



# Digital Design, Verification and Functional Safety of Lane Detection Algorithm

A safety Critical Automotive Application

*Team Members:*

**Abdulrahman Mohamed El-Badawy**

**Amr Mahmoud Ahmed Mohamed**

**Hazem Ayman AbdelFattah**

**Mohamed Amr Farouk Zahran**

**Mohanad Mohamed Ali**

**Nada Mohamed Ahmed**

**Youssef Ibrahim AbdelHameed**

*Under Supervision of:*

**Dr. Hassan Mostafa**

Communication and Electronics Department,

Faculty of Engineering, Cairo University

Class of 2022

## Acknowledgment

First and foremost, we would like to thank our supervisor, Dr. Hassan Mostafa, for his assistance and dedicated involvement in every step throughout the process providing a continuous full and professional support.

In addition, we would also like to show gratitude to SIEMENS EDA (Mentor Graphics) Company for their cooperation and support and for providing us with the all the tools needed to accomplish this work.

Finally, we would also like to thank Eng. Islam Ahmed, Eng. Mohamed Ayman, and Eng. Ahmed Khalil, who were always there for any questions and provided us with insightful comments and immense knowledge.

## Table of Contents

Acknowledgment .....	1
Abstract.....	8
Chapter 01: Introduction.....	9
1.1. Organization of the thesis: .....	9
1.2. Motivation: .....	9
1.3. What is Lane Detection: .....	10
1.4. Lane Detection Techniques: .....	10
1.5. Functional safety.....	10
Chapter 02: Hough Transform Related Work and Algorithm .....	11
2.1. The idea of the Hough Transform .....	11
2.1.1. Representation of lines in Hough Space (HS):.....	11
2.1.2. Mapping of Points to Hough Transform:.....	11
2.1.3. Algorithm:.....	12
2.1.4. Transformation of lines to Hough Space:.....	12
2.1.5. The Hough Space Accumulator:.....	12
2.1.6 Detection of Infinite Lines:.....	13
2.1.7. Finite Lines: .....	13
2.2. Proposed Hardware Architecture:.....	13
Summary:.....	15
Chapter 03: Edge Detection and Cordic Algorithm .....	16
3.1. Edge Detection.....	16
3.1.1. Proposed Algorithm.....	16
3.1.2. Sobel Filter.....	16
3.1.3. Gauss Filter.....	18
3.2. Co-Ordinate Rotational Digital Computer.....	18
Chapter 04: Software Model.....	20
4.1. Edge Detection.....	20
4.1.1. Gauss Filter.....	20
4.1.2. Sobel Filter.....	21
4.2. Hough Transform.....	23
4.2.1. Region of Interest (ROI).....	23
4.2.2. Transformation .....	23

4.2.3. Voting .....	24
4.2.3. Inverse Hough Transform and Line Drawing.....	24
Chapter 05: Hardware Implementation .....	25
5.1. Edge Detection Block .....	25
5.1.1. Image ROM & Address Generator .....	25
5.1.2. Line Buffers .....	25
5.1.3. Line Buffers Control Unit.....	26
5.1.4. Gaussian Filter .....	26
5.1.5. Modified Sobel Operator .....	27
5.1.6. Position Counters.....	28
5.2. Theta Detection Module .....	29
5.2.1. Quadrant Detector.....	29
5.2.2. CORDIC .....	29
5.2.3. The Comparator .....	30
5.3. Hough Transform.....	31
5.3.1. FIFO.....	31
5.3.2. Controller.....	32
5.3.3. LUT ( $\cot(\theta)$ ).....	32
5.3.4. Fixed-Point Multiplier .....	33
5.3.5. Address Flattener .....	34
5.3.6. Accumulator .....	34
5.3.7. Maximum Detector.....	35
5.3.8. Line Drawer .....	35
Chapter 06: Formal Verification.....	36
6.1. Questa Autocheck .....	36
6.2. Questa Lint.....	37
6.3. Questa Propcheck .....	38
6.3.1. Edge Detection Properties .....	38
6.3.2. FIFO Properties .....	38
6.3.3. Hough Transform Unit Properties .....	39
6.3.4. Line Drawer Properties.....	39
6.3.5. Propcheck Results.....	40
Chapter 07: UVM .....	41

7.1. UVM environment.....	41
7.1.1. TESTER.....	41
7.1.2. SCOREBOARD .....	41
7.1.3. COVERAGE.....	41
7.1.4. BUS FUNCTIONAL MODULE (BFM).....	42
7.1.5. DESIGN UNDER TEST (DUT) .....	42
7.2. Testing process .....	42
7.3. Internal Modules Testing.....	42
7.3.1. Edge Detection Modules.....	42
7.3.2. Theta Detection Modules.....	42
7.4. Main Blocks Testing.....	43
7.4.1. Edge Detection Block.....	43
7.4.2. Theta Detection Block .....	43
7.4.3. Hough transform (HT) Block .....	44
7.5. Test the fully integrated system.....	46
Chapter 08: FPGA Implementation and Results .....	47
8.1. Edge Detection.....	47
8.1.1. Edge detection utilization and frequency before adding ROM .....	47
8.1.2. Edge detection utilization and frequency after adding the ROM .....	47
8.2. Theta Detection Module .....	48
8.3. Hough Transform.....	49
8.4. Lane Detection Reports: .....	49
8.5. Experimental Work and FPGA Results:.....	50
8.6. Error Calculations .....	51
Chapter 09: Functional Safety .....	55
9.1. Introduction.....	55
9.2. Functional Safety Tools.....	56
9.2.1. SafetyScope .....	56
9.2.2. Annealer.....	57
9.2.3. RadioScope.....	57
9.2.4. KaleidoScope.....	57
9.3. Results.....	57
Conclusion .....	59

References.....	60
-----------------	----

## List of Figures

Figure 1: NHTSA Data.....	9
Figure 2: Mapping of a line to the Hough space .....	11
Figure 3: Transformation of a single point (po) to a line in the Hough space.....	11
Figure 4: Transformation of two points to two lines in the Hough space. ....	12
Figure 5: Representation of multiple edges on the lanes in Hough Space .....	13
Figure 6: Represents Hardware Architecture for Conventional HT .....	14
Figure 7: Definition of the two regions of interest (ROIs) for left and right lane boundaries.....	15
Figure 8: Proposed Architecture for Straight Lane Detection .....	15
Figure 9: Gradient vector at some pixel .....	16
Figure 10: Sobel Operator Filters .....	17
Figure 11: (a) Original Image, (b) Sobel Operator Output.....	17
Figure 12: Gauss Kernel. ....	18
Figure 13: (a) Original Image, (b) Gauss Operator Output. ....	18
Figure 14: Illustrative Diagram for region of interest of theta .....	19
Figure 15: Rotating input vector Axis .....	19
Figure 16: Convolution Between filter and matrix.....	20
Figure 17: Image before and after the Gaussian Filter. ....	20
Figure 18: Edge detection with a small, moderate, and large threshold values.....	21
Figure 19: Edge detection output with and without Gauss filter .....	22
Figure 20: Final output of the edge detection block.....	22
Figure 21: The Region of Interest.....	23
Figure 22: Voting Matrix .....	24
Figure 23: Final Software Output.....	24
Figure 24: Edge Detection Block Diagram .....	25
Figure 25: Line Buffer Block Diagram .....	26
Figure 26: Gaussian Filter Block Diagram.....	27
Figure 27: Sobel Filter Block Diagram .....	28
Figure 28: Theta Detection Module Block Diagram .....	29
Figure 29: Normalized values of tan(angles) used .....	30
Figure 30: Hough Transform Stage Block Diagram.....	31

Figure 31: ROI, Range of b and  $\theta$ ..... 32

Figure 32: Output before and after increasing accuracy..... 32

Figure 33: Inverse Hough Transform Block Diagram..... 33

Figure 34: Address Flattener illustration ..... 34

Figure 35: The absolute difference between software and hardware outputs before formal verification ..... 36

Figure 36: The absolute difference between software and hardware outputs after formal verification ..... 37

Figure 37: Quality score before fixing issues given by Lint tool ..... 37

Figure 38: Quality score after fixing issues given by Lint tool ..... 38

Figure 39: UVM Structure..... 41

Figure 40: Sobel filter coverage ..... 42

Figure 41: Gauss filter coverage..... 42

Figure 42: Cordic coverage ..... 43

Figure 43: average error of Cordic ..... 43

Figure 44: Sample of generated maps ..... 43

Figure 45: Theta detection coverage 1..... 44

Figure 46: Correct cases of theta detection..... 44

Figure 47: Input random edges ..... 44

Figure 48: The result of random edges example ..... 45

Figure 49: Output Results with Different Seeds..... 45

Figure 50: Hough Transform Coverage..... 45

Figure 51 : Edge Detection utilization before adding the ROM and address generator..... 47

Figure 52 : Edge Detection max frequency before adding the ROM and address generator ..... 47

Figure 53 : Edge Detection timing constrains before adding the ROM and address generator ..... 47

Figure 54: Edge Detection utilization after adding the ROM and address generator..... 48

Figure 55 : Edge Detection max frequency after adding the ROM and address generator ..... 48

Figure 56 : Edge Detection timing constrains after adding the ROM and address generator ..... 48

Figure 57: Theta Detection Module Utilization..... 48

Figure 58: Maximum frequency of theta detection module block ..... 48

Figure 59: Hough Transform Utilization..... 49

Figure 60: Hough Transform Timing Report ..... 49

Figure 61:Hough Transform Timing Constrains ..... 49

Figure 62:Total Lane Detection Utilization..... 49

Figure 63: Lane Detection Timing Report..... 49

Figure 64: Lane Detection Timing Constrains ..... 50

Figure 65: Shows the Chosen Region of Interest (ROI)..... 50

Figure 66: Gauss\_CU module after applying duplication safety mechanism using Annealer ..... 58

**List of Tables**

Table 1: Theta values for random inputs and different iterations..... 30

Table 2: The average error of hardware model ..... 51

Table 3: Lane detection results in various conditions ..... 52

Table 4: The ASIL levels corresponding to certain hazardous events ..... 55

Table 5: Full Utilization of the 3 main blocks before and after applying safety mechanisms ..... 58

Table 6: Max frequency (with 10% slack) and dynamic power before and after applying safety mechanisms 58



## Abstract

Autonomous vehicles are the next big step in the transportation sector due to their efficiency and safety benefits. A self-driven car needs to understand its surroundings so it can navigate its way in the streets with minimal human assistance. Driver and passenger safety systems are one of the rising topics in automotive development. The main aim of this thesis is to obtain a real-time compatible lane detection system that satisfies functional safety automotive standard ISO 26262 to assist the driver/autonomous vehicle systems in efficiently detecting road lanes. This project investigates an optimized architecture of Hough transform suitable for FPGA implementation and with the help of designed Edge Detection and Cordic blocks, the lanes of a road can be detected and drawn accurately. The system is also verified using UVM and formal verification methods in addition to inferring safety mechanisms to achieve ASIL-C in ISO 26262 standard. Zynq Ultrascale+ ZCU104 Evaluation Board is used to test the lane detection system that operates on 260 MHz with 1000 fps which makes it suitable to modern high-speed cameras used in automotive safety testing.

Keywords: Lane Detection, Hough Transform, Edge Detection, Cordic, UVM, Formal Verification, Functional Safety, ISO 26262

## Chapter 01: Introduction

### 1.1. Organization of the thesis:

Chapter 1 provides an overview of the thesis, motivation and explanation of Lane Detection. Chapter 2 shows Hough Transform related work and algorithm and how it is optimized to be suitable for real-time applications. Chapter 3 illustrates Edge Detection and Cordic Algorithms. Chapter 4 explains the Software modeling of Lane Detection system. Chapter 5 shows the hardware implementation of Edge Detection, Cordic block and the proposed optimized architecture of the Hough Transform algorithm. Chapter 6 is concerned with the Formal Verification of the design and discusses the required assertions. Chapter 7 illustrates how the design was functionally verified using the UVM environment. In chapter 8, FPGA implementation on Zynq UltraScale+ and experimental results are discussed. Chapter 9 shows the functional safety of the design including ASIL levels, Safety Mechanisms and Functional Safety tools.

### 1.2. Motivation:

In recent years, there is an increasing number of individuals showing a growing dependence on transport, it obviously serves an important function in our daily lives. However, such increasing dependence has also increased traffic accident occurrence and fatality rates. The Fatality Analysis Reporting System data of the National Highway Traffic Safety Administration (NHTSA) shows that traffic accidents caused by driver negligence and unintentional lane deviations account for 41% of all trac accidents in the United States, as shown in Figure1:

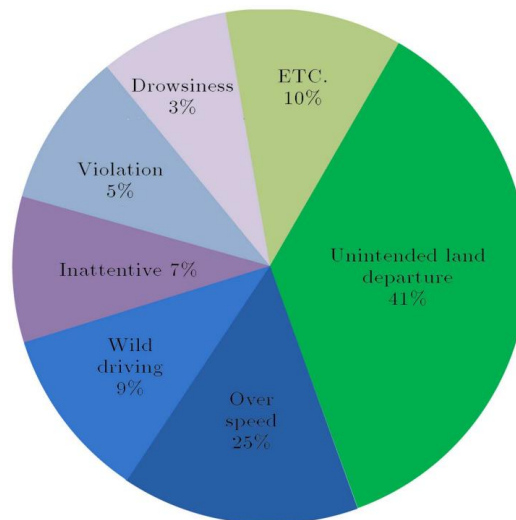


Figure 1: NHTSA Data

ADAS (Advanced Driver Assistance Systems) is considered the first step in the future development of unmanned intelligent vehicles. A self-driven car **needs to understand its surroundings** so it can navigate its way in the streets with minimal human assistance, so for this target, driver and passenger safety systems are one of the rising topics in automotive development. Currently, ADAS cannot control a vehicle independently from the driver. The system provides several working and environmental situations around a vehicle which can be analyzed and determined by a microprocessor so that ADAS can warn drivers of traffic accidents in advance.

ADAS system consists of nearly nine functions and the most notable is Lane Departure System. A Lane Departure Detection System (LDS) is an alarm system that helps drivers avoid switching lanes unintentionally. A survey by Mercedes-Benz suggests that 90% of road collisions can be avoided if the driver can be effectively warned one second before the accident. [1]

So, Lane detection is an important foundation in the development of intelligent vehicles (i.e. ADAS and LDS) and to address problems such as low detection accuracy of traditional methods and poor real-time performance of deep learning-based methodologies, a lane detection algorithm for intelligent vehicles in complex road conditions and dynamic environments is proposed.

### 1.3. What is Lane Detection:

As stated in the previous section, Lane detection is the basis of many Advanced Driver Assistance Systems (ADAS), such as Lane Departure Warning System (LDWS) and Lane Keeping Assist System (LKAS). Lane Detection is the process of detecting white or yellow markings on a dark road; and then drawing the lanes or in case of exit of the lanes, they allow triggering a visual or audible warning to inform the driver, or to control the vehicle thanks to automated control systems to avoid an accident. [2]

### 1.4. Lane Detection Techniques:

In [3] several vision-based techniques deployed for marked roads are presented. Using Gaussian Based Road model, lanes could be detected using a multi-stage algorithm. The processes involved in these stages are histogram generation, fitting, and normalization. Different ordered polynomial functions are used for lane fitting. Popular algorithms include the GOLD system developed by Broggi which uses an edge-based lane boundary detection algorithm. Remapping of the acquired image is done to represent in bird's eye view of the road. Extraction of quasi-vertical bright lines that are concatenated into specific larger segments is done using specific adaptive filtering. Also, Hough Transform (HT) is a commonly used method for the detection of the straight road lane. The merit of the HT is its robustness and its detection efficiency even though noise is present in the image. In this thesis, HT is implemented making the use of optimization presented in [2]. The Process of Designing a "Functional Safe Lane Detection Hardware Accelerator" goes into several stages; starting with "Hardware Design" followed by "Formal and UVM Verification" and finally ending with "Functional Safety" and all these phases are described in detail in this thesis.

### 1.5. Functional safety

As discussed in previous sections lane detection block should be functionally safe which means it should have high immunity to any probable failure in the design so Functional Safety is implemented in the proposed design.

