solution. Nowadays, color image processing has attracted more and more attention, so more research has been done in this field, because edge detection in color images is more complicated than edge detection in gray images. In this project, Very-high-speed integrated circuit Hardware Description Language (VHDL) and Python language are used to study the Sobel edge detection technology for color image segmentation. VHDL is a general-purpose parallel programming language used to describe digital and mixed-signal systems and used in real-time embedded system automation for documentation, simulation, verification, and synthesis. On the other hand, Python provides highly optimized libraries from a large developer community, and it is used in all fields from scientific computing to image processing and machine learning. Therefore, this research focuses on the effectiveness of color image edge detection based on Sobel operator using these two methods.

CHAPTER-1

Introduction

1.1 Introduction

Digital image processing is an exciting field for obtaining pictorial information about processing of data from different images for storage, transmission, and representation. Image processing is a technique used to add raw images that are expected from cameras or sensors placed on satellites, space probes, and airplanes, or photos taken in normal daily life for multiple applications. This field of image processing shows higher quality in modern times and in the field of science and technology. Image processing involves image acquisition, image enhancement, image segmentation, feature extraction, and image classification.

Digital image processing proposes a general method of edge detection. The edge detection process simplifies image analysis by greatly reducing the amount of data to be processed, while retaining useful structural information about object boundaries [1]. The edge includes noteworthy features and contains important information. It reduces the size of the image and filters out less relevant information, thereby preserving the important attributes of the image [2]. In the conversion process from color image to grayscale image, due to the loss of color information, edge detection in grayscale image cannot provide completely accurate results. Therefore, to get the desired accuracy, it is necessary to detect edges in a color image. [3][16]

Edge detection in color images is more challenging than grayscale images because the color space is considered as vector space. Nearly 90% of the edge information in the color image can be found due to the corresponding grayscale image. However, the remaining 10% is still important in certain computer vision tasks. There are different types of models for color images, such as RGB color model, YUV model, CMY color model, CMYK color model, HIS color model, etc. [4].

The method of edge detection is the First-Order Derivative (gradient method), the Second-Order Derivative and the Optimal Edge Detection to detect the edges. The type of edge detection belongs to the Second-Order Derivative, which are Laplacian, Laplacian of Gaussian, and Difference of Gaussian. The Second-Order Derivative is very sensitive to the noise present in the image, that is why it is generally not used for edge detection operations [5]. But the Second-Order Derivative is used to extend some auxiliary information, such as to determine whether the point is located on the darker side of the

image or on the lighter side of the image. On the other hand, First-Order Derivative can be calculated using gradient operators. [6]

Today, the most challenging problems in designing an efficient driver assistance system are the choice of perception modules and the development of robust and accurate data processing algorithms. Edge detection uses different methods, such as Sobel, Prewitt, Canny and other operators. Edge detection based on Sobel operator has been widely used in edge detection, becoming a real-time gray-scale image solution. [7] [8]

Sobel operator is one of the First-Order Derivative families because it is a discrete differential operator, which calculates the gradient approximation of the image intensity function. At each point of the image, the result of the Sobel operator is the corresponding gradient vector or the norm of the vector. The Sobel operator is based on convolving the image using small, separable integer-valued filters in the horizontal and vertical directions, so it is relatively cheap to calculate. [9] [10]

In this project, color image edge detection based on Sobel operator can be achieved. For each color component in the RGB space, a specific edge computing processor is developed. Since the Sobel operator is a sliding window operator, it uses a smart buffer-based memory architecture to move the input pixels in the computing window. Designed a specific data path and chose a gradient to accomplish the task.

This research work discusses the development and performance of an optimized and high-resolution color image edge detection algorithm based on the Very High-Speed Integrated Circuit Hardware Description Language (VHDL). VHDL is a general-purpose parallel programming language to describe digital and mixed signal systems and used in real time embedded systems automation for documentation, simulation, verification, and synthesis. [11] [12]

In addition, in this project, the performance of the high-resolution color image edge detection process using Python will be achieved. Python is an interpreted high-level general-purpose programming language. The Python language provides many free and open source software packages, such as the Video Capture library, PIL library and OPEN CV (the computer version of the open source library) library in image processing, which left researchers with initial impressions. Open CV introduces a new set of tutorials that will guide the various functions available in Open CV-Python with the support of Numpy. [13] [14]

This research is determined to do an analysis of the Sobel operator based high resolution color image edge detection with VHDL and Python. Finally, to investigate the effectiveness of the edge detection of color images.

1.2 Objective with specific aims and possible outcome

The objective of this work is to find out the effectiveness of VHDL and Python based image processing for accurate edge detection. To fulfill this objective, the project has the following specific aims:

- To detect the edges of high-resolution color images using Sobel operator implemented in VHDL programming.
- To apply Sobel operator-based image processing scheme in Python programming.

The possible outcome of this work is the development of edge detection systems using VHDL and Python.

1.3 Outline of this Report

The rest of the report is organized as follows. Chapter 2 presents the related works, Chapter 3 presents the theoretical concept of the study, Chapter 4 describes the methodology, and Chapter 5 presents performance results obtained from computer simulations. Finally, Chapter 6 provides the concluding remarks and future research directions.

CHAPTER-2

Theory of Edge Detection

2.1 Image File Formats

The image file format is a standardized method for organizing and storing digital images. Image file formats can store data in uncompressed format, compressed format (which may be lossless or lossy), or vector format. Image files consist of digital data in one of these formats, so the data can be rasterized for use. Rasterization converts image data into a grid of pixels. Each pixel has many bits to specify its color. There are hundreds of image file types. The PNG, JPEG and GIF formats are most used to display images on the Internet. But they use compression, JPEG is lossy, and PNG is lossless. Most formats provide some form of compression because this can greatly reduce the storage size of the image.

2.2 Bitmap

To be able to process images in software or hardware, it is necessary to access the raw pixel data in the application. If it is needed to store color and brightness data in a byte matrix, one should use bitmaps or raster graphics.

BMP file format (Windows bitmap) is used to process graphics files in Microsoft Windows OS. Generally, BMP files are uncompressed, so large and lossless; their advantages are their simple structure and wide acceptance in Windows programs.

Most famous image editors (such as Photoshop or GIMP) are based on raster. They can open various image formats, but they can all be converted to raster graphics inside the editor.

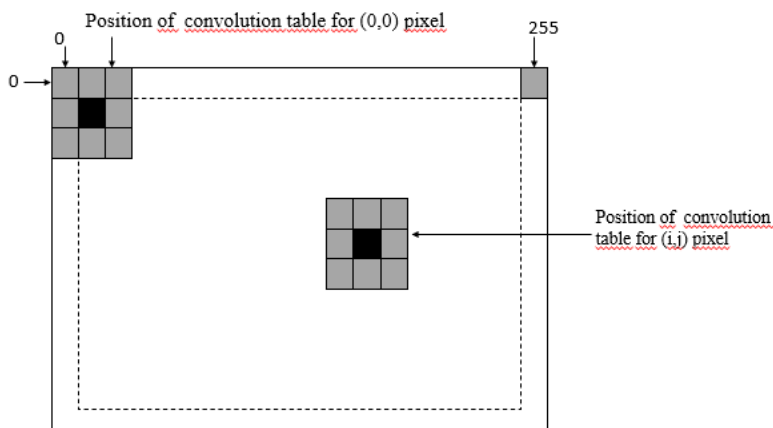
Other image formats for simulation can be used, but since there is no ready-made solution to decode compressed images, a lot of coding work is required. A better solution is to manually convert the test input image to a bitmap format such as BMP or merge it into the script that starts the image.

2.3 Edge Detection

Digital image processing is an expanding and dynamic field, and its application range has been extended to our daily lives, such as medicine, space exploration, surveillance, identity verification, automatic industrial inspection, and intelligent transportation systems, autonomous vehicle and self-propelled weapons. Applications such as these involve different processes, such as image enhancement and object detection. Edge detection includes a variety of mathematical methods, aimed

at identifying points in digital images where the brightness of the image changes sharply or more formally has discontinuities. The points with sharp changes in image brightness are usually organized into a set of curved segments called edges. [18][19]

The operator can be optimized to find vertical, horizontal, or diagonal edges. In noisy images, edge detection is a difficult task because both edges and noise contain high-frequency content. Efforts to reduce noise can result in unclear edges and distortion. The techniques used on noisy images are usually larger; therefore, they can share enough data to reduce local noise pixels. This leads to unsatisfactory positioning of the detected edges. [27][28]



$(i-1, j-1)$	$(i-1, j)$	$(i-1, j+1)$
$(i, j-1)$	(i, j)	$(i, j+1)$
$(i+1, j-1)$	$(i+1, j)$	$(i+1, j+1)$

Figure 2.1: 256×256 image with 3×3 neighborhood of pixels

Figure 2.2: Contents of convolution table to detect edge at coordinate $[i, j]$

In digital image processing, each image is quantized into pixels. For grayscale images, each pixel represents the image brightness level at a specific point. For 8-bit pixels, 0 represents black, and 255 represents white. Edges are sudden changes in pixel brightness. By exploring the pixel brightness near the pixel, the edge information of a specific pixel can be obtained. If all nearby pixels have almost the same brightness, there may be no edges at that point. However, if some neighbors are much brighter than others, there may be an edge at this time. Mathematically, measuring the relative brightness of nearby pixels is like calculating the derivative of brightness. The brightness value is discrete, not continuous, so we approximate the derivative function. There are different edge detection methods. They are Prewitt, Laplacian, Roberts, Sobel, etc. Every method uses different discrete approximations of the derivative function.

In this project, an improved version of the Sobel edge detector algorithm to detect edges in a 640×480 pixel 24-bit bitmap color image is used. When calculating the derivative, the Sobel edge detection algorithm uses a (3×3) pixel table to store pixels and their neighbors. The (3×3) pixel table is called a convolution table because it moves on the image with a convolution style algorithm. In the bitmap,

there are 8 bits assigned to each red, green, and blue component. In each component, a value of 0 means that the color has no contribution, and 255 means that the color is fully saturated. Since each component has 256 different states, there are 16,777,216 possible colors. In addition, bitmap is a native file format used to display images in the Microsoft Windows environment. It is a form of raster file storage with almost no compression.

Figure 2.1 shows the convolution table at two different positions of the image: the first position (calculate whether the pixel at [1,1] is on the edge), and whether the pixel in [i,j] is on the edge.

```

for i=1 to 255-1(
  for j=1 to 255-1(
    for m=0 to 2(
      for n=0 to 2(
        table[m,n] :=image [i+m-1, j+n-1];
      )
    )
  )
)

```

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Figure 2.3: Nested loops to move convolution table over image

Figure 2.4: Coordinates of 3×3 convolution table

Figure 2.2 shows a convolution table, which contains the pixel located at coordinates [i, j] and its eight neighboring pixels. As shown in Figure 2.1, the table moves pixel by pixel on the image. For an image of 256×256 pixels, the convolution table will move in 64516 (254×254) different positions for each color (red, green, and blue). The algorithm in Figure 2.3 shows how to move a 3×3 convolution table on a 256×256 image. The lower and upper limits of the cycle of i and j are 1 and 254, instead of 0 and 255, because we the derivative of the pixels around the image cannot be calculated.

Edges are local changes in image intensity. Edges usually appear on the boundary between two areas. The main features can be extracted from the edges of the image. Edge detection has the main function of image analysis. These functions are used by advanced computer vision algorithms. Edge detection is used for object detection and can be used for various applications such as medical image processing and biometric recognition. Edge detection is an active area of research because it facilitates higher-level image analysis.