Functional safety is the part of a system or device that provides automatic protection for operating correctly even in the cases of a failure in a predictable manner. The automotive industry, has developed the ISO 26262 Road Vehicles FUSA Standard. The certification of systems that follows these standards ensures the compliance with the important regulations and helps to decrease probability of accidents and avoid possible crisis. The ISO 26262 standard defined a risk classification system for the functional safety of road vehicles which is Automotive Safety Integrity Level abbreviated as ASIL. There are four ASIL levels set by ISO 26262 which are A, B, C, and D. Level A represents the lowest hazard rate while Level D represents the highest degree of hazard rate. All Functional Safety concepts are explained in detail in chapter 9.

## Chapter 02: Hough Transform Related Work and Algorithm

As stated in chapter 1, In embedded vision applications such as LDWS and Lane-keeping assistance systems (LKAS), HT is a commonly used method for the detection of straight road lanes. The merit of the HT is its robustness and its detection efficiency even though noise is present in the image.

### 2.1. The idea of the Hough Transform

#### 2.1.1. Representation of lines in Hough Space (HS):

Lines can be represented uniquely by two parameters. Often the form in Equation 2.1 is used with parameters  $m$  (slope) and  $b$  (y-interception).

$$y = m \cdot x + b \quad (2.1)$$

This form is, however, not able to represent vertical lines. Therefore, the Hough transform uses the form in Equation 2.2, which can be rewritten to Equation 2.3 to be similar to Equation 2.1. The parameters  $\theta$  and  $\rho$  are the angle of the line and the distance from the line to the origin respectively.

$$\rho = x \cdot \cos \theta + y \cdot \sin \theta \quad (2.2)$$

$$y = -\frac{\cos \theta}{\sin \theta} \cdot x + \frac{\rho}{\sin \theta} \quad (2.3)$$

All lines can be represented in this form when  $\theta \in [0, 360[$  and  $\rho \geq 0$ . The Hough space for lines has therefore these two dimensions;  $\theta$  and  $\rho$ , and a line is represented by a single point, corresponding to a unique set of parameters  $(\theta_0, \rho_0)$ . The line-to-point mapping is illustrated in Figure 2

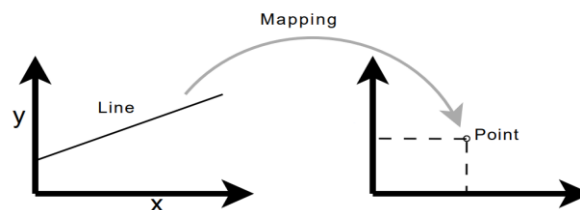


Figure 2: Mapping of a line to the Hough space

#### 2.1.2. Mapping of Points to Hough Transform:

An important concept for the Hough transform is the mapping of single points. The idea is, that a point is mapped to all lines, that can pass through that point. This yields a sine-like line in the Hough space.

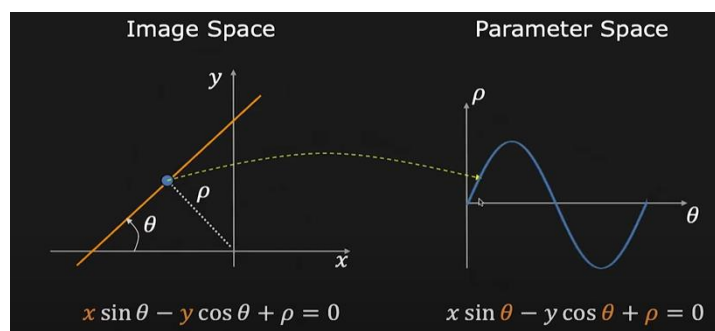


Figure 3: Transformation of a single point ( $po$ ) to a line in the Hough space.

### 2.1.3. Algorithm:

The algorithm for detecting straight lines can be divided into the following steps:

1. Edge detection (will be discussed in detail in chapter 3)
2. Mapping of edge points to the Hough space and storage in an accumulator.
3. Interpretation of the accumulator to yield lines of infinite length.
4. Conversion of infinite lines to finite lines. The finite lines can then be drawn back to the original image.

The Hough transform itself is performed in point 2, but all steps including edge detection are covered in this thesis.

### 2.1.4. Transformation of lines to Hough Space:

The Hough transform takes edges from the “Edge Detection” stage through “Cordic Block” as input and attempts to locate edges placed as straight lines. The idea of the Hough transform is, that every edge point in the edge map is transformed to all possible lines that could pass through that point. Figure 4 illustrates this for two points.

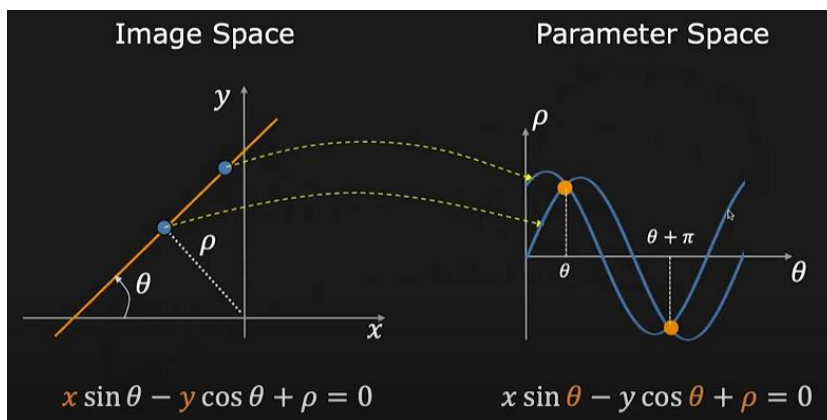


Figure 4: Transformation of two points to two lines in the Hough space.

A typical edge map includes many points, but the principle for line detection is the same as illustrated in Figure 4 for two points. Each edge point is transformed into a line in the Hough space, and the areas where most Hough space lines intersect are interpreted as true lines in the edge map.

### 2.1.5. The Hough Space Accumulator:

To determine the areas where most Hough space lines intersect, an accumulator covering the Hough space is used. When an edge point is transformed, slots in the accumulator are incremented for all lines that could pass through that point. The resolution of the accumulator determines the precision with which lines can be detected. In this thesis, the analysis of precision and the trade-off between precision and speed are discussed in chapter 5.

### 2.1.6 Detection of Infinite Lines:

Infinite lines are detected by interpretation of the accumulator when all edge points have been transformed. An example of the entire line detection process is shown in Figure 5. The most basic way to detect lines is to take the maximum vote slots for the required number of lines (for example, the two maxima for right and left lanes in the image) and then take the corresponding  $\rho$ 's and  $\theta$ 's.

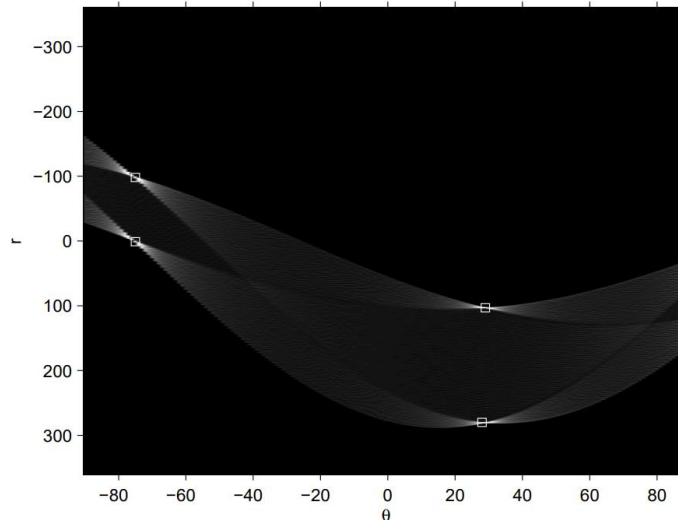


Figure 5: Representation of multiple edges on the lanes in Hough Space

### 2.1.7. Finite Lines:

The classical Hough transform detects lines given only by the parameters  $\rho$  and  $\theta$  and no information with regards to length. Thus, all detected lines are infinite in length. If finite lines are desired, some additional analysis must be performed to determine which areas of the image that contributes to each line. In this thesis, the detected  $\rho$  and  $\theta$  are used to generate the slope and y-interception, which will be discussed later in this chapter, and then using line drawer block it's converted to finite lines drawn within borders of the original image.

After presenting Hough Transform algorithm, now how could it be implemented using hardware?!

## 2.2. Proposed Hardware Architecture:

Several hardware (HW) and software (SW) implementations are investigated in the literature in order to make HT computationally efficient or to adapt its use to real-time constraints. However, in the high-level and complete tools like Matlab, C++ the requirements in terms of real-time processing and energy consumption are not always taken into account. In addition, advances in the integrated circuits field enable us to have programmable hardware platforms to design and implement HT that are less demanding in terms of material resources and energy consumption while respecting the constraints of real-time. the main drawbacks of the HT are its high memory requirements and computational complexity, which impose a limitation on the use of the transform for real-time applications on basic FPGA devices. Moreover, actual FPGA implements also HW and SW CPU, which increases its attractiveness for embedded vision applications. In order to attain a real-time lane detection system, a new architecture of HT for straight lane detection more adapted for an FPGA implementation is proposed.

Equation 2.2 presents the conventional approach of HT and figure 6 shows the corresponding architecture.

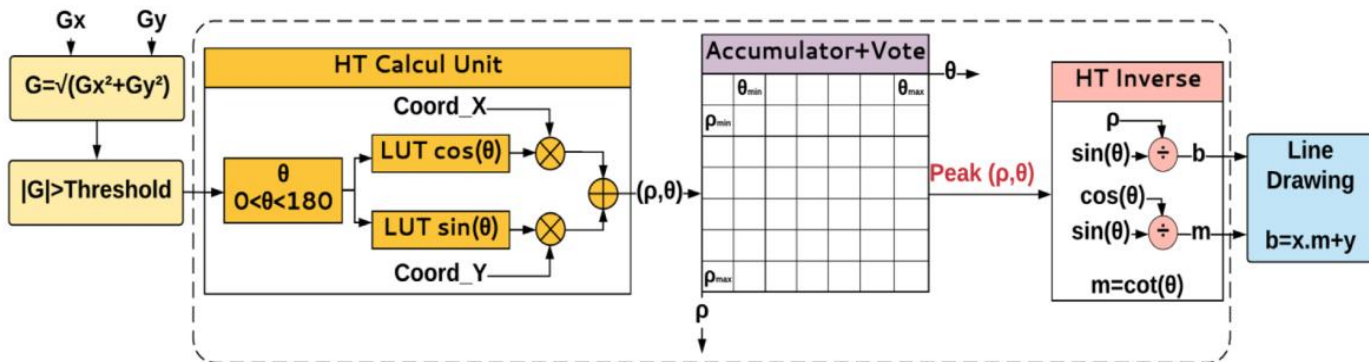


Figure 6: Represents Hardware Architecture for Conventional HT

The input is a contour image and the output is the slope and the Y-intercept of the detected line. At the end of the voting process, Inverse Hough Transform (IHT) must be performed to return to the image space. Therefore, additional polar to Cartesian coordinate transformation is required at the output of the HT stage. If we represent a line, these inverse transformations are calculated by the following equations.

$$m = -\cot(\theta) \text{ and } b = \frac{\rho}{\sin(\theta)} \quad (2.4)$$

$m$  and  $b$  represent slope and y-interception of equation 2.1 which means straight lines equations can be obtained successfully. So, from the conventional HT, it's noted that the last stage requires additional trigonometric and arithmetic calculations requiring significant resources and consuming additional clock cycles.

In the proposed architecture, it's taken into account the specificities of the straight lane in the image plane. Because the transformation from 3D to 2D plan, if the central optical axis of the used camera is positioned to the center of the car's windshields, the left and right lines of the lane are never parallel and cannot be horizontal or vertical on the image plane. If horizontal lines  $\theta = 0$  and  $\theta = 180$  are discarded from the detected edge lines, equation 2.3 can be obtained from equation 2.2 which is very similar to the original straight-line equation 2.1. And with the help of equation 2.4 we get equation 2.5:

$$y = -\cot\theta \cdot x + \frac{\rho}{\sin\theta}, \text{ where } m = \cot\theta \text{ and } b = \frac{\rho}{\sin\theta} \quad (2.5)$$

Now, the accumulation and the voting process are based on equation 2.5. If horizontal or almost horizontal lines are discarded, this parameterization is bounded and can represent all possible lines in a finite parameter space region.

For FPGA implementation, we simplify the HW cost of mapping the image plan  $(x, y)$  to the Hough Space. If for Equation 2.2, the trigonometric values of  $\sin(\theta)$  and  $\cos(\theta)$  are obtained from two Look Up Tables (LUT) and require two multipliers, the proposed implementation for equation 2.5 requires only one LUT and one multiplier. Furthermore, the same equation of mapping is exploited to output the Y-intercept and the slope of the detected line. Therefore, the proposed architecture optimizes both HT and IHT, which reduce the calculation complexity and the memory consumption.

From the other side, many edge points are not a part of a lane contour. As shown in figure 7, the angle  $\theta$  of right and left lane marking varies within limited intervals. Pixels with a gradient direction out of these intervals can be discarded and not processed at HT stage. This is implemented by defining two ROIs for left and right lane lines: the left limit varies between  $\theta_{Lmin}$  and  $\theta_{Rmax}$ , and the right limit varies between  $\theta_{Rmin}$  and  $\theta_{Rmax}$ . The two ROIs are defined experimentally.



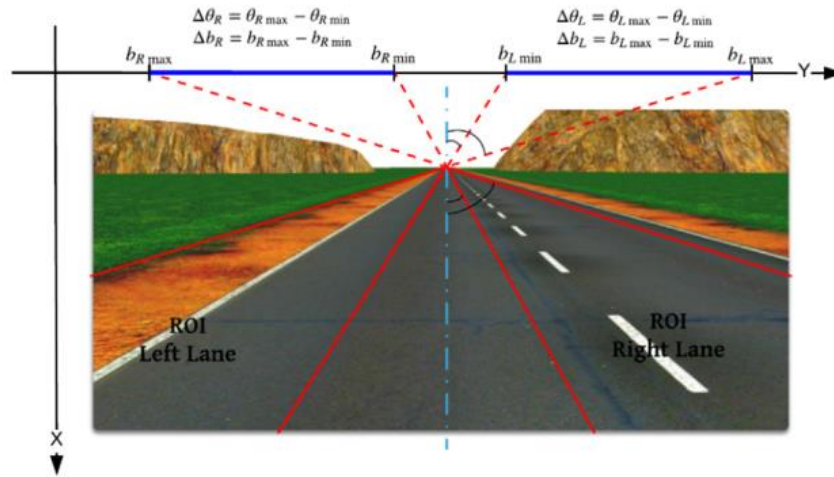


Figure 7: Definition of the two regions of interest (ROIs) for left and right lane boundaries

### Summary:

- According to the methodology described above, the proposed architecture can be easily synthesized on an FPGA platform. The overall system is showed in figure 8

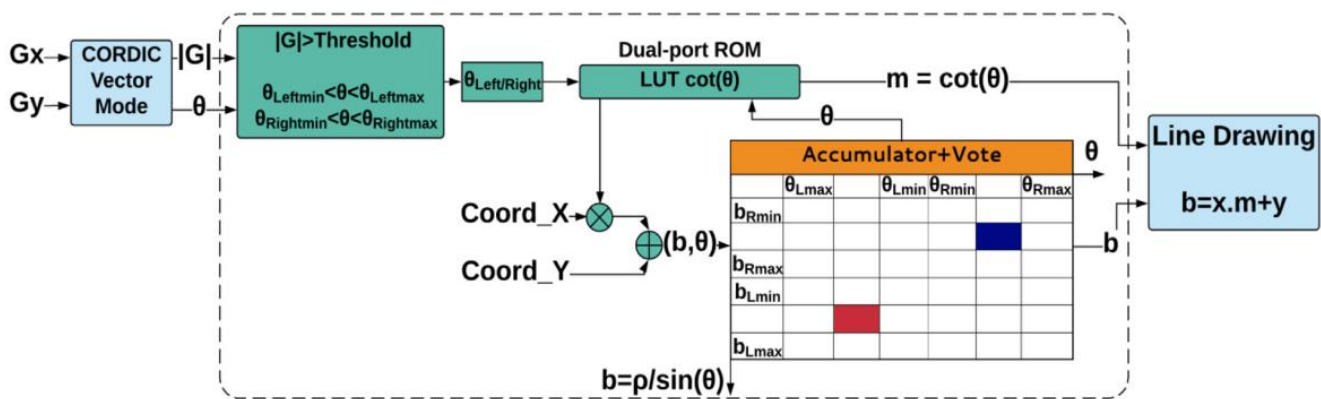


Figure 8: Proposed Architecture for Straight Lane Detection

- The input to Hough Transform Stage comes from Cordic block which provides the edges lying in the left and right regions of interest
- There is an Edge Detection block that does the image processing and then pass the edges, Gx and Gy to be manipulated by Cordic block before being sent to Hough Transform.
- Hough Transform block applies the described methodology and pass the slopes and intersection of the lines to line drawer to be drawn on the original input image (taken by a camera)

The algorithms of Edge Detection and Cordic block are described in chapters 3 and 4, and then hardware implementations of all the three blocks are described in detail in chapter 5.



## Chapter 03: Edge Detection and Cordic Algorithm

### 3.1. Edge Detection

Lane detection is mainly based on digital image processing and features extraction. There are a lot of features that differentiate straight road lanes from any other objects such as straight-line features (slope and y-intercept) as discussed in chapter 2, and edge detection which will be illustrated in this section. [4]

#### 3.1.1. Proposed Algorithm

The idea of edge detection is identifying pixels in an image at which light intensity or the gray scale changes sharply as shown in figure 9. Edge detection is a tool used in image processing for feature detection and extraction. This algorithm significantly reduces data to be processed and may therefore remove less relevant information while preserving important properties of an image. If this algorithm is successful, the task of interpreting the information in original image may be simplified. However, it is not always possible that ideal edges can be obtained from real life images of modern complexity. [4]

In figure 9, an ideal edge pixel and the corresponding gradient vector are shown. At the pixel (the highlighted circle), the intensity changes from 0 (the dark part) to 255 (the light part) at the direction of the gradient. The magnitude of the gradient indicates the strength of the edge. If we calculate the gradient at uniform regions, we end up with a 0 vector which means there is no edge pixel. In natural images we usually do not have the ideal discontinuity or the uniform regions as in the figure 9. and we process the magnitude of the gradient to make a decision to detect the edge pixels such as comparing by a threshold or comparing by minimum and maximum limits. The elementary processing is applied for a threshold. If the gradient magnitude is larger than the threshold, we decide the method in corresponds to the edge pixel. An edge pixel is described by using two important features, primarily the edge strength, which is equal to the magnitude of the gradient and secondarily edge direction, which is equal to the angle of the gradient. [5]

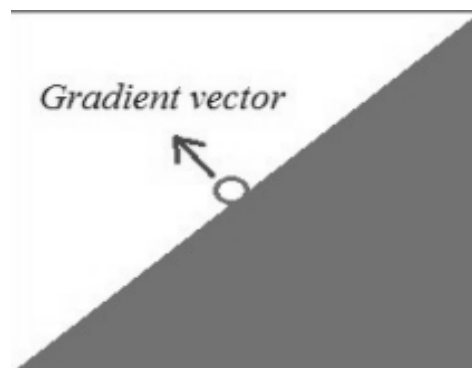


Figure 9: Gradient vector at some pixel

#### 3.1.2. Sobel Filter

In order to calculate the magnitude and the direction of the gradient, many methods can be used. The Sobel operator is the most suitable choice because of its simplicity and ability to detect the edges and their direction but it is sensitive to noise so Gaussian filter is used before it to reduce image noise. [6]

The Sobel operator consists of a pair of  $3 \times 3$  convolution kernels as shown in figure 10. These kernels are designed to respond maximally to edges running vertically and horizontally, one kernel for each of the two perpendicular orientations. The kernels can be applied separately to the input image, to produce separate

measurements of the gradient component in each orientation (call these  $G_x$  and  $G_y$ ). These can then be combined to find the absolute magnitude of the gradient at each point and the orientation of that gradient. [6]

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

$$F_x$$

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

$$F_y$$

Figure 10: Sobel Operator Filters

The magnitude of the gradient is calculated by:

$$|G| = \sqrt{G_x^2 + G_y^2} \quad (3.1)$$

Where  $G_x$  and  $G_y$  are the image matrix convolved with the filters  $F_x$  and  $F_y$  as follows:

$$G_x = Image * F_x \quad (3.2)$$

$$G_y = Image * F_y \quad (3.3)$$

Convolution operation is explained in detail in chapter 4.1.1.

For efficient hardware implementation, the following gradient equation approximation is applied. [7]

$$|G| = |G_x| + |G_y| \quad (3.4)$$

The direction of the gradient (the angle of orientation of the edge) is given by:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right) \quad (3.5)$$

After applying the Sobel filter only pixels with change in intensity are passed.



Figure 11: (a) Original Image, (b) Sobel Operator Output.

### 3.1.3. Gauss Filter

The Gaussian operator is a low pass filter which smoothens the image, reduces sharp changes in the edges, and removes noise by removing the higher frequency components. It consists of a 3x3 kernel which calculates the weighted average of the surrounding pixels of the current pixel. The existence of the Gauss filter with the Sobel filter is very important as the Sobel filter is very sensitive to noise.

$$\frac{1}{16} \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

Figure 12: Gauss Kernel.



Figure 13: (a) Original Image, (b) Gauss Operator Output.

## 3.2. Co-Ordinate Rotational Digital Computer

Co-Ordinate Rotational Digital Computer (CORDIC) is a hardware-efficient iterative method which uses rotations to calculate a wide range of elementary functions, the main idea is simply rotating the input vector. The list of the functions that can be calculated from rotation is relatively long. Inverse trigonometric functions such as tan inverse, sin inverse, cos inverse, hyperbolic and logarithmic functions, multiplication and division are some of the most important operations that can be obtained from variants of rotation. The CORDIC algorithm attempts to provide a hardware-efficient method for calculating these functions. “Hardware-efficient” means that the algorithm avoids the use of multipliers and relies on only shifters, adders and subtractors.

We need CORDIC in our design to obtain the theta of the edge and then decide if this edge is in the right region, left region or out of region of interest as illustrated in the following figure.

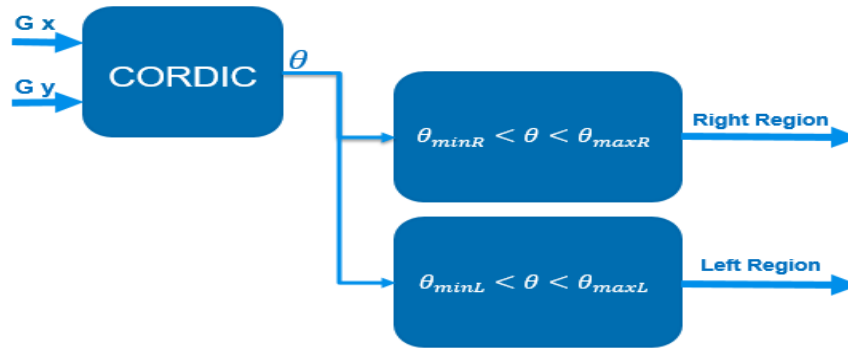


Figure 14: Illustrative Diagram for region of interest of theta

The idea of rotation matrix to obtain theta is so simple, let's say we have co-ordinates  $X_{in}$  and  $Y_{in}$  and we want to calculate the angle  $\beta$  as shown in figure 15, we rotate axis by  $45^\circ$  anti clock wise, then check on value of Y if it is still positive we rotate one more time anti clock wise by  $22.5^\circ$  and if it is negative then we rotate clock wise by  $22.5^\circ$ , we repeat this process many iterations each iteration by  $\theta(i+1) = \frac{\theta(i)}{2}$  till the value of Y converges to zero. Angle  $\beta$  = summation of all rotation angles.

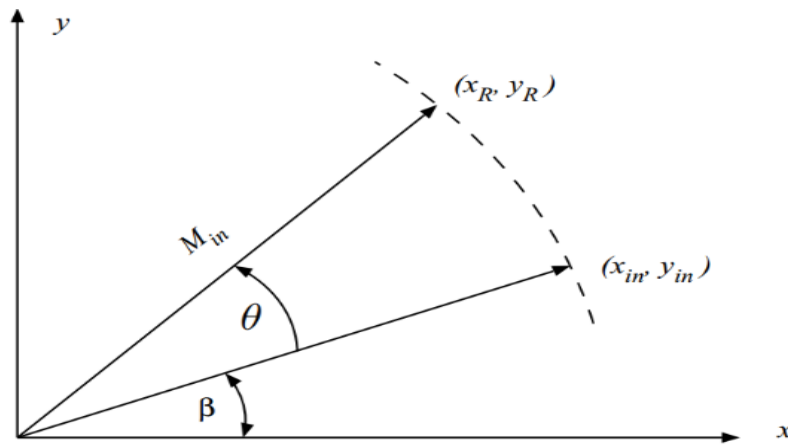


Figure 15: Rotating input vector Axis

The direct idea of rotation matrix to obtain theta costs 2 multiplications, 2 additions or 2 subtractions and 2 LUTs for sine and cosine which is inefficient, CORDIC algorithm introduces efficient approximations.

$$\text{Vector Rotation: } X' = X \cos(\theta) - Y \sin(\theta) \qquad Y' = Y \cos(\theta) + X \sin(\theta)$$

$$\text{Rewrite as: } X' = \cos(\theta) [X - Y \tan(\theta)] \qquad Y' = \cos(\theta) [Y + X \tan(\theta)]$$

Trick: Allow only iterative rotations so that  $\tan(\theta) = \pm 2^{-i}$  as it is implemented simply as a shifter.

$$X_{i+1} = \cos(\tan^{-1}(\pm 2^{-i})) [X - Y d_i 2^{-i}] \quad \text{Where, } d_i = \pm 1$$

$$Y_{i+1} = \cos(\tan^{-1}(\pm 2^{-i})) [Y + X d_i 2^{-i}]$$

The  $\cos(\tan^{-1}(\pm 2^{-i}))$  can be pre-computed for all iterations and this term is called CORDIC gain. So we only have subtraction, addition and shifting operations instead of multipliers and lookup tables.

## Chapter 04: Software Model

### 4.1. Edge Detection

The software model of the edge detection was modified to match the fixed-point representation. The error due to neglecting the decimal part was  $\leq 0.2\%$  for different images, so it can be neglected without affecting the output accuracy. For both of the Gauss filter and the Sobel filter the borders of the image are ignored so for a  $512*512$  image only a window of size  $510*510$  is filtered.

#### 4.1.1. Gauss Filter

What meant by filtering is convolution between the image and the filter and this is done by multiplying the filter by a window from the image which has the same size of the filter then accumulating the result of multiplication. After that, the window is shifted horizontally by one stride then multiply and accumulate until reaching the horizontal end of the image. Lastly the window is shifted vertically by one stride and the previous steps are repeated to cover the whole image as shown in the following figure.

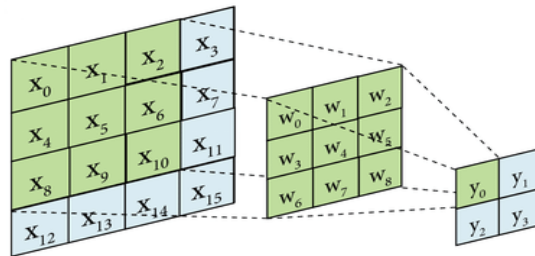


Figure 16: Convolution Between filter and matrix

For more illustrations, the output image pixels are calculated by:

$$y_0 = x_0 * w_0 + x_1 * w_1 + x_2 * w_2 + x_4 * w_3 + x_5 * w_4 + x_6 * w_5 + x_8 * w_6 + x_9 * w_7 + x_{10} * w_9 \quad (4.1)$$

$$y_1 = x_1 * w_0 + x_2 * w_1 + x_3 * w_2 + x_5 * w_3 + x_6 * w_4 + x_7 * w_5 + x_9 * w_6 + x_{10} * w_7 + x_{11} * w_9 \quad (4.2)$$

and so on. The following figure shows the input image and the output of the Gaussian filter:



Figure 17: Image before and after the Gaussian Filter.

### 4.1.2. Sobel Filter

As explained in chapter 3, the Sobel operator has two filters ( $F_x$  and  $F_y$ ) each one of them is convolved with image in the same way as the Gaussian filter to produce two matrices ( $G_x$  and  $G_y$ ) then, the gradient magnitude matrix is calculated by equation (3.4), and the theta (orientation) matrix is calculated by equation (3.5).

Because lanes have sharp changes in the intensity which result in a large gradient magnitude value, the threshold value is set to a proper value to ignore all pixels whose gradient magnitude is less than this threshold as shown in figure 18 to reduce the required processing time because the hardware blocks that are next to the edge detection block work on each pixel individually. The threshold value is determined by trial and error in the software model.

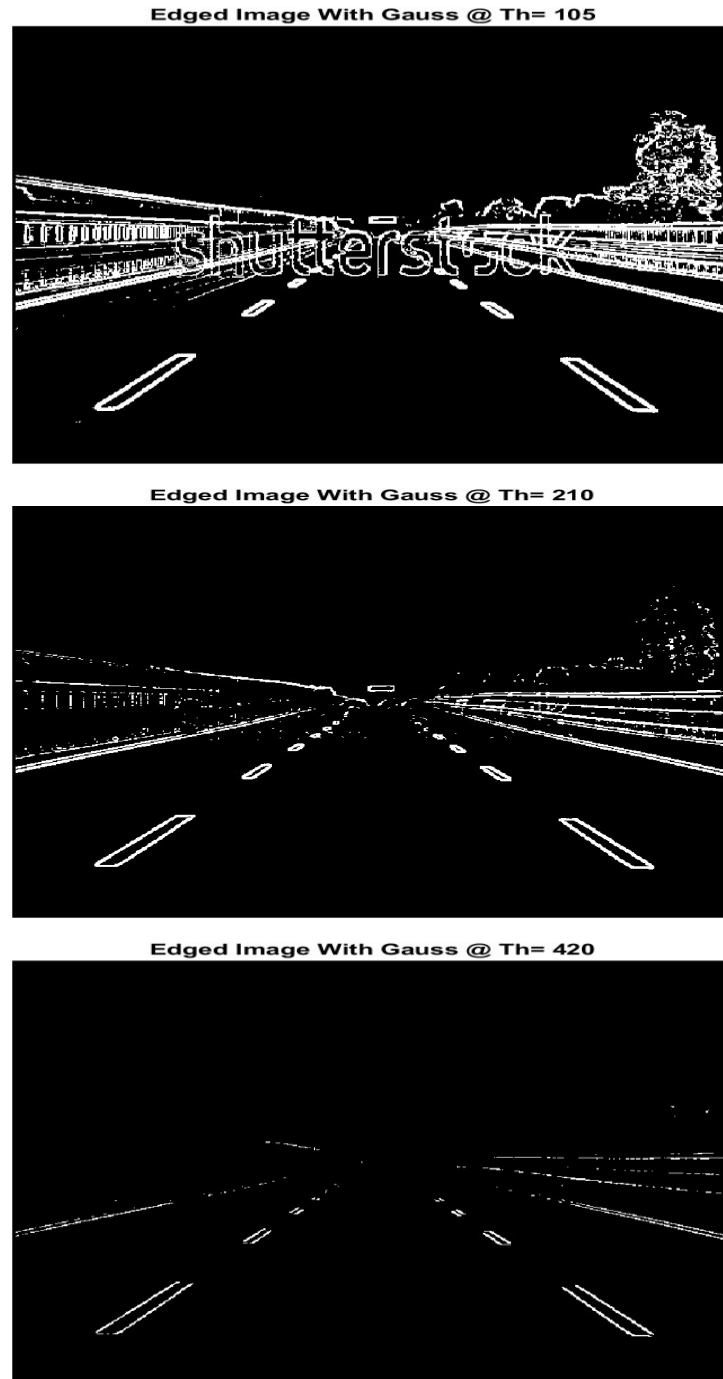


Figure 18: Edge detection with a small, moderate, and large threshold values

As mentioned before the Sobel filter is sensitive to noise, so the Gauss filter is very important to reduce the noise, the following figure shows the difference between the output of the edge detector with and without the Gauss filter.