2.4 Related work of Edge Detection

When a greatly improved edge detection mask is used, edge detection becomes more complicated. Moreover, when it operates with very high-resolution color images, the process becomes longer. Most

of the hardware implementations are faster than their corresponding software implementations. Therefore, implementing edge detection in hardware will be more effective. Because FPGA has the additional function of parallelism, it can effectively realize edge detection. Efficient edge detection algorithms are usually used in high-speed hardware applications. There are various types of edge detection techniques available in image processing applications. They are the algorithms of Prewitt, Canny, Sobel and Roberts, and they are all different in hardware architecture, performance, and accuracy. Sobel edge detector is one of the simple methods for hardware implementation, making real-time edge detection applications easy. [15]

Conventionally, the two hardware description languages VHDL and Verilog can provide rapid prototyping for digital circuits and systems on FPGA devices, thereby providing an fascinating modeling and simulation environment, and MATLAB and C/C++ are the preferred simulations Tools and test the correctness of computer vision algorithms. MATLAB converts the image to an ASCII text file. The ASCII text file is applied to the hardware interface as a vector. Each frame/image is composed of a series of horizontal lines, and each line is composed of a series of points, also called pixels. The lines in each frame are transmitted sequentially from top to bottom, and the pixels in each row are transmitted from left to right. [17]

Since Xilinx has a lot of information about image processing, every time an image is converted to an ASCII text file for use in MATLAB, it is first saved on the hard disk, and then the file can be opened in VHDL. Similarly, the output file is also saved on the hard disk in the same way and converted and viewed in MATLAB. Of course, since the FPGA board cannot access the hard disk, this is only valid in simulation. This method reduces the complexity of appearance and reduces processing time. For a picture with a size of 256×256 , the execution time of the complete program of edge detection is a few seconds. But the whole process consumes a lot of memory from the hard disk. In addition, MATLAB is expensive (not only for students, but also for commercial applications).[30][31]

Edge Detection Methods

3.1 Methods of Edge Detection

Most different methods can be divided into two categories:

- a) Gradient method: The gradient method detects edges by finding the maximum and minimum values in the first derivative of the image.
- b) Laplacian method: It searches for the zero-crossing point in the second derivative of the image to find the edge. The edge has a one-dimensional shape with a slope and calculating the derivative of the image can highlight its location.

Basically, the derivative shows that the maximum is in the center of the edge in the original signal, and if the gradient value exceeds a certain threshold, the pixel position is declared as an edge position. Because the edge will have a higher pixel intensity value than the surrounding pixels. Therefore, once the threshold is set, it can compare the gradient value with the threshold and detect an edge when the threshold is exceeded.

3.1.1 First-Order Derivative

Most edge detection methods assume that edges occur where the intensity function is discontinuous or where the image has a very steep intensity gradient. Using this assumption, if the intensity value on the entire image is derived and the point with the largest derivative is found, the edge can be located. Gradient is a vector whose components can measure the speed of change of pixel values in the x and y directions. Therefore, the components of the gradient are found by using the following equations (1) and (2).

$$\partial f(x, y) / \partial x = (f(x + dx, y) - f(x, y)) / dx = \Delta x \dots \dots \dots (1)$$

$$\partial f(x, y) / \partial y = (f(x, y + dy) - f(x, y)) / dy = \Delta y \dots \dots \dots (2)$$

Where dx and dy measure the distance along the x and y directions, respectively. In a discrete image, dx & dy can be considered based on the number of pixels between two points, dx = dy = 1 (pixel pitch)

is the point with pixel coordinates (i, j) , so $(\Delta x$ and $\Delta y)$ can be obtained by calculating the formula (3) and (4).

$$\Delta x = f(i + 1, j) - f(i, j) \dots \dots \dots (3)$$

$$\Delta y = f(i, j + 1) - f(i, j) \dots \dots \dots (4)$$

To detect the presence of gradient discontinuities, the change in gradient at (i, j) can be calculated. This can be done by finding the following magnitude measure, and the gradient direction θ is given by equation (5).

$$\theta = \tan^{-1} \left[\frac{\Delta y}{\Delta x} \right] \dots \dots \dots (5)$$

3.1.2 Second-Order Derivative

The Laplacian is a two-dimensional measure of the second derivative of an image. The Laplacian of the image highlights areas with rapid changes in intensity, so it is usually used in edge detection zero-crossing edge detectors. The Laplacian operator is usually applied to the image smoothed by the approximate Gaussian smoothing filter to reduce its sensitivity to noise. Operators usually take a single grayscale image as input and generate another binary image as output. The zero-crossing detector looks for the position where the value of the Laplacian crosses zero in the Laplacian of the image, that is, the point where the Laplacian changes the sign. These points usually appear on the edges of the image, that is, the points where the intensity of the image changes rapidly, but they also appear at locations that are not easily associated with the edges. It is best to think of the zero-crossing detector as feature detector rather than a specific edge detector. The zero-crossing is always on the closed contour, so the output of the zero-crossing detector is usually a binary image, in which a single pixel thick and thin line shows the position of the zero-crossing point. Define the derivative operator Laplacian of the image, as shown in formulas (6), (7) and (8):

$$\Delta^2 f = (\partial^2 f)/(\partial x^2) + (\partial^2 f)/(\partial y^2) \dots \dots \dots (6)$$

$$\text{For X - direction, } (\partial^2 f)/(\partial x^2) = f(x + 1, y) + f(x - 1, y) - 2f(x, y) \dots \dots \dots (7)$$

$$\text{For Y- direction, } (\partial^2 f)/(\partial y^2) = f(x, y + 1) + f(x, y - 1) - 2f(x, y) \dots \dots \dots (8)$$

By substituting, Equations (7) and (8) in (6), we obtain the equation (9)

$$\Delta^2 f(x, y) = f(x + 1, y) + f(x - 1, y) + f(x, y + 1) + f(x, y - 1) - 4f(x, y) \dots \dots \dots (9)$$

If we use the value of Mask as explain below, we obtain on equation (9):

0	1	0
1	-4	1
0	1	0

3.2 Types of Edge Detection

There are many edge detection techniques in the literature for image segmentation. This section introduces the most used edge detection techniques based on discontinuities. These technologies include Roberts edge detection, Sobel edge detection, Canny edge detection, Prewitt edge detection, Kirsh edge detection, Robinson edge detection, LoG edge detection, etc. [26][29].

3.2.1 Roberts Edge Detection

Roberts edge detection was proposed by Lawrence Roberts (1965). It can perform simple and fast two-dimensional spatial gradient measurement on images. This method emphasizes high spatial frequency regions that usually correspond to edges. The operator's input is a grayscale image, and the output is the most common use of this technique. The pixel value of each point in the output represents the estimated complete amplitude of the spatial gradient of the input image at that point.

-1	0
0	+1

G_x

0	-1
+1	0

G_y

3.2.2 Sobel Edge Detection

The Sobel edge detection technique is one of the standard edge detection techniques. This edge detection method was introduced by Sobel in 1970 (Rafael C.Gonzalez (2004)). The Sobel method of edge detection for image segmentation finds edges using the Sobel approximation to the derivative. It is before the edges of the points with the highest gradient. Sobel technology performs a two-dimensional spatial gradient on the image, thus highlighting the high spatial frequency area corresponding to the edge. Usually, it is used to find the estimated absolute gradient size at each point of n input grayscale images. It can be speculated that at least the operator consists of a pair of 3x3 complex kernels, as shown below the table. One core just rotates the other by 900. This is very similar to the Roberts Cross operator.

-1	-2	-1
0	0	0
+1	+2	+1

 G_x

-1	0	-1
-2	0	+2
-1	0	+1

 G_y

3.2.3 Canny Edge Detection

Canny edge detection technology was first proposed by John Canny in his master's degree thesis at MIT in 1983. Canny is a very important method to find the edge by separating the noise from the image before finding the edge of the image. After that, Canny applied the trend of finding the edge and the severity value of the threshold without disturbing the edge features of the image. The algorithm steps are as follows:

- Convolve image $f(r, c)$ with a Gaussian function to get smooth image $f^\wedge(r, c)$. $f^\wedge(r, c) = f(r, c) * G(r, c, \sigma)$
- Apply first difference gradient operator to compute edge strength then edge magnitude and direction are obtaining as before.
- Apply non-maximal or critical suppression to the gradient magnitude.
- Apply threshold to the non-maximal suppression image.

3.2.4 Prewitt Edge Detection

Prewitt edge detection was proposed by Prewitt in 1970 (Rafael C. Gonzalez [1]). Prewitt is the correct method to estimate the amplitude and direction of the edge. Even different gradient edge detection requires quiet time-consuming calculations to estimate the direction, from x And the magnitude of the y direction, compass edge detection directly obtains the direction from the kernel with the highest response, and it is limited to 8 possible directions; however, knowledge shows that most direct direction estimates are not so perfect. In the 3x3 neighborhood, for the 8-direction estimation is based on the gradient-based edge detector, which calculates all 8 convolution masks, and then selects a complex mask, that is, the purpose of the largest module.

-1	-1	-1
0	0	0
+1	+1	+1

 G_x

-1	0	+1
-1	0	+1
-1	0	+1

 G_y

Prewitt detection is slightly simpler to implement computationally than the Sobel detection, but it tends to produce somewhat noisier results.

3.2.5 Kirsch Edge detection

Kirsch edge detection was introduced by Kirsch (1971). The mask of this Kirsch technique is defined by considering a single mask and rotating it to the eight main compass directions: North, Northwest, West, Southwest, South, Southeast, East, and Northeast. The difference between masks is as follows:

$$\begin{array}{cccc}
 K_0 & K_1 & K_2 & K_3 \\
 E = \begin{bmatrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{bmatrix} & NE = \begin{bmatrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{bmatrix} & N = \begin{bmatrix} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix} & NW = \begin{bmatrix} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix} \\
 K_4 & K_5 & K_6 & K_7 \\
 W = \begin{bmatrix} 5 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & -3 & -3 \end{bmatrix} & SW = \begin{bmatrix} -3 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & 5 & -3 \end{bmatrix} & S = \begin{bmatrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{bmatrix} & SE = \begin{bmatrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & 5 & 5 \end{bmatrix}
 \end{array}$$

The edge amplitude is defined as the maximum value obtained by convolving each mask with the image. The direction is defined by the mask that produces the largest amplitude. For example, mask K_0 corresponds to a vertical edge, while mask K_5 corresponds to a diagonal edge. Note that the last four masks are the same as the first four masks but flipped around the central axis.

3.2.6 Robinson Edge detection

Robinson methods (Robinson 1977) are like Kirsch masks but are easier to implement because they only rely on coefficients 0, 1, and 2. These masks are symmetrical about their direction axis (the axis with zero). One only needs to calculate the results on the four masks, while the results of the other four can be obtained by inverting the results of the first four. The mask is as follows:

$$\begin{array}{cccc}
 r_0 & r_1 & r_2 & r_3 \\
 E = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} & NE = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix} & N = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} & NW = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix} \\
 r_4 & r_5 & r_6 & r_7 \\
 W = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} & SW = \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix} & S = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} & SE = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}
 \end{array}$$

The size of the gradient is the maximum value obtained by applying all eight masks to the pixel neighborhood, and the angle of the gradient can be approximated to the angle of the zero line in the mask to produce the maximum response.