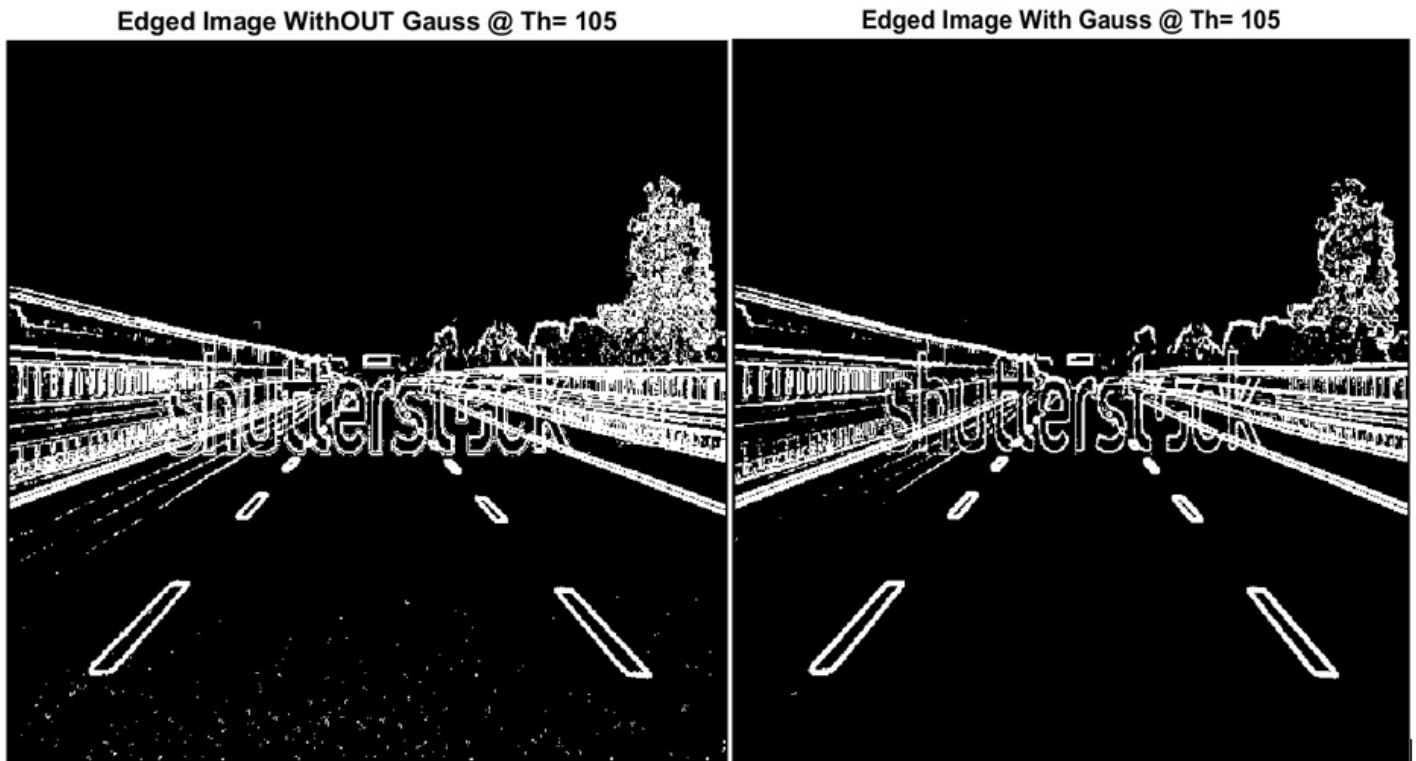


Figure 19: Edge detection output with and without Gauss filter

The final output of the edge detector after adding Gaussian filter and adjusting a proper threshold value is shown in the following figure:

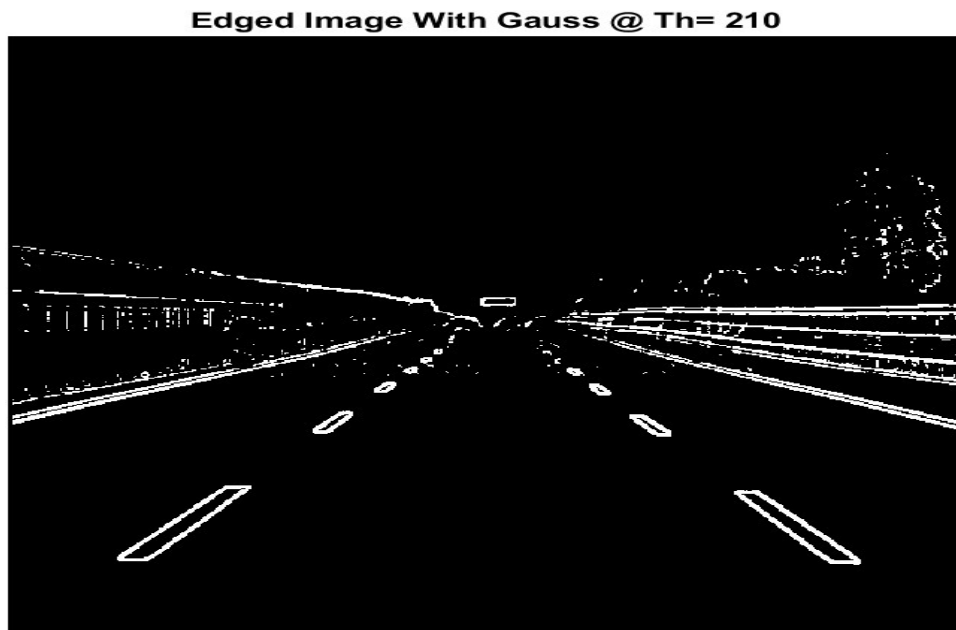


Figure 20: Final output of the edge detection block



## 4.2. Hough Transform

### 4.2.1. Region of Interest (ROI)

The first step in detecting the lanes is determining the region of interest to ignore all the pixels that are out of the ROI. The ROI is described by five parameters:

1. The vanishing point location.
2. Theta minimum left.
3. Theta minimum right.
4. Theta maximum left.
5. Theta maximum right.

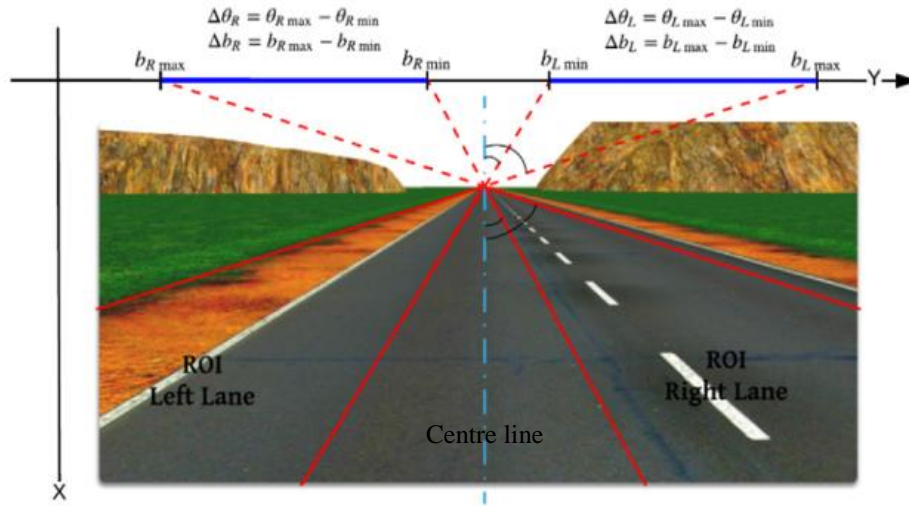


Figure 21: The Region of Interest

The vanishing point is the point of the sight end of the two lanes at the intersection of them as shown in figure 21. It is the most important parameter as it determines the location of the region of interest and on the other hand thetas, minimum and maximum determine the width of the ROI to contain the lanes inside of them to be detectable. Choosing wider ROI increases the processing time but it reduces the probability to miss the lanes for different roads with different widths. All these parameters are fixed parameters for the data set, predefined by trial and error, and they are suitable for images that have close vanishing point locations and close road widths.

### 4.2.2. Transformation

For each edge coming from the edge detection block, its theta is checked if it falls within the left or right region of interest and if it doesn't, it is excluded. If the edge falls in the right or left ROI, the y-intercept ( $b$ ) is calculated for all lines inside the same ROI and pass by this edge location and this is done by equation (4.3) where  $x$  and  $y$  are the location of the edge and theta is the angle of each line with the x-axis inside the ROI. In other words,  $b$  is calculated many times by iteration on the equation (4.3) at each time theta is substituted by theta minimum + theta\_step \* I (where I is the iteration number starting from zero) till reaching theta maximum.

$$b = \cot(\theta) * x + y \quad (4.3)$$



### 4.2.3. Voting

For every calculated  $b$ , it is checked if it falls within the same ROI of the edge (i.e less than  $b$  maximum and greater than  $b$  minimum) and if it doesn't it is ignored. If the  $b$  falls in the ROI, the vote which corresponds to this  $b$  and its theta of the voting matrix is increased by one.

	Theta minimum	.....	Theta maximum
$b$ minimum			
.		Voting values	
.			
.			
$b$ maximum			

Figure 22: Voting Matrix

After iterating on the whole image and finishing iterating on each edge in the left and right ROI, two  $b$ s and thetas in the same ROI that corresponds to the two maximum voting values are taken as the  $b$ s and thetas of the detected lane in this ROI and this is done also for the other ROI to finally draw four lines for four edges (two for each lane).

### 4.2.3. Inverse Hough Transform and Line Drawing

In this step the  $b$ s and thetas are used to calculate the location of the detected lanes. This is done by the equation (4.3) but this time the  $y$  value is calculated by substituting  $x$  by the  $x$ -axis values of the image. Finally, the lines are highlighted over the image to validate the output as shown in the following figure.



Figure 23: Final Software Output

## Chapter 05: Hardware Implementation

### 5.1. Edge Detection Block

As discussed in Chapter 3 (Edge Detection Algorithm), to detect the lanes, the edge positions and theta that is calculated by the cordic block through Gx and Gy parameters are needed. The edge detection block takes a 512\*512 gray encoded image as an input, and outputs the edge positions (X, Y), the parameters (Gx, Gy) and a flag that indicates an edge detection to enable the following blocks as shown in figure 24.

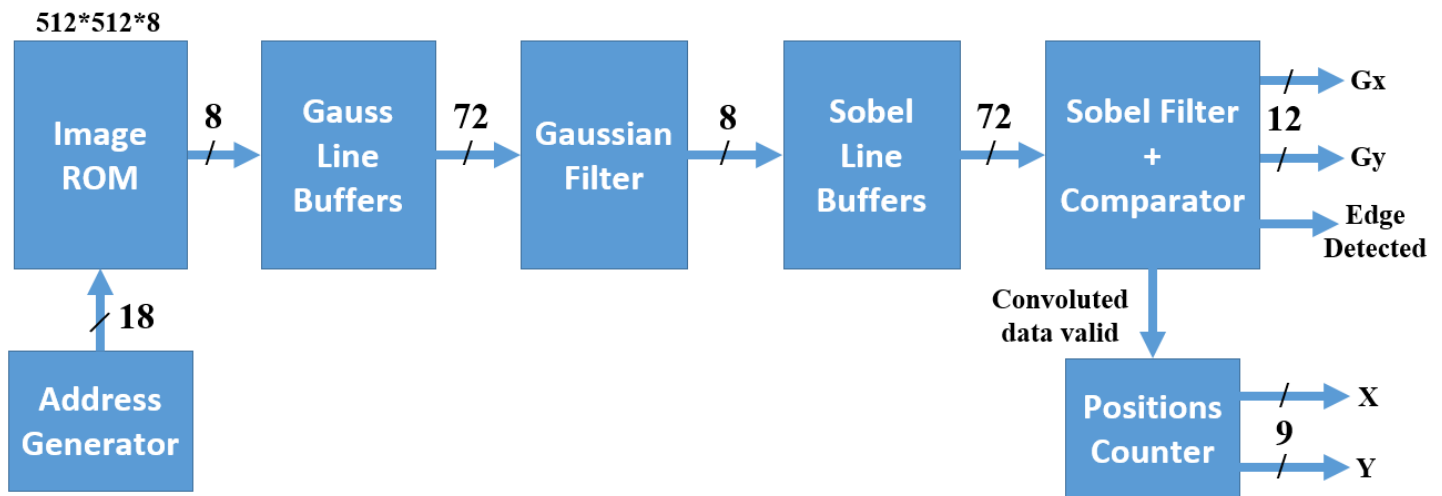


Figure 24: Edge Detection Block Diagram

#### 5.1.1. Image ROM & Address Generator

The chosen method to read the input image, after flattening it into a 2D array of depth = 512\*512 and pixel width = 8 bits using a MATLAB script, it is stored in a simple Read Only Memory. The address generator is a simple incrementing counter, once the operation starts it increments each cycle which means a pixel is read from the ROM each cycle without any interruptions until reading the complete image.

#### 5.1.2. Line Buffers

As discussed in chapter 3, edge detection is done by filtering the image through Gaussian filter and Sobel operator using a 3\*3 kernel. Passing an image through filters is challenging as it means reading 9 pixels at a time following a proper pattern. Using line buffers (LBs) solve the problem, as a line buffer is basically a modified RAM as shown in figure 25. Upon reading, it takes 1 address from the control unit and outputs the data in the addressed slot in addition to the 2 following slots. For example, given address x, the output of a line buffer will be {LB(x), LB(x+1), LB(x+2)} concatenated, which represents a row from the required kernel.

The complete kernel can be acquired by stacking a group of 3 line buffers, each represents a row from the image, i.e.: depth = 512 for Gauss LBs and depth = 510 for Sobel LBs. But, having 3 line buffers only will add a periodic latency of 512 cycles to fill a line buffer each time processing 3 consecutive rows is finished, i.e.: moving the kernel 1 step downward. So to overcome the latency an extra line buffer is added, as having a stack of 4 line buffers will add only an initial latency of 2047 cycles and once the line buffers are full for the first time, the operation will never be paused again as the design is fully pipelined. The dataflow between the 4 line buffers is controlled using line buffers control unit that will be discussed in the following sub-section.