3.2.7 LoG edge detection

The Laplacian of Gaussian (LoG) was proposed by Marr (1982). The LoG of the image $f(x,y)$ is the second order derivative defined as,

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

It has two effects, it smoothes the image and calculates the Laplacian, thereby producing a double-edge image. Then, locating the edges includes finding the zero crossing between the double edges. The digital realization of the Laplace function is usually realized by the following mask,

$$\begin{array}{ccc}
 \begin{array}{|c|c|c|}
 \hline
 0 & -1 & 0 \\
 \hline
 -1 & 4 & -1 \\
 \hline
 0 & -1 & 0 \\
 \hline
 \end{array} & & \begin{array}{|c|c|c|}
 \hline
 -1 & -1 & -1 \\
 \hline
 -1 & 8 & -1 \\
 \hline
 -1 & -1 & -1 \\
 \hline
 \end{array} \\
 G_x & & G_y
 \end{array}$$

The Laplacian is generally used to find whether a pixel is on the dark or light side of an edge.

3.3 Sobel Operator

The Laplacian is generally used to find whether the Sobel operator is an example of the gradient method. It is a discrete differential operator used to calculate the gradient approximation of the image intensity function. This operator uses two (3×3) kernels convolved with the original image to calculate the approximation of the derivative-one for the horizontal change and one for the vertical change. If we define A as the source image, and G_x and G_y are two images, and each image contains approximate values of vertical and horizontal derivatives at each point, the calculation is as follows: a pixel is on the dark or light side of an edge.

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A \quad \text{and} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * A$$

Where (*) denotes the 2-dimensional signal processing convolution operation.

Since the Sobel kernel can be decomposed into the product of the average kernel and the differential kernel, the gradient can be calculated by smoothing. For example, G_x and G_y can be written as,

$$G_x = \begin{bmatrix} +1 \\ +2 \\ +1 \end{bmatrix} * ([-1 \ 0 \ +1] * A) \quad \text{and} \quad G_y = \begin{bmatrix} -1 \\ 0 \\ +1 \end{bmatrix} * ([+1 \ +2 \ +1] * A)$$

The x-coordinate is defined here as increasing in the "right"-direction, and the y-coordinate is defined as increasing in the "down"-direction. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using:

$$G = \sqrt{G_x^2 + G_y^2}$$

Using this information, we can also calculate the gradient's direction:

$$\theta = \text{atan} \left(\frac{G_y}{G_x} \right)$$

Where, for example, θ is 0 for a vertical edge which is lighter on the right side.

An important consideration in implementing neighborhood operations is the question of what happens when the center of the kernel is close to the image boundary. It is indeed possible that when the center of the mask is exactly on the border pixel, one or more rows or columns of the mask are outside the image. In this case, there are different approaches:

1. Limit the offset of the mask to ensure that every part of the mask is inside the image. However, the image obtained using this method will be smaller than the original image, and the intensity of the reduction depends on the size of the mask.
2. Only use a part of the mask that is completely contained in the image. In this case, the size of the image is the same as the original size, but some pixels near the border are calculated using only a part of the mask.
3. Surround the image with rows and columns of constant gray level. The most common value is zero, but sometimes the same pixels as near the copied border are used. As with the previous technology, the generated image will have the same size as the original image, but the pixels near the border will be destroyed.

Edge Detection System Design

4.1 System Architecture

The system design for this project is as shown in Figure 4.1. Here the input image is taken from computer. The processing is done on input image which basically includes resizing and filtering. After this Sobel edge detection algorithm is designed using system generator block sets. Then the output image is displayed in the computer which provides edge detected result of input image.

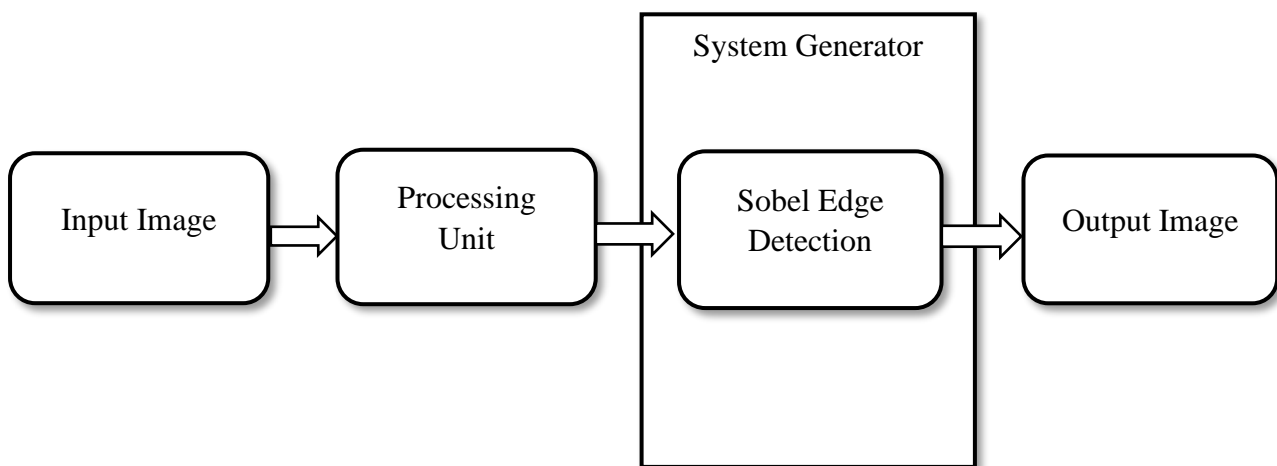


Figure 4.1: Block diagram of proposed system

4.2 Software and Simulation Tools

Edge detection based on Sobel operator has been widely used in edge detection, becoming a real-time gray-scale image solution. Nowadays, color image processing has attracted more and more attention. Therefore, because the edge detection in color images is more complicated than that in gray images, there are more and more research in this field. In this project, an attempt is made to study the Sobel edge detection technique for color image segmentation by using Very High-Speed Integrated Circuit Hardware Description Language (VHDL) and by using Python language. VHDL is a general-purpose parallel programming language used to describe digital and mixed-signal systems and used for recording, simulation, verification, and synthesis in real-time embedded system automation. On the other hand, Python provides highly optimized libraries from a large developer community, and it is used in all fields from scientific computing to image processing and machine learning [24][25].

4.3 VHDL Implementation

In the first state, a design is created and simulated in VHDL. The VHDL code will be run with the help of Xilinx Vivado 2020.1 software tool. Real-time Sobel operator based RGB color bitmap image will be taken as the input directly from the computer.

In Figure 4.2 are depicted the main blocks of the project using VHDL. The module edge-sobel-wrapper implements the Sobel algorithm for edge detection.

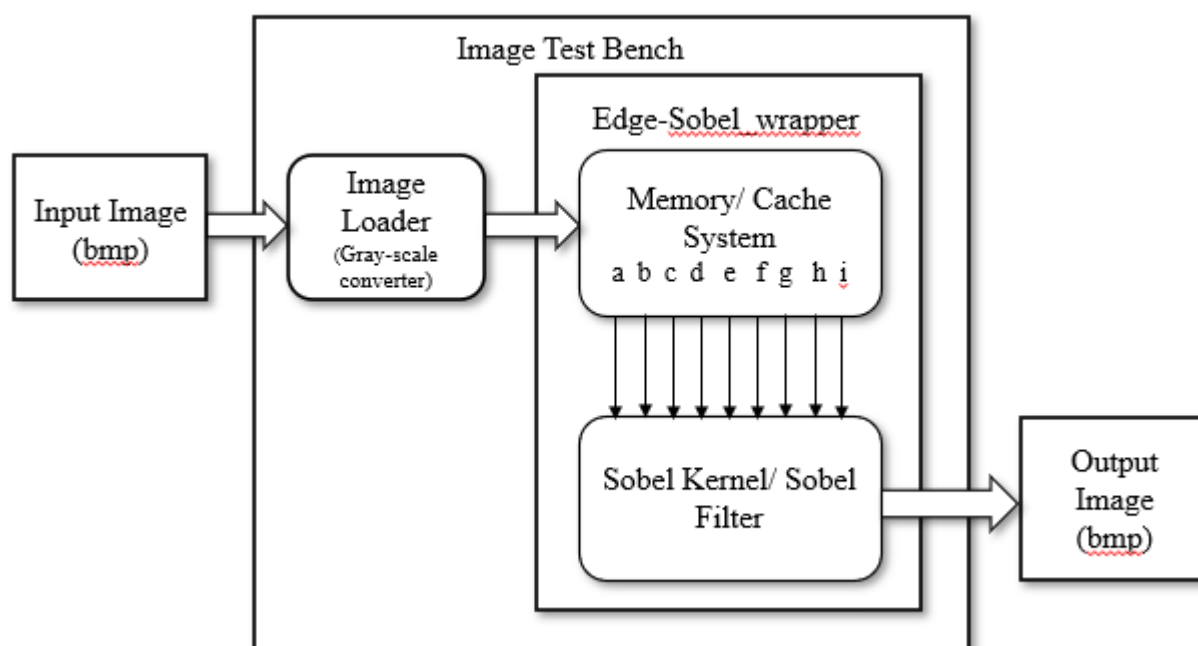


Figure 4.2: Structural block diagram of Sobel Edge Detection using VHDL

BMP is a simple way to store images in digital format. For each BMP file, there is a header part that will provide a lot of information for the application (software), such as the image resolution (the number of pixels in the horizontal and vertical directions), whether the image is in color or grayscale, etc. Used to view images (such as MS Paint, Photoshop, etc.). After the header, there will be image data. As a large matrix, where each row represents each row in the image, and each element in the row represents a pixel. The number of lines in an image is basically the vertical resolution, and the number of pixels in a line is the horizontal resolution. For color images, 3 bytes (24 bits) are used to represent each pixel (dot) in the image. Red 1 byte, green 1 byte, blue 1 byte. Therefore, color images are usually called RGB images. For grayscale images, each pixel is represented by 1 byte. Grayscale images have no other colors. A grayscale image is usually called a black and white image, but strictly speaking it is not because it has a different shade of black (gray). It is also called a binary image. Since in a

grayscale image, each pixel is represented by one byte (8 bits), each pixel can have 256 possible values (0 to 255). Here 0 represents black, 255 represents white, and all values in between represent different shades of black. Image processing for edge detection, it is easier if the image is in grayscale instead of RGB format. Therefore, the first task is to convert RGB images to grayscale images. After extensive research on the sensitivity of the human eye (brain?) to different colors, people discovered how to convert the 3 bytes representing R, G, and B into a single byte representing grayscale. As it is found that human beings are most sensitive to green, then to red, and finally to blue. A formula was developed on this basis,

$$Y = 0.3R + 0.59G + 0.11B$$

Where Y is the gray level (also called brightness or brightness). Figure 4.3 shows the code for converting from color image to grayscale image in VHDL.

```

121 |
122 |         if stop = '0' then
123 |             read(infile, pixelB); -- B
124 |             read(infile, pixelG); -- G
125 |             read(infile, pixelR); -- R
126 |             pixel1 := (ByteT'pos(pixelB)*0.11) + (ByteT'pos(pixelR)*0.3) + (ByteT'pos(pixelG)*0.59);
127 |             pdata_o <= CONV_STD_LOGIC_VECTOR(INTEGER(pixel1), 8);
128 |             col_o    <= col;
129 |             row_o    <= row;
130 |         end if;

```

Figure 4.3: Color to Gray Conversion in VHDL

When converting an RGB image to grayscale, the size of the new image will be approximately 1/3 of the original image. Once converted to grayscale, one pixel in the image is selected at a time and an operation called "neighborhood operation/sliding window operation" is applied. This means that the value of a pixel will change according to the value of its neighboring pixels. For Sobel operation, we usually consider 8 neighbors.