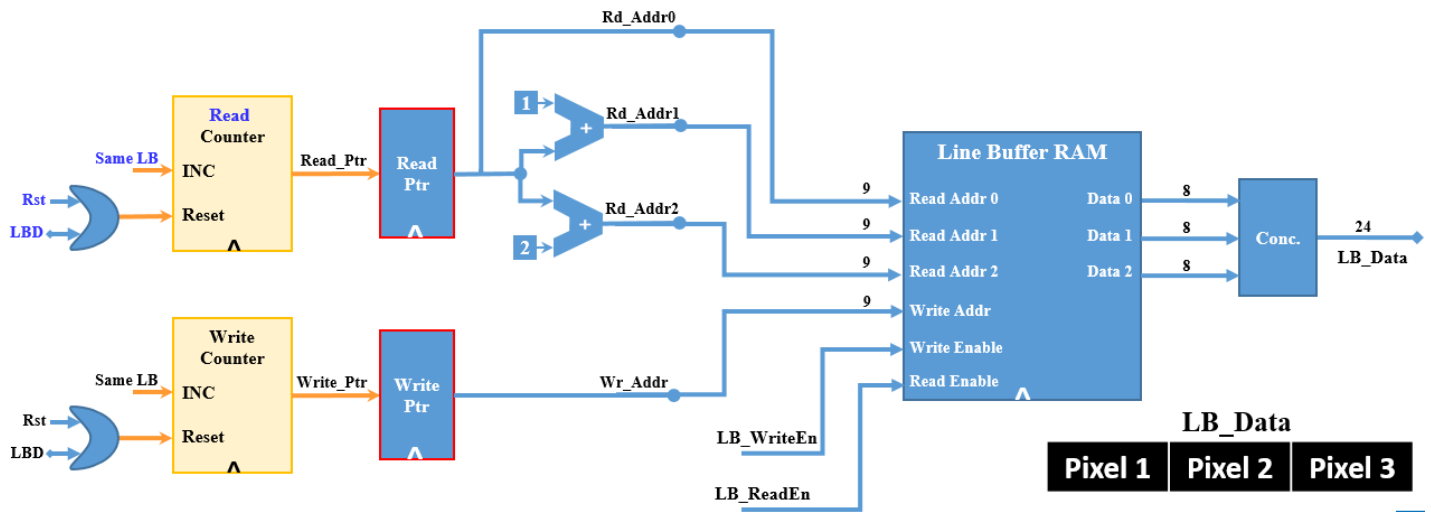


Figure 25: Line Buffer Block Diagram

### 5.1.3. Line Buffers Control Unit

The control unit handles the writing and reading enables of the 4 line buffers. Using counters, decoders and multiplexers it can control the full process which can be simplified as follows:

- State(1): Fill the 4 line buffers (one time latency = (image\_width\*4)-1 cycles)
- State(2): Enable reading from the first 3 line buffers (LB1,2,3) only, while writing a new row in the 1<sup>st</sup> LB such that the reading pointer exceeds the writing pointer (which is always the case)
- State(3): Enable reading from (LB2,3,4) only, while writing a new row in the 2<sup>nd</sup> LB.
- State(4): Enable reading from (LB3,4,1) only, while writing a new row in the 3<sup>rd</sup> LB.
- State(5): Enable reading from (LB4,1,2) only, while writing a new row in the 4<sup>th</sup> LB.
- Go back to State(2) and re-iterate until the full image is done.

The transition between line buffers while writing is done based on a (image\_width) counter due to writing one pixel at a time. But the transition while reading needs a (image\_width-2) counter only as 3 consequence pixels are read from the row at the same time.

The line buffers and their control unit are implemented as completely parametrized blocks to be able to use them twice, one time before each filter. Parametrization is needed because – as discussed in chapter 3 – the image size is reduced from (512\*512) to (510\*510) after passing through the Gaussian filter due to neglecting the borders as discussed in chapter 4 (Software Model).

### 5.1.4. Gaussian Filter

As discussed in chapter 3, a Gaussian filter is needed to eliminate the noise in the image. Convolution between the Gaussian filter and the kernel from the image can be simplified as follows:

$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,1} & P_{3,2} & P_{3,3} \end{bmatrix} = \frac{1}{16} \left( P_{1,1} + P_{1,3} + P_{3,1} + P_{3,3} + 4 \cdot P_{2,2} + 2 \cdot (P_{1,2} + P_{2,1} + P_{2,3} + P_{3,2}) \right)$$

That converges to a Multiply and Accumulate (MAC) unit, and there is a significant optimization that can be done due to the nature of the filter values. Since all the values are multiplicands of 2, the multiplication can be implemented as basic shift registers instead of multipliers as shown in figure 26.

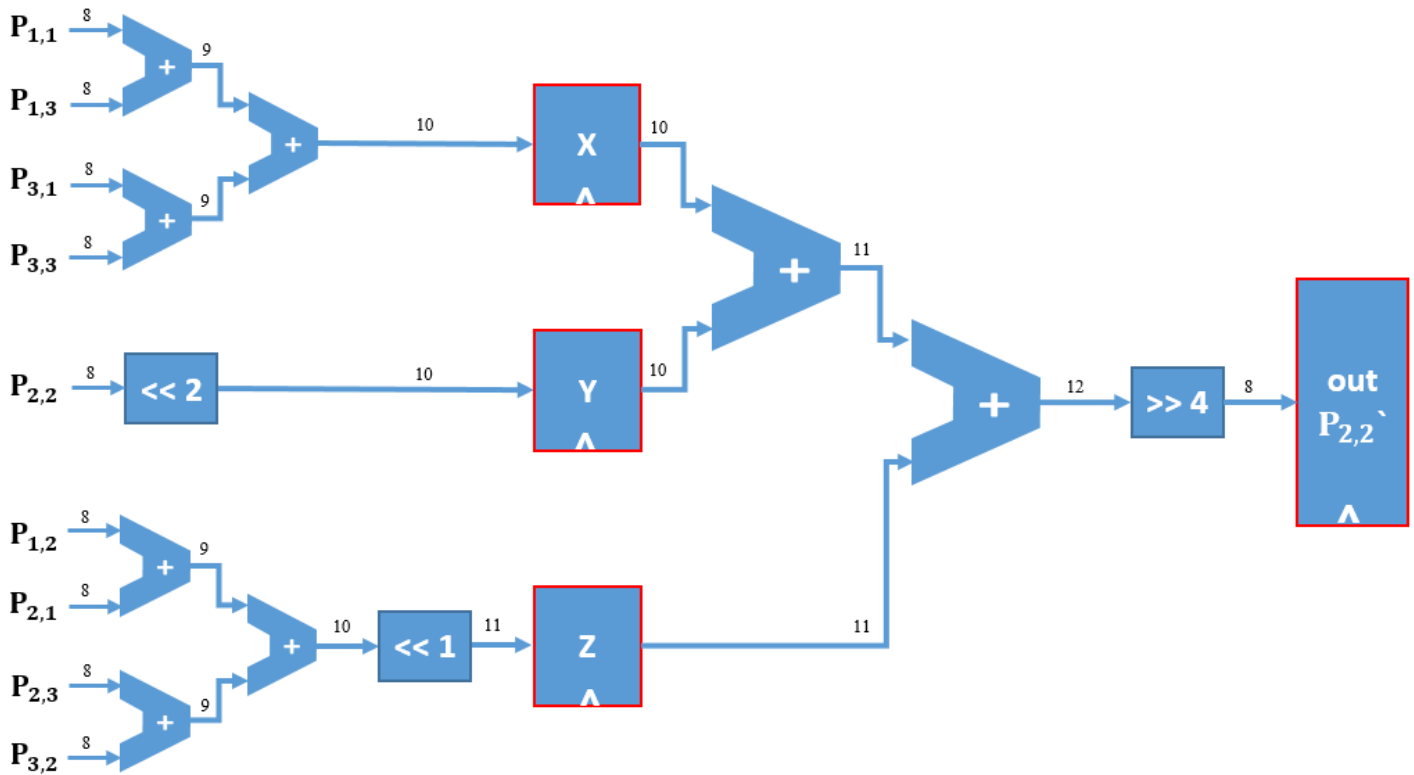


Figure 26: Gaussian Filter Block Diagram

### 5.1.5. Modified Sobel Operator

As mentioned in chapter 3, a basic Sobel operator is used to calculate the magnitude gradient of the pixel using gradient components  $G_x$  and  $G_y$ . Where  $|G| = \sqrt{G_x^2 + G_y^2}$  and the gradient components are the result of convoluting a kernel from the image with 3\*3 filters which can be represented as follows:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,1} & P_{3,2} & P_{3,3} \end{bmatrix} = (P_{1,3} - P_{1,1}) + 2 \cdot (P_{2,3} - P_{2,1}) + (P_{3,3} - P_{3,1})$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,1} & P_{3,2} & P_{3,3} \end{bmatrix} = (P_{3,1} - P_{1,1}) + 2 \cdot (P_{3,2} - P_{1,2}) + (P_{3,3} - P_{1,3})$$

As done in the Gaussian filter, these convolution operations can be optimized to a simple unit and shift registers as shown in figure 27-a. Also, since the gradient components are needed by the Cordic block to calculate the value of theta, so they are synchronized to propagate to the output with the ED flag as shown in figure 27.

As discussed in chapter 3 and since the outputs required from edge detection block in the main application (lane detection) does not include the gradient value, it can be simplified to  $|G'| = |G_x| + |G_y|$ . [7] That is implemented as shown in figure 27-b, where the absolute unit is built based on basic principles of logic design to achieve the best performance. Eventually the modified gradient magnitude is compared to a threshold value (Th.) and the “Edge Detected” (ED) flag is triggered if  $(|G'| > Th.)$  as shown in figure 27-c.

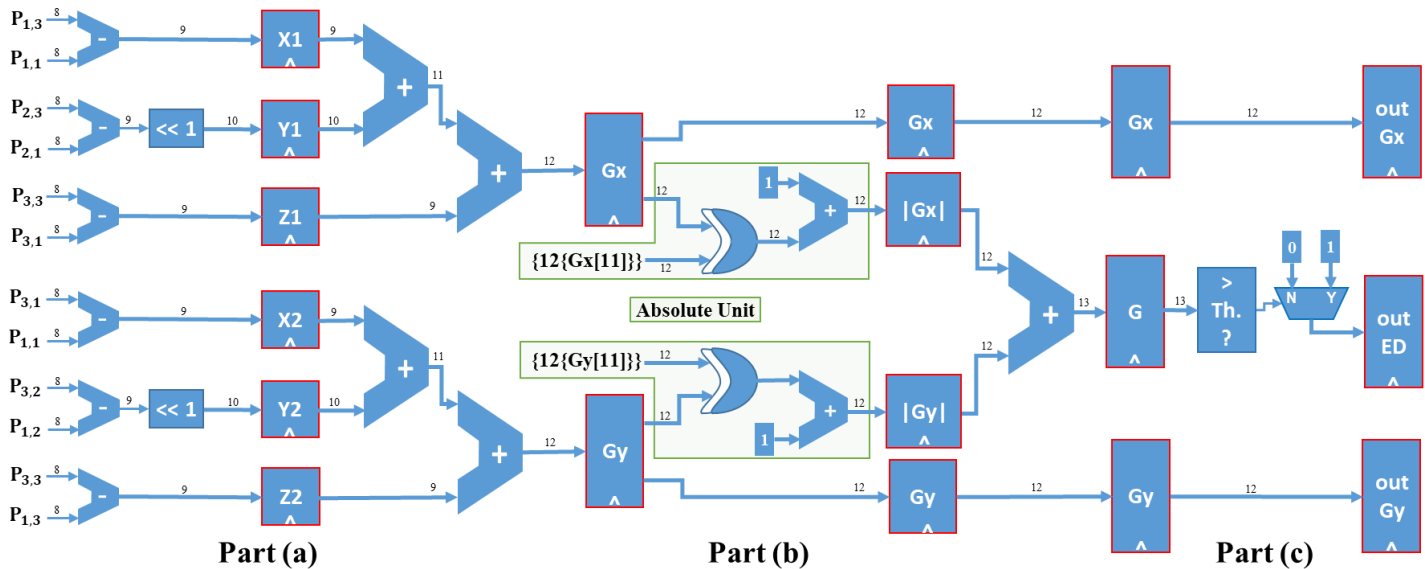


Figure 27: Sobel Filter Block Diagram

### 5.1.6. Position Counters

Two counters are used to detect the edge position represented as  $(X, Y)$  starting from the top level corner  $(1, 1)$ . But, since the borders are neglected each time the image passes through a filter – as discussed in chapter 4 –, so actually the Sobel operator will start processing from the pixel  $(3, 3)$ . Both counters are initialized to 3 and waits for an Enable signal to start counting, which is the “*Convolved data valid*” signal shown in figure 24. This signal is triggered once the Sobel filter starts to output data, as it is actually the main start signal propagating through the internal pipeline stages to be synchronized with the outputs.

Since the design is fully pipelined, once the “*Convolved data valid*” signal is triggered, the Sobel filter output is updated each cycle while the position counters are counting in parallel displaying the position of the pixel that was just processed by the filter.

**Y-Counter:** For columns, it counts from 3 to 510, 1 count each cycle and wraps around to 3 again.  
**X-Counter:** For rows, it counts from 3 to 510, increments every time the Y reaches 510 only. Both counters stop at  $(510, 510)$  indicating that the full image is processed (discarding the borders).

The pixel position  $(X, Y)$  and gradient parameters  $(G_x, G_y)$  are captured by the Cordic block if and only if “*Edge Detected*” flag is triggered by the Sobel filter indicating that this pixel is actually an edge. That edge position  $(X, Y)$  propagates through the internal pipeline registers of Cordic block to be synchronized with  $(\theta)$  which is calculated by the Cordic using the gradient parameters  $(G_x, G_y)$  as explained in the following section. Eventually,  $(X, Y, \theta)$  are used by the Hough Transform block to detect the lanes as explained in chapter 5 - section 3.

## 5.2. Theta Detection Module

As Discussed in Chapter 3 section 2, The theta of each edge must be known to detect the edges in the region of interest and if they are in right or left region. Figure 28 shows the block diagram of this module, the output from the previous block which is the input of theta detection module block is  $(G_x, G_y)$  to detect theta, start signal,  $(X, Y)$  co-ordinates of edges which is buffered to be sent to the next block “Hough transform”.

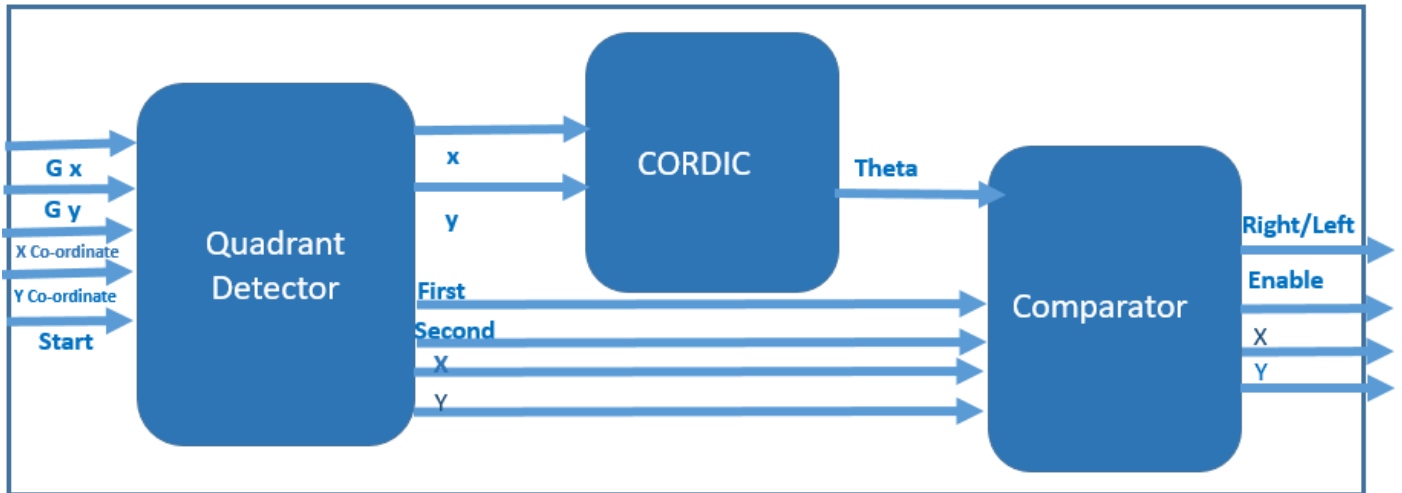


Figure 28: Theta Detection Module Block Diagram

### 5.2.1. Quadrant Detector

As discussed in chapter 3 the algorithm of CORDIC and how does it calculates angle from input vector, this operation is valid only in the first quadrant, CORDIC calculates angle between  $0^\circ$  and  $90^\circ$  only so a module to detect the quadrant of the edge which theta is calculated for. Quadrant detector takes edges signed and then gives absolute of the edge to the CORDIC, and a flag represents first or second quadrant to comparator module if first quadrant the output theta from CORDIC is correct, if second quadrant so we need to add  $90^\circ$  to the output theta of the CORDIC.

X and Y co-ordinates of the edge is buffered through shift registers in quadrant detector to be ready as an output to the Hough transform block with the whole module output.

### 5.2.2. CORDIC

The Implementation of the CORDIC is designed to be pipelined so we can change input every clock cycle, it takes number of cycles equals to number of iterations to calculate theta given x & y. CORDIC calculates theta by make many iterations till reaching acceptable error as discussed in 2-2, it is found that 10 iterations makes error acceptable as shown in table 1, random values of Gx and Gy are tested for number of iterations = {7,10,16}. Comparing the error in 10 iterations to reference model it is found that the error is only 1 degree which is acceptable in our design. The angles in the table are huge decimal number not  $45^\circ$ ,  $22.5^\circ$  and so on because the lookup table of the used tan values  $(2^{-i})$  are normalized by the value of the smallest angle tan inverse  $(2^{-15})$  to avoid any fixed point representations, First angle =  $\frac{\arctan(2^0)}{\arctan(2^{-15})}$ , Second angle =  $\frac{\arctan(2^1)}{\arctan(2^{-15})}$ , Third angle =  $\frac{\arctan(2^2)}{\arctan(2^{-15})}$  and so on as shown in the following figure 29 all values are integers.

Table 1: Theta values for random inputs and different iterations

Gx	Gy	Theta for 16 iterations	Theta for 10 iterations	Theta for 7 iterations
20	10	15514	15388	15516
-3	20	46922	46796	45900
3	1	10628	10556	9660
1	3	41066	40940	40044
2	-4	32844	33010	33904
1	1	26736	26210	25714

```
function [`THETA_BITS:0] tanangle;
input [3:0] i;
begin
  case (i)
    4'b0000: tanangle = 17'd25735 ; // 1/1
    4'b0001: tanangle = 17'd15192; // 1/2
    4'b0010: tanangle = 17'd8027; // 1/4
    4'b0011: tanangle = 17'd4075; // 1/8
    4'b0100: tanangle = 17'd2045; // 1/16
    4'b0101: tanangle = 17'd1024; // 1/32
    4'b0110: tanangle = 17'd512; // 1/64
    4'b0111: tanangle = 17'd256; // 1/128
    4'b1000: tanangle = 17'd128; // 1/256
    4'b1001: tanangle = 17'd64; // 1/512
    4'b1010: tanangle = 17'd32; // 1/1024
    4'b1011: tanangle = 17'd16; // 1/2048
    4'b1100: tanangle = 17'd8; // 1/4096
    4'b1101: tanangle = 17'd4; // 1/8192
    4'b1110: tanangle = 17'd2; // 1/16k
    4'b1111: tanangle = 17'd1; // 1/32k
  endcase
end
endfunction
```

Figure 29: Normalized values of  $\tan(\text{angles})$  used

### 5.2.3. The Comparator

This module takes the theta calculated from the CORDIC and the flags of first or second quadrant from the quadrant detector to decide if the calculated theta is in right region, left region or out of region of interest the values of minimum and maximum thetas are taken from Matlab software model.

There is a FIFO between this module and Hough transform module to manage communication between them, comparator has enable signal if the theta of the edge is out of region of interest nothing is written in the FIFO, if it is in the right region signal R=1, left region signal R=0. X co-ordinate, Y co-ordinate is although ready at the same cycle to be written in the FIFO.



### 5.3. Hough Transform

As referred in chapter 2 HT's function is to find the most likely line in the edged image to be the lane in the street. This section will discuss the hardware implementation of its two main stages:

- Hough Transform.
- Inverse Hough Transform.

#### 1- Hough Transform:

The function of this stage is to transform from  $(x, y)$  domain to  $(b, \theta)$  domain to achieve the equation  $b = x \cdot \cot(\theta) + y$ . It processes on every edge to calculate  $b$  and  $\theta$  by iterating on range of thetas by a certain step which will be explained later. After some latency from the pipeline at every clock edge a certain  $(b, \theta)$  is generated.

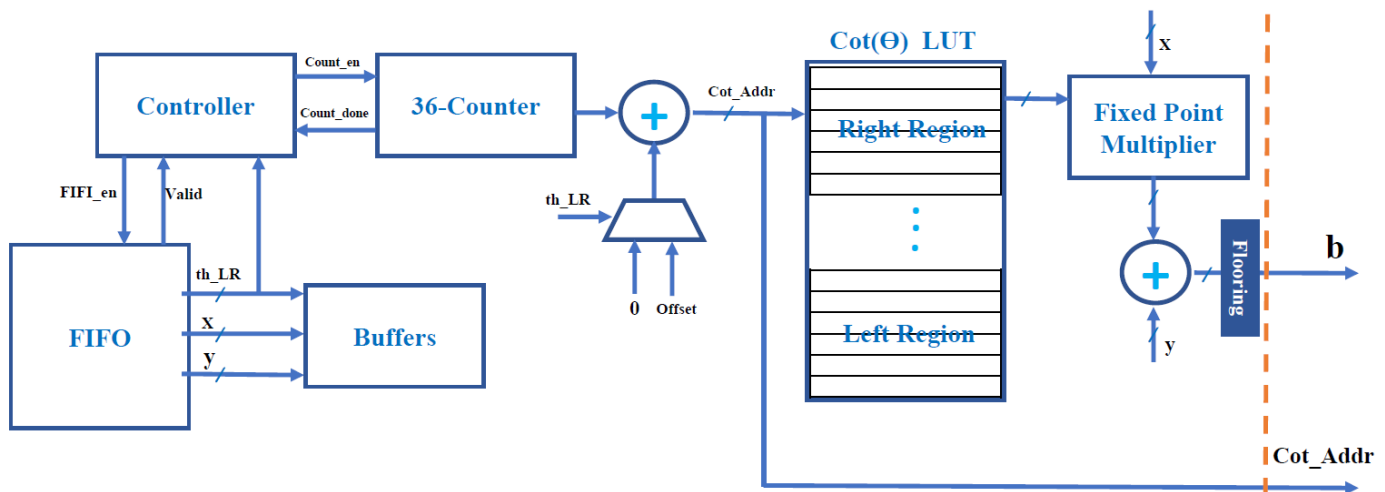


Figure 30: Hough Transform Stage Block Diagram

#### 5.3.1. FIFO

The start point of the stage is the FIFO where edges within the range of thetas that were explained in the Cordic section are stored. This FIFO is designed so that whenever an edge within the range comes from the Cordic a *write\_enable* control signal is high and the edge is stored with a flag called *Theta\_LR* indicates whether this edge is in the right or left region (right = 1, left = 0). Therefore, we have 3 FIFO blocks, one for X coordinate of the edge, one for the Y coordinate of the edge and one to store the *Theta\_LR* flag for each edge.

The FIFO is essentially an envelope logic around a synchronous 2-port RAM, where the read and write addresses of the RAM are not inputs, but rather they are stored in registers inside the FIFO since they are incremented automatically whenever data is read from or written into the FIFO.

There are 3 flags inside the FIFO, the Full flag, the Empty flag and the Valid flag. The Full flag is set to high when the all the registers inside the FIFO have been written into and the next register to be written into has not been read from yet, so when the Full flag is high the FIFO stops taking inputs from the Cordic stage. The second flag is the Empty flag, which is set to high when all the registers in the FIFO have already been read, so it is high when the next address to read from is the same address as the address we are to write in next, which means we have to wait for a write to occur once at least before we read. The Valid flag is the inverse of the Empty flag and it is exported as an output to the controller to notify the controller that the FIFO has at least 1 edge written into it and the Hough Transform unit can start processing this edge.



### 5.3.2. Controller

A simple FSM starts to read from the FIFO at the end of current edge processing if the valid flag is high (FIFO is not empty). It also starts the counter to read from the LUT which will be explained later.

### 5.3.3. LUT ( $\cot(\theta)$ )

As introduced in chapter 3 for every edge the equation  $b = x \cdot \cot(\theta) + y$  should be calculated where  $\theta$  takes a specific range depending on whether it's left or right region which is defined by the signal Theta\_LR.

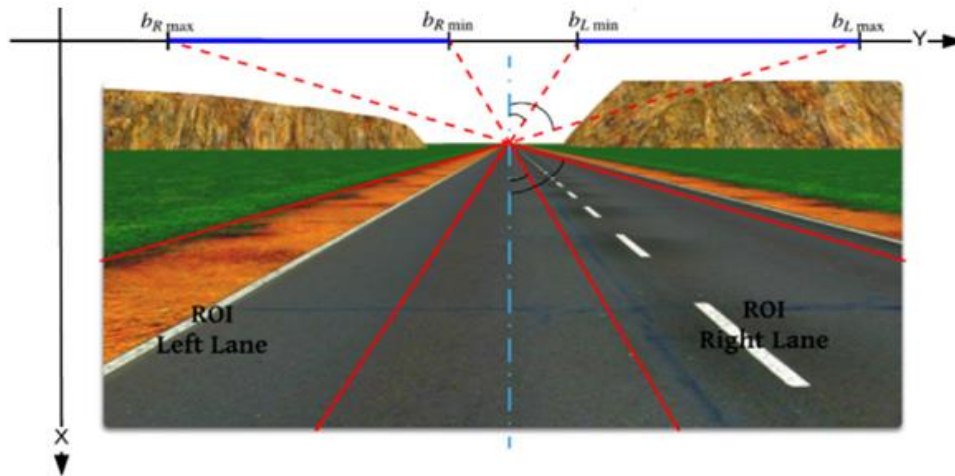


Figure 31: ROI, Range of  $b$  and  $\theta$

In this architecture the values of  $\cot(\theta)$  of both left and right region are stored in LUT in 24-bit fixed point representation. The integer part of the fixed point is 10-bit as the input image is 512\*512 and the fraction part (Q) needed some trial and error to determine the best number of bits to be represented in to achieve the required frequency with a good accuracy (increasing number of bits improves the accuracy but increases the delay in the path). At first it was represented by 8-bit but it wasn't accurate enough to distinguish between the two parts of the lane as shown in figure 32-a, then after trials to find the best accuracy taking area and delays into considerations it was found that 14-bit precision was the best compromise as shown in figure 32-b, the accuracy was significantly improved as it was identical to the golden reference (software model) and the desired frequency wasn't significantly degraded which will be discussed later in the results section.

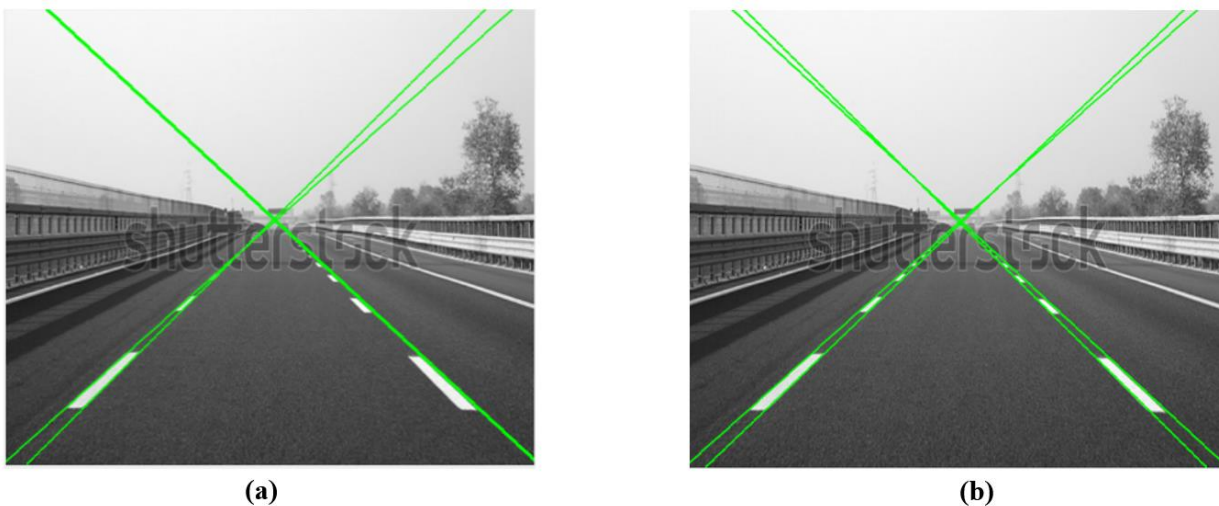


Figure 32: Output before and after increasing accuracy

The LUT contains both left and right regions  $\cot(\theta)$  values (36 values for the region with 1 degree step) an offset is added to select which region will be read from the LUT the signal Theta\_LR determines the value of the offset. All values are stored in absolute as all the right range is negative and all the left range is positive so the sign can be handled later, the equation  $\mathbf{b} = \mathbf{x} \cdot \mathbf{cot}(\theta) + \mathbf{y}$  can be rephrased to  $\mathbf{b} = \mathbf{y} \pm \mathbf{x} \cdot \mathbf{cot}(\theta)$  and the sign will be determined by the flag Theta\_LR. A 36-counter is started by the controller to iterate on the LUT and read the full range of the needed region.

### 5.3.4. Fixed-Point Multiplier

This fixed-point multiplier is unsigned as discussed before the sign will be handled by the addition operation. Both of the operands ( $\mathbf{x}$  and  $\mathbf{cot}(\theta)$ ) are 24-bit (10 integer and 14 Q-point) as  $\mathbf{x}$  is fixed-point extended, output is also the same as the operands as the normal output should be 48-bit but only 24-bit is needed because the full range of  $\mathbf{x}$  and  $\mathbf{cot}(\theta)$  is known so 24-bit is enough to represent the multiplication  $\mathbf{x} \cdot \mathbf{cot}(\theta)$ .



- Red are the ignored bits and green are the output bits.

After multiplication the addition operation takes place considering the sign. For the right region (Theta\_LR = 1) it will be a subtraction and for the left region (Theta\_LR = 0) it will be a normal addition. At the end of this stage the calculated value from the equation  $\mathbf{b} = \mathbf{y} \pm \mathbf{x} \cdot \mathbf{cot}(\theta)$  is floored because the next stage isn't interested in the fraction part as it's used in an address so only integer part is needed.

## 2- Inverse Hough Transform:

In this stage every line generated at the last stage defined by  $\mathbf{b}$  and  $\theta$  is checked whether it's in the region of interest (ROI) shown in figure 33, if it is not then it is discarded and if it is in the ROI then it will be counted as a vote, at the end the most voted two lines for both regions left and right will represent the lane.

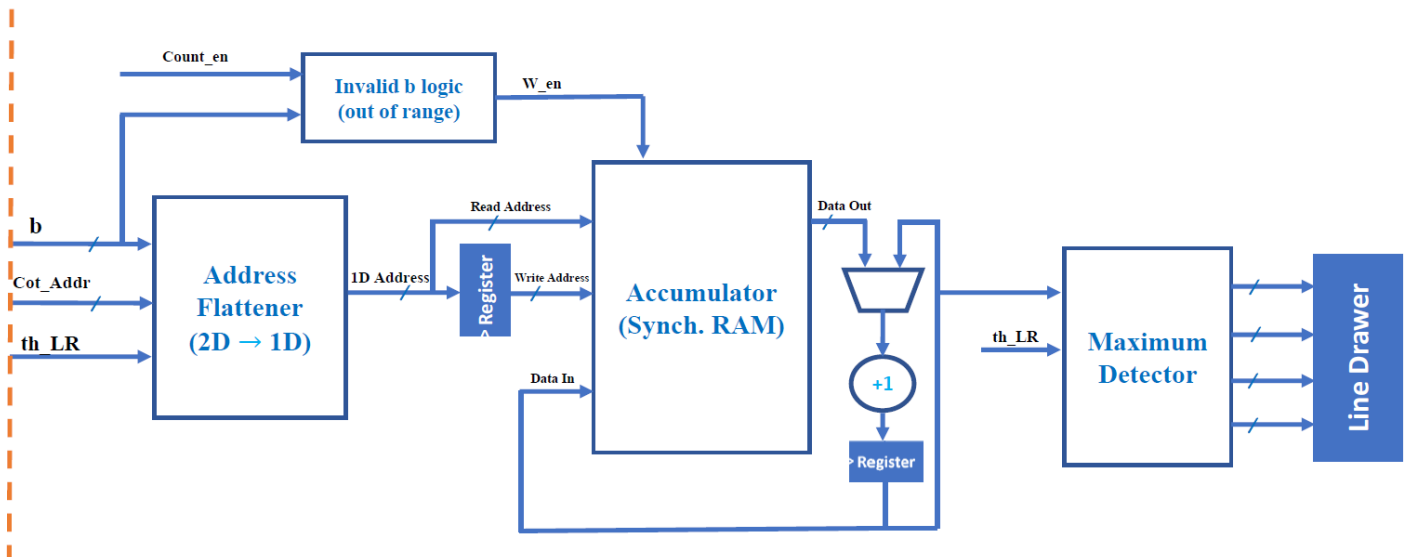


Figure 33: Inverse Hough Transform Block Diagram

### 5.3.5. Address Flattener

A line in Hough space is defined by two coordinates  $(b, \theta)$  and to vote for every line generated a 2D address memory is needed but there is no such a memory like that. So, the solution is to convert the 2D address to 1D address as shown in figure 34 and use the normal RAM as accumulator for voting.