Before the sliding window operation, the header data of the image should be stored, because the processed image data (edge) must be written into a new file at the end of the test bench. Then, the original header needs to be included in the output image.

```

88 ⊕ for i in 0 to 53 loop -- read header infos
89 ⊕   read(infile, pixel);
90 ⊕   write(outfile, pixel);
91 ⊕   case i is
92 ⊕     when 18 => -- 1st byte of cols
93 ⊕       cols(7 downto 0 ) := To_Stdlogicvector(int2bit_vec(ByteT'pos(pixel), 8));
94 ⊕     when 19 => -- 2nd byte of cols
95 ⊕       cols(15 downto 8) := To_Stdlogicvector(int2bit_vec(ByteT'pos(pixel), 8));
96 ⊕     when 22 => -- 1st byte of rows
97 ⊕       rows(7 downto 0 ) := To_Stdlogicvector(int2bit_vec(ByteT'pos(pixel), 8));
98 ⊕     when 23 => -- 2nd byte of rows
99 ⊕       rows(15 downto 8) := to_Stdlogicvector(int2bit_vec(ByteT'pos(pixel), 8));
100 ⊕     when 24 => -- do important things
101 ⊕       cols_o <= cols;
102 ⊕       rows_o <= rows;
103 ⊕       cols := cols - 1;
104 ⊕       rows := rows - 1;
105 ⊕     when others =>
106 ⊕       null;
107 ⊕   end case;
108 ⊕ end loop; -- i

```

Figure 4.4: Store header data to a file

Then, to implement the sliding window operator, a smart buffer-based memory architecture (as described earlier) will be used to move the input pixels in the calculation window, and the Sobel filter/Sobel kernel will calculate the gradient components on adjacent rows or columns, respectively. If the approximate value of the gradient is greater than the user-defined threshold, the comparator output for that color component will be 1 or 0. Finally, the Edge-sobel-wrapper module will find out whether the pixel is an edge. Each process inside the module will run in parallel for three independent processes in the red, green, and blue channels.

4.3.1 (3x3) Sliding Window Architecture

By using the FPGA-based parallel architecture, color image edge detection based on real-time Sobel operator can be realized. For each color component in the RGB space, a specific edge computing processor is developed. Since the Sobel operator is a sliding window operator, it uses a smart buffer-based memory architecture to move the input pixels in the calculation window.

The proposed method uses a 3×3 sliding window architecture. Within a 3×3 sliding window, three first-in first-out (FIFO) buffers are used to reduce memory access to one pixel per clock cycle. To access all the values of the window every clock cycle, these FIFOs must be full. When the signal "start-win" is valid, the content of the window is shifted to the right, and the input data "d" with a width of 8 bits is read. Thereafter, during the rising edge of the clock "clk", the pixel value of the input "din [7:0]" is sent to the sliding window module pixel by pixel in raster scan order. Figure 4.6 shows a schematic diagram of a 3×3 sliding window for an 8-bit pixel row buffer. When the data valid for the sliding

window "dwin" is determined, the 8-bit output from the sliding window module "w11" to "w33" will be sent to the next module to calculate the gradient.

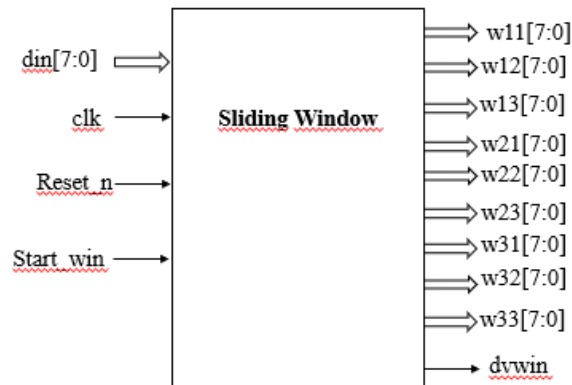


Figure 4.5: Architecture of 3x3 Sliding Window Module

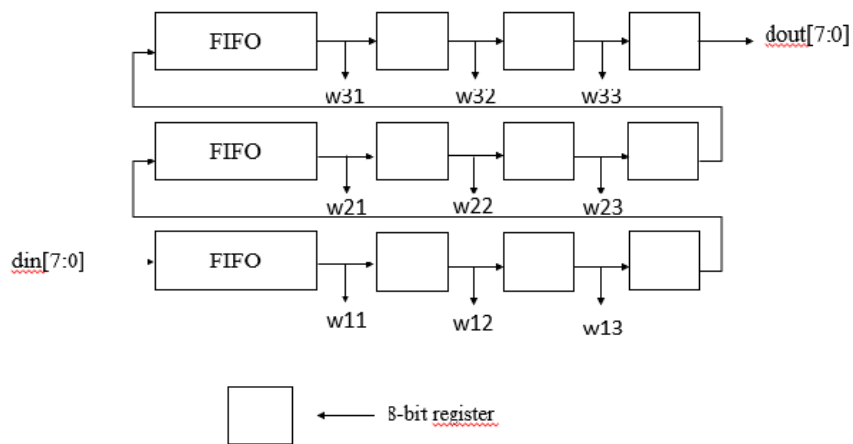


Figure 4.6: Schematic Diagram of 3x3 Sliding Window

4.3.2 Sobel Operator Module



Figure 4.7: Edge-sobel-wrapper Module

Figure 4.7 describes the edge-sobel-wrapper module. This module is responsible for obtaining synchronization signals and pixel values from the computer/camera and using the Sobel operator to calculate the gradient for each pixel.

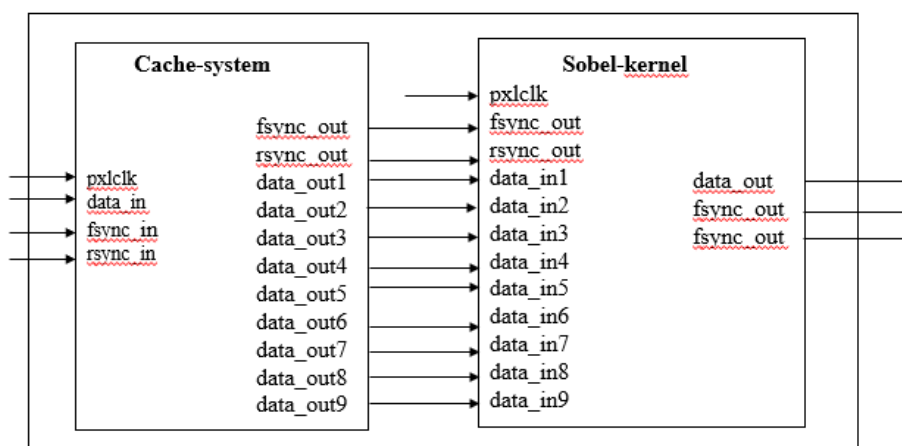


Figure 4.8: Edge_sobel_wrapper Internal Connection

The design of the edge_sobel_wrapper module is divided into two macro modules:

1. **Cache_system:** responsible for the accumulation of the pixel values of the first two rows and the delay of the synchronization signal.
2. **Sobel_kernel:** responsible for using Sobel operator to calculate gradient approximation in two directions (horizontal and vertical).

4.3.3 Cache System Module

This module obtains two synchronization signals (frame synchronization and line synchronization) and pixel value (gray 8 bits) directly from the camera (here from the computer), and outputs the value of eight pixels representing the neighborhood of the pixels are about the center of the 3x3 kernel. The module must also output the correct time synchronization signal.

When calculating pixels, it is impossible to emit the first pixel of the result image after reading the first pixel of the original image, because all the pixels near the pixel are not yet available. Similarly, to calculate the pixel value at position (0, 0) in the result image, nine pixels (position 1 from z1 to z9) are required, of which five pixels (z1, z2, z3, z4, z7) are assumed to be black (gray the value is zero). The other four pixels (z5, z6, z8, z9) are only available after reading the first two pixels of the entire first and second rows. This basically means that relative to the reading of the original image, the emission of pixels in the resulting image is in the "post" of one line and two pixels. This also means that to send

out each pixel in the resulting image, the first two lines of the original image (lines 3 and 4 at positions 4, 5, and 6) and the third line (line 5) must be read. Pixels are emitted in row 4 of the resulting image.

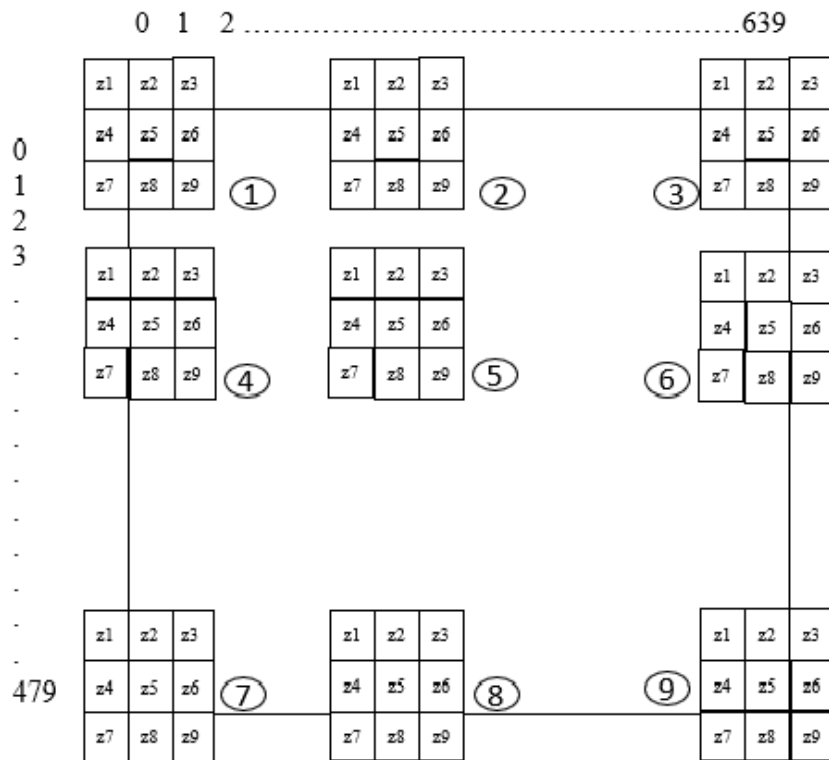


Figure 4.9: Convolution (3x3) mask with an image with numbers from 1 to 9 to the relevant positions of the mask

To solve this problem, the basic idea is to create a memory architecture that can store two rows of images. The basic structure of this memory is shown in Figure 4.10.

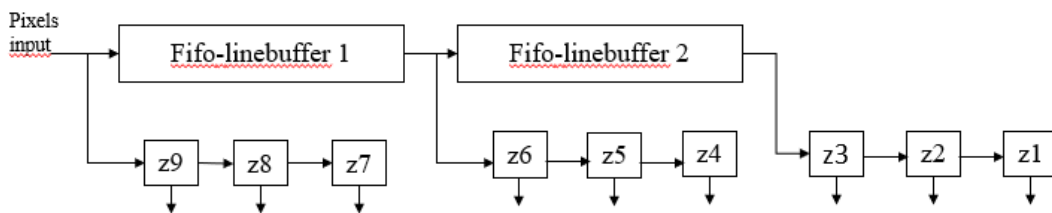


Figure 4.10: Memory Architecture

In Figure 4.10, there are two fifo line buffers responsible for accurately accumulating the number of pixels in a line (640). These buffers have a first-in, first-out strategy, which means that pixel values flow out of the buffer in the same order as the input entered, and only start to flow when the line buffer is full. The elements labeled z1 to z9 (referred to as pixel elements from now on) represent smaller copies of pixel values. These pixel values are also stored in the line buffer. The exact extraction of the

pixel values that must be used from the line buffer is required Sobel kernel. All these elements are driven by the pixel clock, which means that on each rising edge of the pixel clock, the pixel value is stored in the line buffer and passes through the small elements. For example, for position 5 (Figure 4.9), fifo-linebuffer 1 will hold line number 4, fifo-linebuffer 2 will hold line number 3, and when reading pixel z9 in row 5, the small element will contain all pixels in in. The neighborhood of z5.

Therefore, the cache-system module must delay the synchronization signal appropriately to ensure that the sobel-kernel module will know when the nine-pixel values are valid. In fact, the sobel-kernel module must wait until the cache-system module has accumulated the correct number of rows and pixels so that it can start calculating the gradient with the correct values for the 9 pixels. This synchronization can be achieved by delaying the two synchronization signals (frame synchronization and line synchronization) at an appropriate time. In this case, the cache-system module will improve the frame synchronization and line synchronization after one line and two pixels, because of the position number In Figure 4.9, only the first row is necessary because the pixels z1, z2, and z3 are assumed to be black.

The implementation of this module is divided into several parts:

- A. **Double-fifo-linebuffer:** responsible for accumulating two lines of original images.
- B. **counter:** a general-purpose counter used to count rows and pixels during reading. Variables inside the cache-system itself are used to create a pixel element that is only responsible for storing the neighborhood pixels of a given pixel.
- C. **Sync-signals-delayer:** responsible for delaying the synchronization signal.

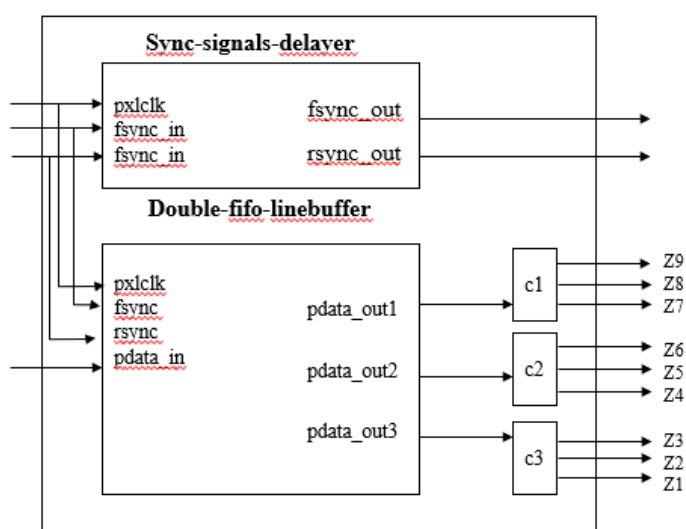


Figure 4.11: Cache-system Internal Structure

A. Fifo Linebuffer Module

The following figure describes the module in the module, which is responsible for storing the entire row (640 pixels) using the FIFO strategy.

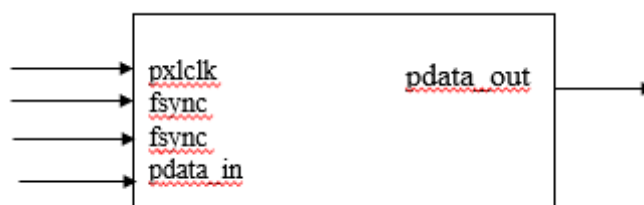


Figure 4.12: FIFO-linebuffer Module

To implement the FIFO strategy, the following principle is used: Before writing to the same unit, always read the pixel value from the memory. To achieve this behavior, the pixel clock is used and inverted (clk2). In this way, the rising edge of the pixel clock is used to read from the memory, and the rising edge of clk2 is used to write it into the memory. The same cell of the reading. The variable is used as a counter, which increments after each write, and is used to determine the unit used in the read/write phase in the memory. This method is used during the reading of the first row, since the memory is not initialized, the output of this module is not defined, but during the reading of the second row in the output, the pixels of the first row will be displayed in the same order as the first row. The value is entered in memory.

In order to obtain this result, two processes were created inside the code, one process was used for the reading phase driven by the pixel clock, and the other process was used for the writing phase driven by the inverse of the pixel clock. Frame synchronization and line synchronization signals are used as enable signals and read/write enable signals, respectively.

The following Figure 4.13 depicts the code of this module, where the image-testbench module is used to read images from file-system, and Figure 4.14 depicts the code of this module, where the rising edge is used to write to the memory.

```

38 ⊕ -- reading from the memory
39 ⊕ p : process(clk)
40 ⊕ begin
41 ⊕     if clk'event and clk='1' then
42 ⊕         if fsync = '1' then
43 ⊕             if rsync = '1' then
44 ⊕                 pdata_out <= ram_array(ColsCounter);
45 ⊕             end if;
46 ⊕         end if;
47 ⊕     end if; -- clk
48 ⊕ end process;

```

Figure 4.13: Read an image from filesystem in VHDL

```

50 ⊕ -- writing into the memory
51 ⊕ p2 : process (clk2)
52 ⊕ begin
53 ⊕     if clk2'event and clk2='1' then
54 ⊕         if fsync = '1' then
55 ⊕             if rsync = '1' then
56 ⊕                 ram_array(ColsCounter) <= pdata_in;
57 ⊕             end if;
58 ⊕         end if;
59 ⊕         if ColsCounter < 639 then
60 ⊕             ColsCounter <= ColsCounter+1;
61 ⊕         else
62 ⊕             ColsCounter <= 0;
63 ⊕         end if;
64 ⊕     else
65 ⊕         ColsCounter <= 0;
66 ⊕     end if; -- rsync
67 ⊕ end if; -- fsync
68 ⊕ end if; -- clk2
69 ⊕ end process p2;

```

Figure 4.14: Write edges into memory in VHDL

As mentioned earlier, two rows must now be stored so that all pixels are located near a given pixel. Therefore, a module named double-fifo-linebuffer is created, which combines two instances of the fifo-linebuffer module, as shown in Figure 4.10. It is also noticeable that there are three output signals: pdata-out1 is the current row, pdata-out2 is the previous row, and pdata-out3 is the previous row on pdata-out2.

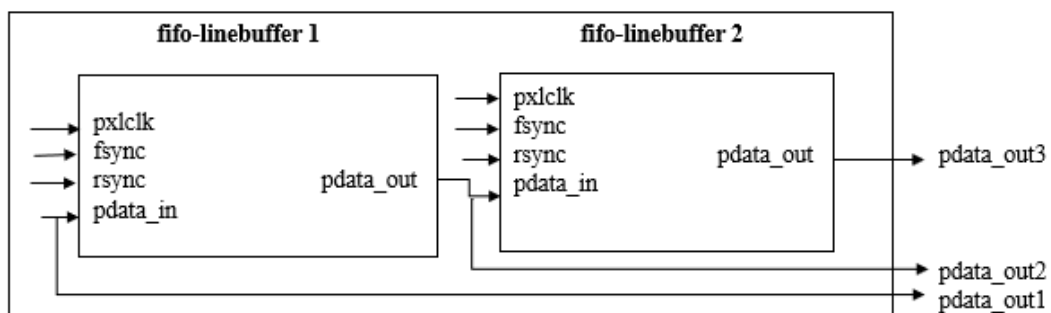


Figure 4.15: Double-fifo-linebuffer Internal Structure

Pixel Elements

This part is designed using three variables called cache1, cache2 and cache3. They are logically divided into three parts, as shown in Figure 4.11. Each of these sections is 8 pixels wide and is responsible for retaining a pixel value. The three outputs of the double-fifo-linebuffer module are "connected" with the high part of the variable (23-16), especially: pdata-out1 is connected to cache1, pdata-out2 is connected to cache2, and pdata-out3 is connected to cache3, as shown in Figure 4.15.

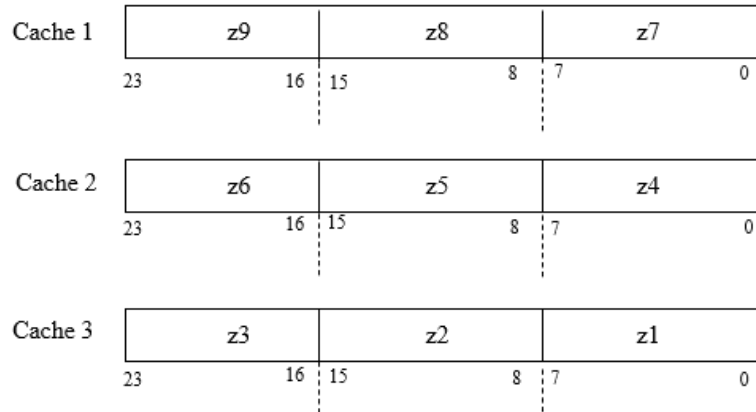


Figure 4.16: Logical Division of Pixel Elements Variables

For each rising edge of the pixel clock, the pixels inside the variable move from the higher part (23-16) to the lower part (7-0), after which they will be lost. This mechanism is an implementation of the structure shown in Figure 4.16, and Figure 4.17 depicts the code used by the module to move pixels.

```

117 ShiftingProcess : process (clk)
118     begin
119         if clk'event and clk='1' then
120             -- the pixel in the middle part is copied into the low part
121             cache1(DATA_WIDTH -1 downto 0):=cache1(((WINDOW_SIZE-1)*DATA_WIDTH - 1) downto ((WINDOW_SIZE-2)*DATA_WIDTH) );
122             cache2(DATA_WIDTH -1 downto 0):=cache2(((WINDOW_SIZE-1)*DATA_WIDTH - 1) downto ((WINDOW_SIZE-2)*DATA_WIDTH) );
123             cache3(DATA_WIDTH -1 downto 0):=cache3(((WINDOW_SIZE-1)*DATA_WIDTH - 1) downto ((WINDOW_SIZE-2)*DATA_WIDTH) );
124             -- the pixel in the high part is copied into the middle part
125             cache1(((WINDOW_SIZE-1)*DATA_WIDTH - 1) downto ((WINDOW_SIZE-2)*DATA_WIDTH) ):=cache1((WINDOW_SIZE*DATA_WIDTH) -1 downto ((WINDOW_SIZE-1)*DATA_WIDTH) );
126             cache2(((WINDOW_SIZE-1)*DATA_WIDTH - 1) downto ((WINDOW_SIZE-2)*DATA_WIDTH) ):=cache2((WINDOW_SIZE*DATA_WIDTH) -1 downto ((WINDOW_SIZE-1)*DATA_WIDTH) );
127             cache3(((WINDOW_SIZE-1)*DATA_WIDTH - 1) downto ((WINDOW_SIZE-2)*DATA_WIDTH) ):=cache3((WINDOW_SIZE*DATA_WIDTH) -1 downto ((WINDOW_SIZE-1)*DATA_WIDTH) );
128             -- the output of the ram is put in the high part of the variable
129             cache1((WINDOW_SIZE*DATA_WIDTH) -1 downto ((WINDOW_SIZE-1)*DATA_WIDTH) ):=dout1;
130             cache2((WINDOW_SIZE*DATA_WIDTH) -1 downto ((WINDOW_SIZE-1)*DATA_WIDTH) ):=dout2;
131             cache3((WINDOW_SIZE*DATA_WIDTH) -1 downto ((WINDOW_SIZE-1)*DATA_WIDTH) ):=dout3;
132         end if; -- clk
133     end process ShiftingProcess;

```

Figure 4.17: Sliding Window Process in VHDL

B. Rows and Columns Counter

As mentioned earlier, the pixels outside the image boundary are assumed to be black, so it is important to detect the various positions where the kernel will stay, as shown in Figure 4.18, because the module

must emit black pixels for positions close to the boundary. The easiest way to detect these positions is to count the rows and columns within the display time. Therefore, a general-purpose counter is designed with a low-level active reset function and then instantiated twice: a counter is used to perform the row Count, and another counter is used to count the rows. Used to count columns (pixels in a row).