To simplify the design of the rest of blocks Theta address (in the LUT) is used instead of  $\theta$  itself then to convert the 2D address to 1D address the following equations are used:

$$\text{- For right region: Addr} = \left( \frac{b - b_{\min R}}{b_{\text{step}}} \right) * n + \text{cot\_addr}$$

$$\text{- For left region: Addr} = \left( \frac{b - b_{\min L}}{b_{\text{step}}} \right) * n + \text{cot\_addr} + \text{offset}, \text{ Where:}$$

$n$  is number of thetas for one region.

$b_{\min R}$  and  $b_{\min L}$  are the minimum  $b$  for right and left region in the ROI.

$b_{\text{step}}$  is the step of  $b$  it can take values  $2^m$  and  $m$  is positive integer, In this design  $m = 1$

$\text{Cot\_addr}$  is  $\cot(\theta)$  address in the LUT.

$$\text{offset} = n(b_{\max} - b_{\min}) - n$$

These equations are implemented using arithmetic units (adders, multiplier and shifters).

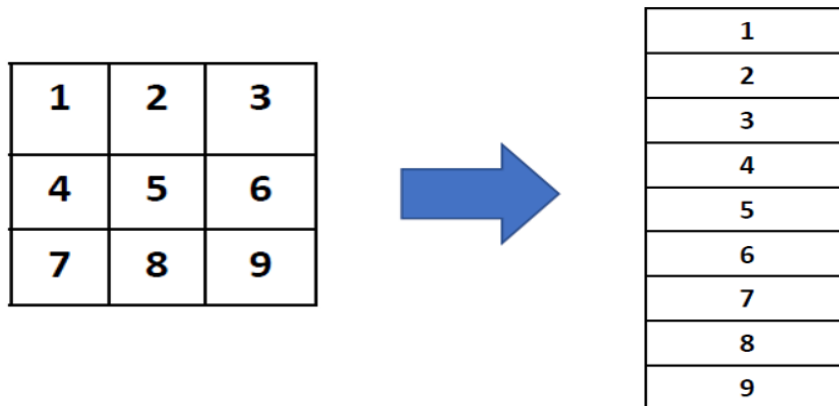


Figure 34: Address Flattener illustration

### 5.3.6. Accumulator

It's a two-port synchronous RAM that adds "one" to the read data (voting value) and store it again one cycle later. A bypass is added in case of reading the same address two consecutive cycles so that the new value of voting is accumulated instead of reading the old one from the RAM.

When the process of reading from the image has been completed and all the edges have been processed by the Hough Transform block we will have completed the voting for the possible lanes and they are all stored inside the accumulator and the next block will have stored the most voted for straight lines.

### **5.3.7. Maximum Detector**

The function of this block is to detect the maximum voting values for left and right region and for locate the lane accurately it is designed to detect the two maximum values for each region as shown in figure 32-b. It is implemented using comparators and simple logic gates.

### **5.3.8. Line Drawer**

When image reading is done and all edges are processed by HT a done signal is triggered and Line Drawer captures the coordinates of the lanes and transform them back to the normal domain (X, Y).

## Chapter 06: Formal Verification

Formal Verification (FV) is the use of tools that mathematically analyze the space of possible behaviors of a design, rather than computing results for particular values.

Where UVM tries specific values, an FV tool will look at the full space of possible simulations, it will use clever mathematical techniques to consider all their possible behaviors.

Three Formal Verification tools, provided by Siemens EDA, were used to make sure that all the bugs that are in our design were covered. The tools are as follows:

### 6.1. Questa Autocheck

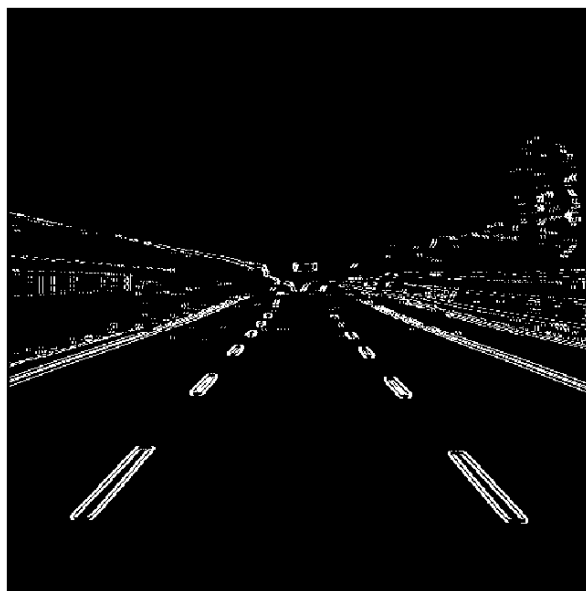
This is a static tool that checks possible issues in the design, Autocheck can be seen as a tool that checks general predefined assertions that are essential to be checked in all designs.

Some of the checks performed are:

- Index Illegal
- Block Unreachable
- Reg Stuck At  
and others

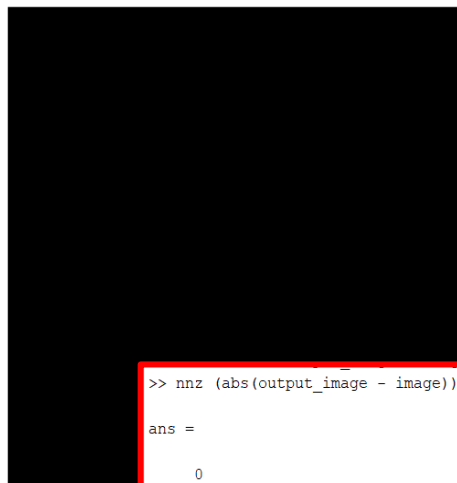
The tool helped in discovering bugs in the design that weren't going to be detected before when using testbench, this shows the importance of using that tool.

A major problem that was fixed using Autocheck was that the output image from the Edge Detection block was shifted from the output that was obtained from the software model. After fixing the warnings that came from the Autocheck tool the shifting that occurred in the output of the Edge Detection block was completely removed.



*Figure 35: The absolute difference between software and hardware outputs before formal verification*

In figure 35 it is seen that by subtracting the output image from software and our code there is a slight shift in the pixels. Figure 36 shows that the absolute difference between the images from both software and hardware now gives zero after fixing the issue given by Autocheck.



```
>> nnz (abs(output_image - image))
ans =
    0
```

Figure 36: The absolute difference between software and hardware outputs after formal verification

## 6.2. Questa Lint

This tool informs about potential issues that can occur due to the style of the written RTL, Lint gives issues such as:

- Concurrent Block with Duplicate Assign
  - MUX Select Constant
  - Flop Redundant
- And others

Most of the issues that the Lint tool gave a warning about related to the written code style which does not necessarily translate into bugs in how the design itself operates, but fixing some of these issues helped with the readability of the written RTL.

When the tool was first used, it gave a score to measure the quality of the written RTL, from Figure 37 it is seen that the score given was 90.4% which is considered not up to standards.



Figure 37: Quality score before fixing issues given by Lint tool

An example of an issue that the tool helped in fixing was the fact that there was a redundant flop in both the Gaussian and Sobel Filters, this flop was used in a Finite State Machine to control the flow of the blocks, after fixing the issue and removing the redundant flops it was seen that there was no need to create a Finite State Machine, this heavily improved the readability of the RTL and reduced the area since unnecessary registers were removed.

After fixing all the issues the tool was revisited to check the quality score, in figure 38 it is shown that the quality score was brought up to 100%.



Figure 38: Quality score after fixing issues given by Lint tool

## 6.3. Questa Propcheck

This tool was the main focus of the Formal Verification phase, this is the tool where the assertions, assumptions and covers are inserted as inputs so the behavior of the design can be checked. The former two tools checked for general behaviors that are common between designs that had to be fixed, but in Propcheck, properties and sequences that describe the specific behavior of the design were written and used.

When writing the properties to be checked, assumed and asserted the system was divided into 4 main blocks which are the Edge Detection Block, the FIFO, Hough Transform Block and Line Drawer Block.

### 6.3.1. Edge Detection Properties

The operation of the Edge Detection Block is that the input data valid signal remains high as long as the image is still being read from. Once the input data valid signal is high the write signals of the line buffers are set to high, each line buffer should read data from the input image for 512 cycles, when it is done reading then the next line buffer starts reading from the image for another 512 cycles, this is repeated for each of the 4 line buffers until the entire image is read. For this operation, an assertion was written for the first line buffer that it has to remain high for 512 cycles after the rising edge of the input data valid signal, this is the only assertion that is specific for a single line buffer, for the rest of the line buffers, including the first one, the two following assertions were written, the first assertion is that after the falling edge of the write enable of a line buffer while the input data valid signal is still high, then the next line buffer must remain up for 512 cycles, the other assertion is that once a line buffer write enable is set to low then it must remain low for 1536 cycles before being set to high again, to make sure that the 3 remaining buffers have finished reading from the image.

Since there is only 1-line buffer reading from the image at a time, then an assertion had to be written to make sure this property is true, so one-hot assertion for the write enable signals of the line buffers was written.

When the filters are reading from the line buffers, reading is started 2047 cycles after the input data valid signal was set to high, so an assertion was written for this property, it is important to note that the filters read from 3 line buffers in parallel, so the read enable signals of 3 line buffers are set to high at the same time, so an assertion was written to check that read enable of only 1-line buffer would be set to low at a time.

The mentioned assertions were written for both the Gaussian Filter and the Sobel Filter modules since they operate exactly the same way.

### 6.3.2. FIFO Properties

To recall the operation of the FIFO, it was inserted there since the Hough Transform unit requires multiple cycles to process each edge on its own, so there had to be a way to let the previous blocks store the edges that are to be processed by the Hough Transform unit. The FIFO can receive data from the Cordic unit as long as

it is not full, and once there is at least element in it then a valid signal is raised to high so the Hough Transform can send a read enable signal to the FIFO and start processing that edge.

For the FIFO, even if there are multiple elements that have been stored inside it, the Hough Transform Block can only read a signal each 36 cycles to be able to finish the processing of the current edge point. For this property an assertion was written that checks that once read enable signal was raised for the FIFO, it must remain low for at least 36 cycles.

Another property that has to be checked was that the FIFO read enable signal must only be raised when the FIFO sets the valid signal to high, so an assertion was written to check that property.

### 6.3.3. Hough Transform Unit Properties

For each edge position, represented by X and Y, the coordinates are multiplied by the Cotan of 36 angles depending on the region of interest it belongs to.

First, an assumption had to be written for the tool that the values of X and Y should be between 0 and 512, any other input will not be valid since this is the size of our image.

Inside the Hough Transform (HT) unit there is a counter that gives the index of the  $\text{Cot}(\theta)$ , this counter has to count for 36 cycles so the edge point can be multiplied with all the angles in the edge's region of interest and after the counter finishes a count done signal is raised to high. So, an assertion was written to check that this property is true.

Another assertion that was written was that once the FIFO read enable signal was set to high the counter count enable signal has to remain high for 36 cycles.

Another behavior specific to the HT unit is that if the output of the multiplication process (b) is out of the range of b that is inside the region of interest then the output b and  $\theta$  pair must be rejected so no vote values should be incremented in the accumulator, because this means that the straight line that was obtained from the multiplication process is not in the region of interest and it must be ignored. For this behavior two assertions were written, the first one is to check that if the b is bigger than the maximum b or smaller than the minimum b, this being the triggering sequence, the write enable of the accumulator must be low after 4 clock cycles, since this is the number of cycles it takes to reach the accumulator. The second assertion is that if the same triggering sequence occurs, the controller must raise the invalid b signal to high.

### 6.3.4. Line Drawer Properties

In the Line Drawer unit, the outputs of the previous stage, which are  $(b_L, \theta_L)$  and  $(b_R, \theta_R)$  pairs, are to be multiplied by all the X values of the image, which is from 0 to 512, so we can obtain the Y value for each X for each lane.

Two assumptions were made, the first is that the input valid signal must be high for only one clock cycle and remains low for the next 512 cycles, the second assumption is that once the valid signal is high, the inputs to the Line Drawer unit, which are the b and  $\text{Cot}(\theta)$  that will be used to obtain the positions of the line, must remain stable for 512 cycles.

There is a counter in the unit that counts the X values, the counter enable is set to high when the valid signal is set to high and the starting value of X is set to all zeros. So, an assertion was written to check that this behavior holds. Another assertion was written for when the counter enable is set to low, and that is when the value of X is all ones, which means we finished calculating for all values of X.



There is a read enable signal which is normally set to high in the unit, but it has to be set to low if the output Y is not in the range of the image, which is from 0 to 512, so if the MSB of the output Y is high then it means that the value is out of the range of the image and it has to be ignored. An assertion was written to check that this behavior holds.

### 6.3.5. Propcheck Results

All the assertions that were discussed have been inserted into the Propcheck tool along with the RTL so the tool tries to find an example where the assertions don't hold, and the result of running the tool was that all assertions that were written were proved to hold, which means that the system is behaving as intended. And with this the Formal Verification phase is concluded.

## Chapter 07: UVM

UVM is a methodology for the functional verification of digital hardware, primarily using simulation. UVM stands for the Universal Verification Methodology. It was created by Accellera based on the OVM (Open Verification Methodology). The targeted hardware or system to be verified would typically be described using Verilog, SystemVerilog, VHDL, or SystemC. This could be behavioral, RTL register transfer level, or gate level. The UVM allows you to implement this dynamically configurable testbench. This is done by defining object classes and then instantiating different objects for different tests. The core of UVM is to help Verification engineers develop modular, reusable, and scalable testing structures with the help of a library of classes that are supported by multiple EDA vendors like Mentor graphics (Siemens Business), Cadence, and Synopsis.

### 7.1. UVM environment

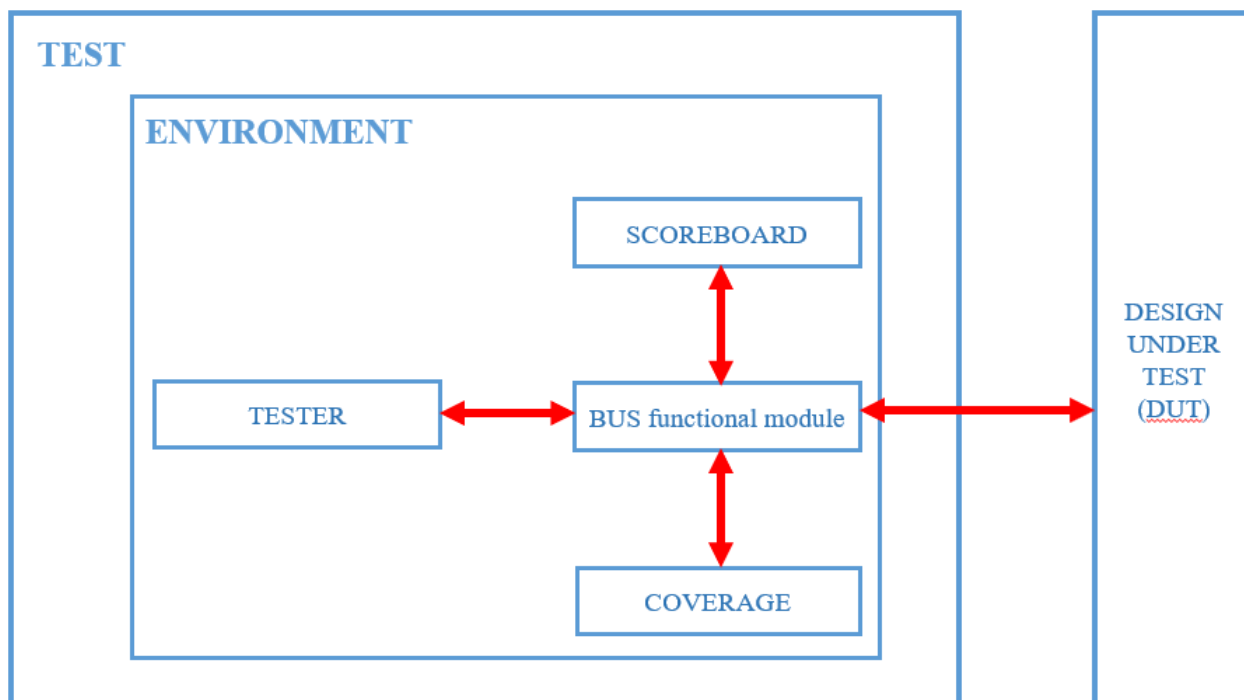


Figure 39: UVM Structure

#### 7.1.1. TESTER

Tester generates random constrained data. The other modules read this data through the bus functional module which connects all objects together.