The counter of the row must count the rising edge of the rsync signal, so the counter uses the rsync signal as a clock. When the signal fsync goes low, this counter will be reset, because it means that a frame is displayed and a new frame will be started, so the line counter should restart from 0. Instead, the column counter must count the rising edges of the pixel clock only when the rsync signal is high, because this means that the monitor is displaying a specific row, so this counter is used as a clock, pixel clock and rsync is enabled signal. When the signal rsync goes low, the column counter will be reset, because this means that one row is displayed and a new row will start, so the column counter should restart from 0.

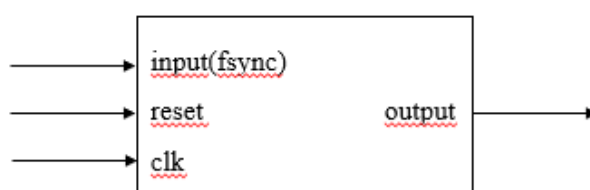


Figure 4.18: Rows Counter Connections

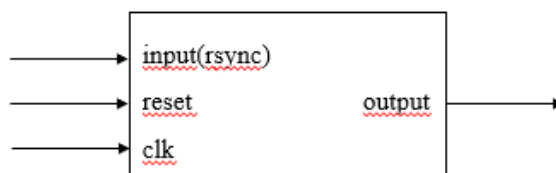


Figure 4.19: Columns Counter Connections

The output of these two counters is used internally in the process to understand the position of the Sobel core so that the correct pixel value obtained from the three cache variables can be issued, or a constant value of zero can be used for black.

```

15 architecture Behavioral of Counter is
16     signal num : STD_LOGIC_VECTOR(n-1 downto 0);
17
18 begin
19     process (clk, reset)
20
21     begin
22         if(reset = '0') then
23             num <= (others => '0');
24         elsif(clk'event and clk = '1') then
25             if (en = '1') then
26                 num <= num + 1;    -- When num goes in overflow it will change automatically
27             end if;
28         end if;
29     end process;
30
31     output <= num;
32
33 end Behavioral;

```

Figure 4.20: Counting row and columns in VHDL

C. Sync-signals-delayer

This module is responsible for delaying the synchronization signal of one row and two pixels to ensure that the memory is filled with appropriate pixel values. The design is divided into five stages:

1. Delay the line synchronization signal of two pixels (two clock cycles), this signal is called `rsync2`.
2. Use an example of the previous counter to count the rising edges of the signal `rsync2`. The output of the counter will be put into a signal named `rowsDelayCounterRising`;
3. When the `rowsDelayCounterRising` signal takes the value 2, the output frame synchronization signal will become high.
4. Calculate the falling edge of the signal `rsync2` and put the count value into the signal `rowsDelayCounterFalling`.
5. When the signal `rowsDelayCounterFalling` value is 0, the output frame synchronization signal will go low.

4.3.4 Sobel Kernel Module

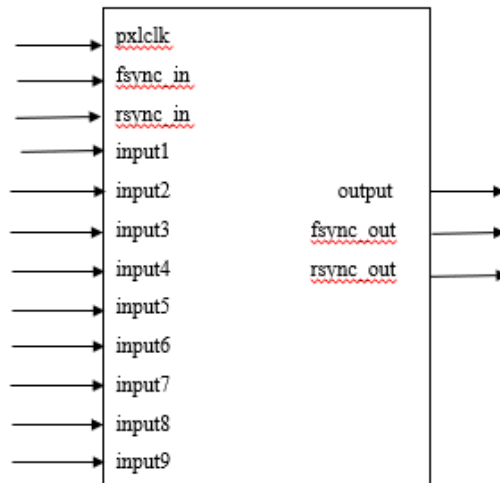


Figure 4.21: Sobel_kernel Module

The next step is to use the Sobel operator to design a module responsible for calculating the approximate value of the gradient in each pixel of the image. The module is implemented using the following process: Use the Sobel kernel to store the horizontal and vertical amplitudes into two variables (if the number of results obtained is set to a negative number, then the coefficient 2 in the mask is moved to the left of a position of the pixel value). It is converted to a positive number through the complement operation (not a number + 1), and the 11th bit is used to know whether the number is negative.

```

39 ;      begin
40 ⊕      if (pclk_i'event and pclk_i = '1') then
41 :
42 :      rsync_o <= rsync_i;
43 :      fsvnc_o <= fsvnc_i;
44 :
45 ⊕      if fsvnc_i = '1' then
46 ⊕          if rsvnc_i = '1' then
47 :
48 :              -- x2
49 :              summax:=( "000" & pdata3)+("00" & pdata6 & '0')+( "000" & pdata9)
50 :                  -("000" & pdata1)-("00" & pdata4 & '0')-( "000" & pdata7);
51 :
52 :              -- x2
53 :              summay:=( "000" & pdata7)+("00" & pdata8 & '0')+( "000" & pdata9)
54 :                  -("000" & pdata1)-("00" & pdata2 & '0')-( "000" & pdata3);
55 ⊕
56 :              -- Here is computed the absolute value of the numbers
57 :              if summax(10)='1' then
58 :                  summa1:= not summax+1;
59 :              else
60 :                  summa1:= summax;
61 :              end if;
62 :
63 :              if summay(10)='1' then
64 :                  summa2:= not summay+1;
65 :              else
66 :                  summa2:= summay;
67 :              end if;
68 :
69 :              summa:=summa1+summa2;

```

Figure 4.22: Sobel Kernel in VHDL

Then, combine the two magnitudes, as shown in Figure 4.22. Finally, apply a threshold of 127, because it is the global thresholding value for 8-bit image. If the value is greater than the threshold, the output will be white (255), otherwise, only the last 8 bits are taken and placed in the output.

```

69 |
70 | □
71 | □
72 | □
73 | □
74 | □

```

```

-- Threshold = 127
if summa>"000011111111" then
  pdata_o<=(others => '1');
else
  pdata_o<=summa(DATA_WIDTH- 1 downto 0);
end if;

```

Figure 4.23: Using threshold value in VHDL

4.4 Python Implementation

The Python (version 3.7.8) programming language will also be used for edge detection. To complete the task, the Visual Studio 2019 editor will be used. Python provides many good libraries to process images, such as open-cv, Pillow, etc. In this project, OpenCV (version 4.4.0) will be used, which is an open source library for computer vision. OpenCV has introduced a new set of tutorials that will guide us through the various functions available in OpenCV-Python. To do this, first, it must be ensured that Numpy is running in python, and then OpenCV has to be installed. Numpy is a highly optimized library for number operations [20][21]. It provides a MATLAB style grammar. Figure 4.24 contains the following libraries.

```

1   import cv2
2   import numpy as np

```

Figure 4.24: Libraries used in Python

All OpenCV array structures are converted to and from Numpy arrays. Therefore, no matter what it can perform in Numpy, it can be used in conjunction with OpenCV. In addition, other libraries that support Numpy (such as SciPy, Matplotlib) can also be used with this. Therefore, OpenCV-Python is a suitable tool for rapid prototyping of computer vision problems for image processing [22][23].

4.4.1 (3 x 3) Matrix Kernel Convolution Method

Sobel edge detector is a gradient-based method based on the first derivative. It calculates the first derivative of the image for the X and Y axis, respectively. The operator uses two 3X3 kernels convolved with the original image to calculate the approximation of the derivative-one for the horizontal change and one for the vertical change. Figure 4.25 describes edge detection using Python.

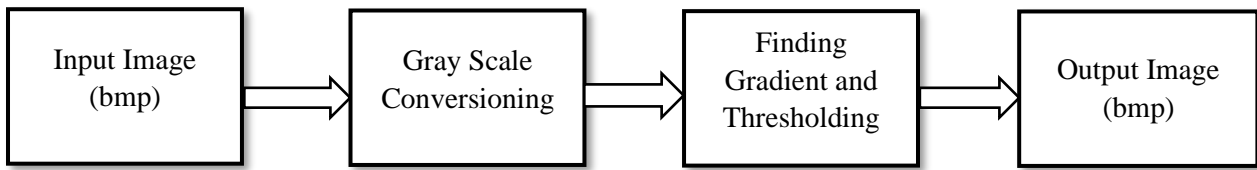


Figure 4.25: Structural block diagram of Sobel Edge Detection using Python

At first, `cv2.imread()` function has to be used which loads an image from the specified file. It specifies the way in which the image should be read. Its default value is `cv2.IMREAD_COLOR`. The syntax is `cv2.imread(path, flag)`. If the image cannot be read (because of the missing file, improper permissions, unsupported or invalid format) then this function returns an empty matrix. It is the default flag. Alternatively, the integer value 1 can be passed for this flag. Then the color image must be converted into grayscale. There are more than 150 color-space conversion methods available in OpenCV. But only two which are the most widely used ones, `BGR ↔ Gray` and `BGR ↔ HSV` have to be concerned. For color conversion, we use function `cv2.cvtColor(input_image, flag)` where flag determines the type of conversion. For BGR to Gray conversion, `cv2.COLOR_BGR2GRAY` flag has to be used.

```

4     #read image from memory#
5     img1= cv2.imread('D:/python/color_test.bmp',1)
6     #color image to gray scale conversion#
7     gray= cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
8     #Gaussian smoothing#
9     img= cv2.GaussianBlur(gray, (3,3),0,0,cv2.BORDER_DEFAULT)
--
  
```

Figure 4.26: Image read and gray scale conversion in Python

After that, the `GaussianBlur()` function have to be applied. OpenCV provides `cv2.GaussianBlur()` function to apply Gaussian Smoothing on the input source image. For Gaussian smoothing it should be specified the width and height of the kernel size, her (3 x 3) matrices has to be used for the x and y-axis, respectively. It should also be specified the standard deviation in the X and Y directions, sigma X and sigma Y respectively. If only sigma X is specified, sigma Y is taken as equal to sigma X. If both are given as zeros, they are calculated from the kernel size. Gaussian filtering is highly effective in removing Gaussian noise from the image.

```

12    #Gx,Gy and final gradient calculation#
13    grad_x= cv2.Sobel(img, cv2.CV_16S, 1, 0, ksize=3, scale=1, delta=0)
14    abs_grad_x= cv2.convertScaleAbs(grad_x)
15    grad_y= cv2.Sobel(img, cv2.CV_16S,0,1,ksize=3,scale=1,delta=0)
16    abs_grad_y= cv2.convertScaleAbs(grad_y)
17    grad= cv2.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
18
  
```

Figure 4.27: Gradient Calculation in Python

It can be iterated through all pixels in the original image and apply the image convolution kernels to calculate gradient magnitude for x and y derivatives, G_x and G_y respectively. This convolution operation will return a single integer for each kernel. With the function `cv2.addWeighted()`, the weighted sum of the kernels can be calculated which will create a new image of the same dimensions as the original image and store for the pixel data. It would also be converted the magnitude of each direction into 8 bits using `cv2.convertScaleAbs()` function. Next, the threshold value is required which is used to classify the pixel values. OpenCV provides different styles of thresholding. Optimal global threshold binarization and morphological close transformation are used to improve the result. Then to decide which pixels are edges, the magnitude of the gradient to be compared to the threshold value.

```
20     #thresholding and edge calculation#
21     threshold= cv2.threshold(grad, 0, 255, cv2.THRESH_OTSU + cv2.THRESH_BINARY)
22     threshold= cv2.morphologyEx(grad, cv2.MORPH_CLOSE, (3,3))
```

Figure 4.28: Thresholding and edge calculation in Python

Experimental Result

5.1 VHDL Test Bench Simulation

In this section, the edge-sobel-wrapper module is tested using img-testbench and the different test images reported below. The first Image-1 (Color-test) has a white background, only three colors (red, green, blue), and its pixel line width is 1 pixel: select a white background to test the module's response along the border and color of the image. It is desirable to obtain an image with a black background and a white border, which depends on the fact that pixels located outside the border of the image are assumed to be black.



Figure 5.1 (a): Image 1-Color test

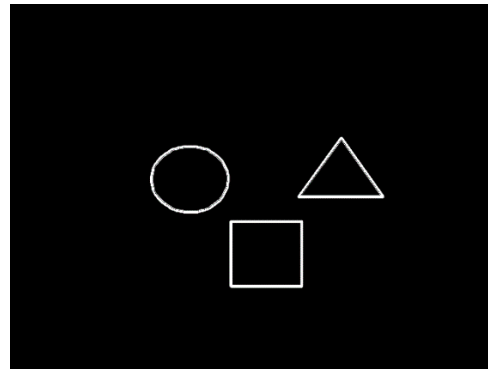


Figure 5.1 (b): Edges of Image-1

On the contrary, in the second Image-2 (Image-Car), the border of the generated image will have a different color because the background of the image is not perfect black or white, but there are many pixels with different values (such as noise).



Figure 5.2 (a): Image 2-Car

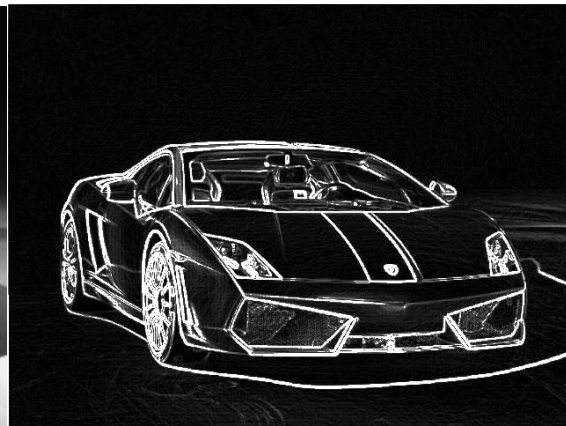


Figure 5.2 (b): Edges of Image-2

The following two figures depict the signals of the edge-sobel-wrapper during the simulation. From Figure 5.3 it is possible to notice that the entire first line is white (pdata-in) and the output is black (pdata-out) were from Figure 5.4 it is possible to see that each line starts and ends with different pixels (pdata-in).

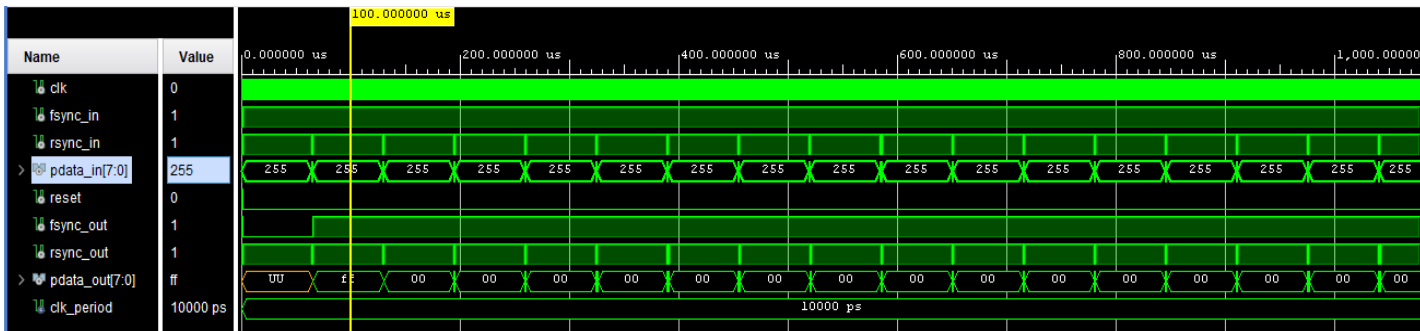


Figure 5.3: Simulation of Image 1

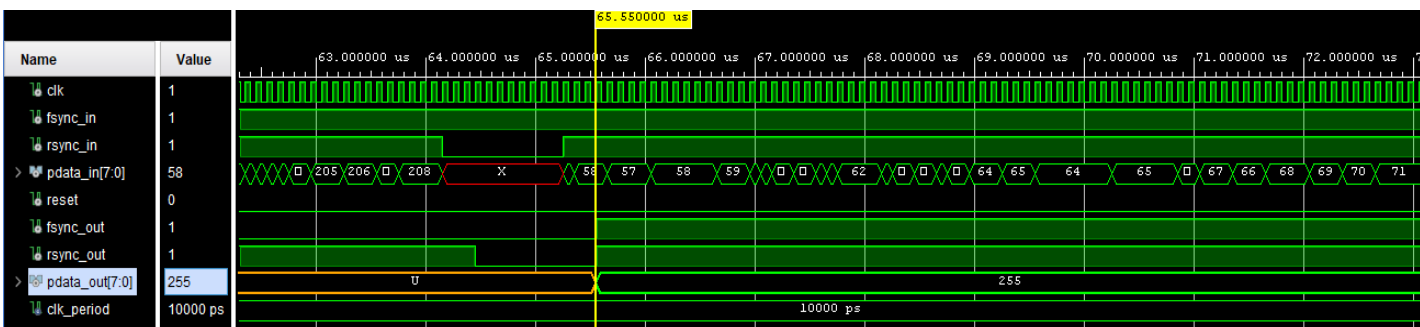


Figure 5.4: Simulation of Image 2

The figure above reports the simulation of the module: when the input signal changes, the output signal will also change. In this example, when all inputs are zero, the output is zero (if the input is 255, the output is the same). Conversely, when some pixels are black and some pixels are white, the output result is white because the magnitude value is higher than the threshold.

Here the unit under test (UUT) module is synthesizable so it can be used in hardware applications that emphasize the edge. Therefore, we can implement this module in FPGA to analyze the performance of real-time Sobel edge detection technology.

5.2 Simulation with Python

In this section, the results of Sobel edge detection using Python in Visual Studio Code 2019 are shown below. The final outputs are as like as the Vivado HLS for the same images. The hardware-defined Language module is working fine as the software module. Also, the noise has been removed which will help to find an efficient edge detected image also in Python.

By using Python code, we will obtain three edges (x-axis, y-axis, and weighted sum) for each image, so we can correctly identify how the final edge is constructed. Here, the weighted sum or final edge is the same as Vivado. Although the definition of edges using VHDL is better than that using Python, Visual Studio Code can detect the edges of each image as accurately as Vivado HLS.

Results of the edges for Image-1 (Color test) and Image-2 (Car) are shown in below:

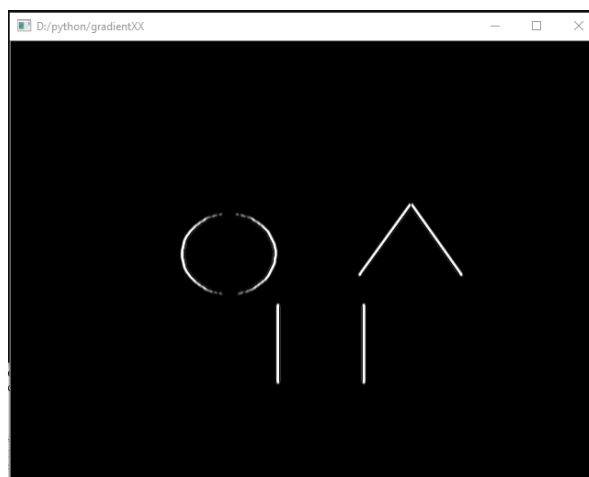


Figure 5.5(a): Image 1-Color test

Figure 5.5 (b): Edges of Image-1 (x-axis)

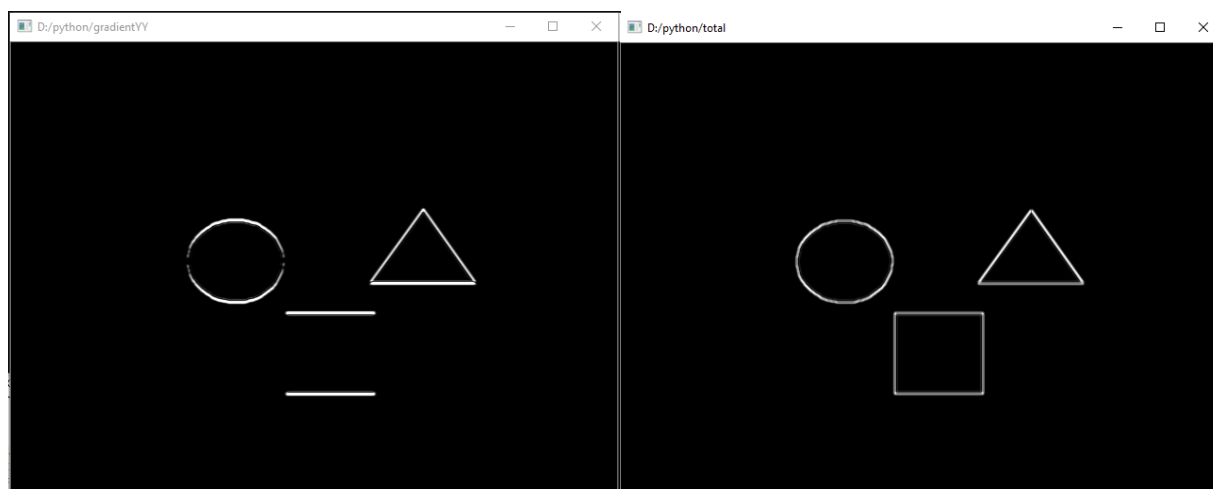


Figure 5.5 (c): Edges of Image-1 (y-axis)

Figure 5.5 (d): Edges of Image-1 (weighted sum)

In this section, Figure 5.5(b), Figure 5.5 (c), Figure 5.5 (d) illustrate the result of simulation that calculates the edges of gray scale images respectively in vertical, horizontal, and weighted sum of these two diagonals for Image-1 (Color-test) which has a white background, only three colors (red, green, blue), and its pixel line width is 1 pixel: select a white background to test the module's response along the border and color of the image.



Figure 5.6 (a): Image 2-Car

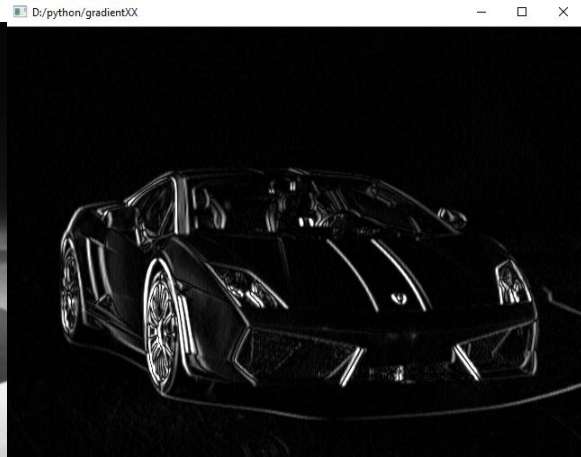


Figure 5.6 (b): Edges of Image 2 (x-axis)

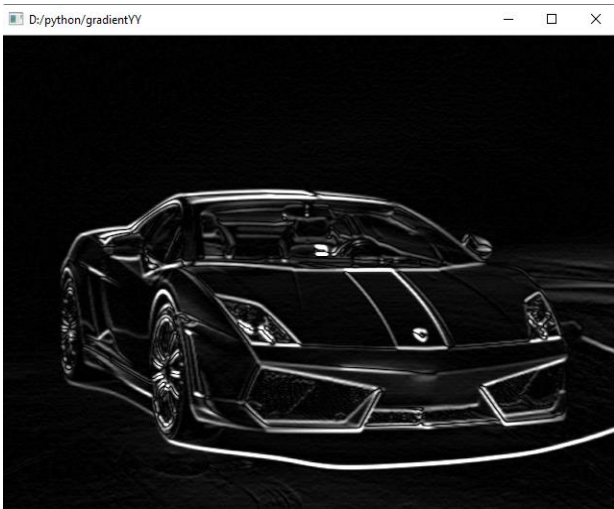


Figure 5.6 (c): Edges of Image 2 (y-axis)

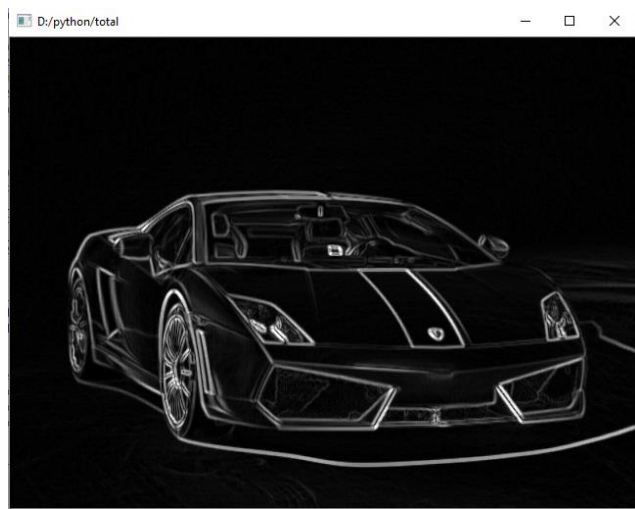


Figure 5.6 (d): Edges of Image 2 (weighted sum)

In the second Image-2 (Image-Car), the border of the generated image will have a different color because the background of the image is not total black or white, but there are many pixels with different values (such as noise). Here we can see the result of simulation in Figure 5.6(b), Figure 5.6 (c), Figure 5.6 (d) respectively the vertical, horizontal, and weighted sum of these two diagonals for Image-2 (Image-Car).

CHAPTER-6

Conclusion

6.1 Conclusion

After the analysis and observations in Chapters 4 and 5, the goal has been successfully achieved and the project has been successfully completed to detect the edges of high-resolution color images. This preliminary study provides insights on how VHDL and Python work with the Sobel Edge Detection algorithm. The performance results are encouraging, and we are currently evaluating other application benchmarks in various scientific computing fields. As the Unit Under Test (UUT) module of VHDL can be used in hardware applications that emphasize the edge where the remove from the camera to the question is significant. Therefore, we can implement this module in FPGA to better understand the performance of real-time Sobel edge detection technology.

In addition, the modular structure of the VHDL permits to reuse the components in other ventures, for illustration the range-sensor module or the Binary-coded decimal converter module can be utilized in numerous diverse ventures without issues since these modules are designed to be free shape the full extend. Same thinking applies to the number-displayer module that can be effortlessly adjusted in arrange to show more numbers or different information.

6.2 Future Work

Although the expected results were achieved, there are still several possible areas for improvement. For example, through the realization of the digital display module, the numbers in the first pixel of each line on the display frame can be printed. The important question that should be explored is the possibility of making the design so universal so that the module can automatically adapt to different mask sizes, for example, if a 5x5 mask is required. Therefore, it may be a good idea to spend some time in the future to promote the system to eliminate approximations in the measurement, while at the same time being able to design quickly and easily. The test platform can also be improved to read/write images from the file system for VGA signals.

In addition, Python offers profoundly optimized libraries from an unimaginably huge designer community, however, is constrained to the execution of the equipment framework. To use Python directly to the hardware, Xilinx recently released PYNQ, which aims to support software developers

who use Python instead of VHDL to access FPGAs. The PYNQ application improvement system is an open-source exertion outlined to permit application engineers to attain a “fast start” in FPGA application advancement through utilize of the Python dialect and standard “overlay” bitstreams that are utilized to connected with the chip's I/O gadgets. The PYNQ environment comes with a standard overlay that underpins HDMI and Sound inputs and yields, as well as two 12-pin PMOD connectors and an Arduino-compatible connector that can associated with Arduino shields.

PYNQ moreover offers an API and expands common Python libraries and bundles to incorporate back for Bitstream programming, straightforwardly get to the programmable texture through Memory-Mapped I/O (MMIO) and Coordinate Memory Get to (DMA) exchanges without requiring the creation of gadget drivers and part modules. Therefore, studying the combination of Python and FPGA may be another step.

References

- [1] J. F. Canny, "A computational approach to edge detection", *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 8, no. 6, pp. 679-698, Nov. 1986.
- [2] H. Jiang, H. Ardo, and V. Owall, "A Hardware Architecture for Real-Time Video Segmentation Utilizing Memory Reduction Techniques", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 2, pp. 226–236, 2009.
- [3] C. L. Novak and S. A. Shafer, "Color edge detection," in *Proc. DARPA Image Understanding Workshop*, pp. 35–37, 1987.
- [4] Dibya Jyoti Bora, "A Novel Approach for Color Image Edge Detection Using Multidirectional Sobel Filter on HSV Color Space", *International Journal of Computer Sciences and Engineering*, vol. 5, no. 2, pp. 2347-2693, Feb. 2017.
- [5] Bovik, 2009. "The Essential Guide to Image Processing", Academic Press, Elsevier Inc.
- [6] Mohamed Nasir Bin Mohamed Shukor, Lo HaiHiung, Patrick Sebastian, "Implementation of Real-time Simple Edge Detection on FPGA", *IEEE*, pp. 1404-1405, 2007.
- [7] Rashmi, Mukesh Kumar, and Rohini Saxena, "Algorithm and technique on various edge detection: a survey", in *Signal & Image Processing: An International Journal (SIPIJ)*, vol.4, no.3, June 2013.
- [8] Muthukrishnan.R, "Edge detection techniques for image segmentation", in *International Journal of Computer Science & Information Technology (IJCSIT)*, vol 3, no 6, Dec 2011.
- [9] Wikipedia link: https://en.wikipedia.org/wiki/Sobel_operator
- [10] Nasseer M. Basheer, Ashty M. Aaref, Dhafer J. Ayyed, "Digital Image Sobel Edge Detection Using FPGA", in *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 5, no. 7, pp. 183-190, July 2015.
- [11] Bhojar, S. D., Pathan, T., Landge, A. D., "Design and Implementation of Sobel Edge Detection technique using VHDL" *International Journal of Advanced Research in Science, Engineering and Technology*, vol. 3, Issue 5, May 2016.
- [12] Ankush R. Bhagat, Swati R. Dixit, Dr. A.Y. Deshmukh, "VHDL based Sobel Edge Detection", in *International Journal of Engineering Research and General Science*, vol 3, no. 1, pp. 1217-1223, Jan 2015.
- [13] Chuanwei Zhang, Zhengyang Yu, "Translation of Image Edge Detection Based on Python", in *IOP Conf. Series: Earth and Environmental Science*, 2019.
- [14] Andrew G. Schmidt, Gabriel Weisz, and Matthew French, "Evaluating Rapid Application Development with Python for Heterogeneous Processor-based FPGAs", *IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*, 2017.
- [15] Dimitris Tsiktiris, Dimitris Ziouzos, Minas Dasygenis, "A High-Level Synthesis Implementation and Evaluation of an Image Processing Accelerator", in *7th International Conference on Modern Circuit and System Technologies on Electronics and Communications (MOCAST 2018)*, Thessaloniki, Greece, 7–9 May 2018.

- [16] Punam Mahesh Ingale, "The importance of Digital Image Processing and its applications", in International Journal of Scientific Research in Computer Science and Engineering, vol.6, no.1, pp.31-32, Jan 2018.
- [17] Chao-Chao Zhang, Jian-Dong Fang, "Edge Detection Based on Improved Sobel Operator", International Conference on Computer Engineering and Information Systems (CEIS-16), Advances in Computer Science Research, (ACSR), vol. 52, pp. 129-132, 2016.
- [18] James Clerk Maxwell, "Digital Image Processing Mathematical and Computational Methods", Horwood Publishing, 2005.
- [19] Sunanda Gupta, Charu Gupta and S.K. Chakarvarti, "Image Edge Detection: A Review", International Journal of Advanced Research in Computer Engineering & Technology (IJARCET), vol. 2, pp. 2278-1323, July 2013.
- [20] Truls Asheim, Kenneth Skovhede, "VHDLgeneration from Python synchronous message exchange networks", Communicating Process Architectures, vol. 38, pp. 137-158, 2016.
- [21] Joseph Howse. "Open CV Computer Vision with Pytho", Birmingham: Packt Publishing, 2013.
- [22] Python edge detection: <https://www.geeksforgeeks.org/python-program-to-detect-the-edges-of-an-image-using-opencv-sobel-edge-detection/>
- [23] Sobel edge detection and python:
https://www.bogotobogo.com/python/OpenCV_Python/python_opencv3_Image_Gradient_Sobel_Laplacian_Derivatives_Edge_Detection.php
- [24] Sysnthesis and Simulation Design Guide:
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/sim.pdf
- [25] XST User Guide: <http://www.xilinx.com/itp/xilinx10/books/docs/xst/xst.pdf>
- [26] Raman Maini, Dr. Aggarwal, "Study and Comparison of Various Image Edge Detection Techniques", International Journal of Image Processing (IJIP), vol. 3, no. 1, pp. 1-11, 2009.
- [27] E&CE 427 2005t3 (Fall) Project: Sobel Edge Detector.
- [28] Chandresh Pratap Singh, "R-tree implementation of image databases", Signal & Image Processing: An International Journal (SIPIJ) vol.2, no.4, Dec. 2011.
- [29] Vallapu Radha, P. Chandra Sekhar, "Comparision of Robert, Pewit, Sobel Operators Based Edge Detection Methods", International Journal of VLSI system design and communication system, vol.4, no.9, pp. 0882-0885, Sep. 2016.
- [30] Rahul R.Gaulkar , Swati R. Dixit and A.Y.Deshmukh, "Design of VHDL based Multi-Directional Sobel Edge Detection Processor", International Journal of Current Engineering and Technology, vol.4, no.2, pp. 745-748, april 2014.
- [31] Jayalaxmi H and S. Ramachandran, "Design of Sliding Window Based Corner Detection Algorithm and Architecture for Image Mosaicing", International Journal of Signal Processing, Image Processing and Pattern Recognition, vol.8, no.3, pp.235-250, 2015.