#### 7.1.2. SCOREBOARD

The scoreboard reads the output from RTL to be either compared with a software model coded inside the scoreboard object (filters, Cordic and Hough Transform) or be dumped into text files to be compared with the reference model coded using MATLAB (edge detection).

#### 7.1.3. COVERAGE

The Coverage reads the random constrained from the tester to calculate the coverage percentage of how much of the possible inputs have been generated by the tester.

### 7.1.4. BUS FUNCTIONAL MODULE (BFM)

The module connects all objects together to allow full connectivity between objects.

### 7.1.5. DESIGN UNDER TEST (DUT)

The piece of code that the environment is built to test its functionality.

## 7.2. Testing process

- 1- Test internal modules used to build the main 3 blocks (Edge detection, theta detection, and Hough transform) as mentioned earlier.
- 2- Test the main 3 blocks individually.
- 3- Test the fully integrated system.

## 7.3. Internal Modules Testing

### 7.3.1. Edge Detection Modules

Edge detection is composed of filters (Sobel and Gauss filters), Line buffers, and a control unit. Filters have been tested at this level while the integration of filters and memories of this block has been tested in the higher level of testing block level as mentioned above. Input to the filters is 72 bits (9 pixels) and the output is 8 bits (1 pixel). To cover all possible values of 72 bits signal need huge computational power so corner cases and some random inputs have been covered to make sure the block works properly. More than one million random cases in addition to some corner cases all ones and all zeros have been covered. Figures 40 and 41 show the coverage for Gauss and Sobel filters.

Name	Coverage	Goal	% of Goal	Status
/Sobel_pkg/Sobel_...	100.00%			✓
TYPE sobel_pi...	100.00%	100	100.00...	✓
CVP sobel_...	100.00%	100	100.00...	✓
bin one...	5	1	100.00...	✓
bin othe...	1000500	1	100.00...	✓
bin zero...	2	1	100.00...	✓

Figure 40: Sobel filter coverage

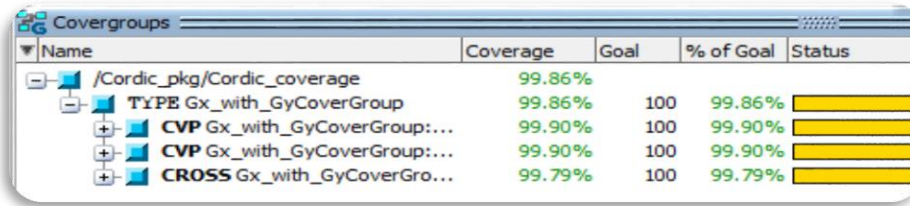
Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage	Comment
/Gauss_pkg/Gauss...		100.00%							
TYPE input_ra...		100.00%	100	100.00...	✓			auto(1)	
CVP input_...		100.00%	1	100.00...	✓				
bin zero...		1	1	100.00...	✓				
bin one...		1	1	100.00...	✓				
bin othe...		1000047	1	100.00...	✓				

Figure 41: Gauss filter coverage

### 7.3.2. Theta Detection Modules

Theta detection is composed of a Quadrant detector, Cordic, and, comparator. Cordic has been tested at this level of testing using UVM to find out the average error in angle calculated using Cordic block as it is an approximate way depends on the number of iterations as mentioned in the previous chapters. In figure 42 the

coverage of this block The coverage is 99.86% as inputs are constrained with a range of 0 up to 1023. Figure 43 shows the average error in angle calculated by Cordic.



Name	Coverage	Goal	% of Goal	Status
/Cordic_pkg/Cordic_coverage	99.86%			
TYPE Gx_with_GyCoverGroup	99.86%	100	99.86%	<div style="width: 99.86%;"></div>
CVP Gx_with_GyCoverGroup:...	99.90%	100	99.90%	<div style="width: 99.90%;"></div>
CROSS Gx_with_GyCoverGro...	99.79%	100	99.79%	<div style="width: 99.79%;"></div>

Figure 42: Cordic coverage

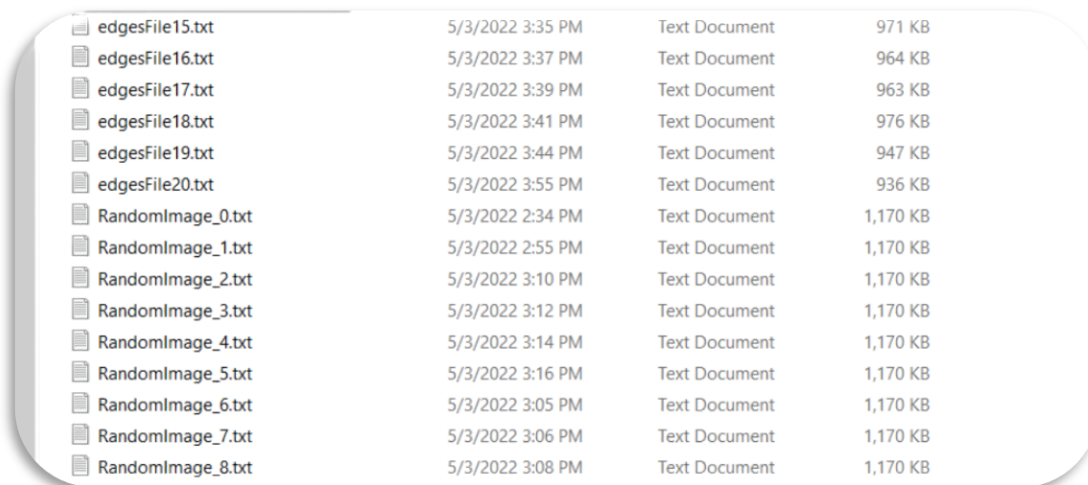
```
Average of the error in degrees = 0.3002568093385214
Average of the error in radian = 0.005237813229571984
errors above 0.0001 rad= 470196
errors above 0.0001 rad percnetage= 4.701966112555946
```

Figure 43: average error of Cordic

## 7.4. Main Blocks Testing

### 7.4.1. Edge Detection Block

At this level of testing, we test the Edge detection block after testing internal block filters using UVM and the other blocks using direct testing. We test this block by generating random images using the UVM environment feeding them to the edge detection block and the MATLAB software model and comparing outputs to find out if there are any errors. We have generated 1000 maps with about 20 million edges and the outputs are identical to the software and RTL. The approximations in filters have been also tested by comparing Output from RTL and the Gauss and Sobel filters of MATLAB in the 1000 maps error of around 1.2% of total edges which won't affect lane detection.



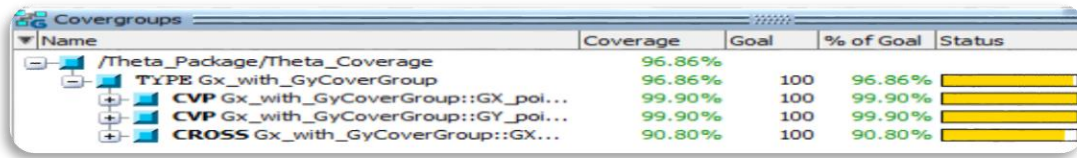
File Name	Date	Time	Type	Size
edgesFile15.txt	5/3/2022	3:35 PM	Text Document	971 KB
edgesFile16.txt	5/3/2022	3:37 PM	Text Document	964 KB
edgesFile17.txt	5/3/2022	3:39 PM	Text Document	963 KB
edgesFile18.txt	5/3/2022	3:41 PM	Text Document	976 KB
edgesFile19.txt	5/3/2022	3:44 PM	Text Document	947 KB
edgesFile20.txt	5/3/2022	3:55 PM	Text Document	936 KB
RandomImage_0.txt	5/3/2022	2:34 PM	Text Document	1,170 KB
RandomImage_1.txt	5/3/2022	2:55 PM	Text Document	1,170 KB
RandomImage_2.txt	5/3/2022	3:10 PM	Text Document	1,170 KB
RandomImage_3.txt	5/3/2022	3:12 PM	Text Document	1,170 KB
RandomImage_4.txt	5/3/2022	3:14 PM	Text Document	1,170 KB
RandomImage_5.txt	5/3/2022	3:16 PM	Text Document	1,170 KB
RandomImage_6.txt	5/3/2022	3:05 PM	Text Document	1,170 KB
RandomImage_7.txt	5/3/2022	3:06 PM	Text Document	1,170 KB
RandomImage_8.txt	5/3/2022	3:08 PM	Text Document	1,170 KB

Figure 44: Sample of generated maps

### 7.4.2. Theta Detection Block

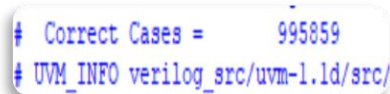
Theta detection's main function is to detect angles and then detect regions to which this pixel belongs. Generating random Values of GX and GY and feeding them to the block and comparing the region to which RTL decided that this pixel belongs with a software model which uses the tan inverse function of

SystemVerilog. The error percentage is about 0.41%. In figure 45 the coverage of this block is 96.86%. Figure 46 shows that theta detection block locates the correct region in 995859 case out of 1000000 case.



Name	Coverage	Goal	% of Goal	Status
/Theta_Package/Theta_Coverage	96.86%			
TYPE Gx_with_GyCoverGroup	96.86%	100	96.86%	<div style="width: 96.86%;"></div>
CVP Gx_with_GyCoverGroup::GX_pol...	99.90%	100	99.90%	<div style="width: 99.90%;"></div>
CVP Gx_with_GyCoverGroup::GY_pol...	99.90%	100	99.90%	<div style="width: 99.90%;"></div>
CROSS Gx_with_GyCoverGroup::GX...	90.80%	100	90.80%	<div style="width: 90.80%;"></div>

Figure 45: Theta detection coverage 1



```
# Correct Cases = 995859
# UVM_INFO verilog_src/uvm-1.1d/src/
```

Figure 46: Correct cases of theta detection

### 7.4.3. Hough transform (HT) Block

After testing Edge detection and Cordic, it's needed to test Hough Transform (Lane Detection algorithm).

The methodology of testing is as follows; It's known that HT gets the slopes ( $\cot(\theta_{R1})$ ,  $\cot(\theta_{R2})$ ,  $\cot(\theta_{L1})$ ,  $\cot(\theta_{L2})$ ), and y-interceptions ( $b_{L1}$ ,  $b_{L2}$ ,  $b_{R1}$ ,  $b_{R2}$ ) of the lanes. So, using the sequencer, random edges are generated and a percentage of them is handled so that 2 lines with random slopes and y-interceptions (which are known to the sequencer) exist and the same edges enter the HT block to get slopes and y-interceptions of the hardware block and then a comparison between SW and HW results takes place in the scoreboard.

Edges coordinates (x, y) are constrained to be within the image ( $512 > x$  (or  $y$ )  $> 2$ , for 512 x 512 image). Also, a percentage of the edges are entered as noise edges to test the system reliability. Finally, the resultant random image will be as figure 47



Figure 47: Input random edges

For the previous edges image example, the results are shown in figure 48:

```

Right Line1: slope= 0.699632 and b = 99
Left Line1: slope= -0.602018 and b = 402
Right Line2: slope= 0.699632 and b = 111
Left Line2: slope= -0.602018 and b = 414
b_L1 = 413
b_L2 = 401
b_R1 = 111
b_R2 = 100
cot_theta_L1 = 60
cot_theta_L2 = 60
cot_theta_R1 = 15
cot_theta_R2 = 15

```

Figure 48: The result of random edges example

The first four lines in figure 48 show the generated software lines (expected outputs) and then the hardware outputs are shown. From figure 48, it's shown that software and hardware are nearly identical.

Figures 49 shows different outputs for different seeds.

Right Line1: slope= 0.576788 and b = 71	Right Line1: slope= 1.326327 and b = 22	Left Line1: slope= -0.602018 and b = 382
Left Line1: slope= -1.001899 and b = 463	Left Line1: slope= -0.446189 and b = 510	Right Line2: slope= 1.326327 and b = 66
Right Line2: slope= 0.576788 and b = 83	Right Line2: slope= 1.326327 and b = 34	Left Line2: slope= -0.602018 and b = 394
Left Line2: slope= -1.001899 and b = 475	Left Line2: slope= -0.446189 and b = 522	b_L1 = 381
b_L1 = 475	b_L1 = 509	b_L2 = 393
b_L2 = 463	b_L2 = 522	b_R1 = 54
b_R1 = 90	b_R1 = 23	b_R2 = 66
b_R2 = 71	b_R2 = 34	cot_theta_L1 = 60
cot_theta_L1 = 46	cot_theta_L1 = 67	cot_theta_L2 = 60
cot_theta_L2 = 46	cot_theta_L2 = 67	cot_theta_R1 = 33
cot_theta_R1 = 8	cot_theta_R1 = 33	cot_theta_R2 = 33
cot_theta_R2 = 10	cot_theta_R2 = 33	

Figure 49: Output Results with Different Seeds

So, the algorithm is verified to work properly even if there is a significant percentage of noise, but to what extent? By trial and error, it was found that the algorithm can detect the lines of a noisy image with a noise edges percentage of approximately 7 % of the total edges which doesn't occur due to the Gaussian filter and also the filtration process of the Cordic block. Figure 50 shows the coverage of the Hough Transform block which ensures going into corner cases.

Component	Coverage	Count	Target	Status
/HT_pkg/HT_cover...	100.00%	100	100.00...	✓
TYPE input_ra...	100.00%	100	100.00...	✓
CVP input_...	100.00%	100	100.00...	✓
bin zero...	2	1	100.00...	✓
bin one...	2	1	100.00...	✓
bin othe...	93204	1	100.00...	✓
CVP input_...	100.00%	100	100.00...	✓
bin zero...	2	1	100.00...	✓
bin one...	2	1	100.00...	✓
bin othe...	93204	1	100.00...	✓

Figure 50: Hough Transform Coverage

## 7.5. Test the fully integrated system

Testing the fully integrated has been done using many images and comparing the output of RTL to the reference model instead of generating random images and extracting lanes from randomly generated images. It is very complex to ensure generation pixels so it is further to be detected as edges lie in the region of interest. This is a very complex task and doesn't reflect what will our design face concerning input images from real environment.

## Chapter 08: FPGA Implementation and Results

The following sections will introduce the utilization, frequency, and power of our design and then discuss the experimental work of the “Land Detection” System.

### 8.1. Edge Detection

Edge detection block as mentioned before detected lane edges to be fed into Theta detection block which filters the edges lays outside region of interest to decrease noise entering Hough transform block

#### 8.1.1. Edge detection utilization and frequency before adding ROM

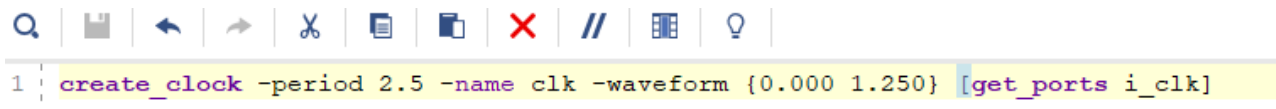
As discussed in chapter 5, the ROM and address generator are added as a method to read the input image only. Other approaches can be used, so the utilization for the implemented Edge Detection block without the ROM is shown in figure 51. The block can operate on frequency up to 400 MHz with 10% slack as shown in figure 52 and figure 53.

Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	F7 Muxes (115200)	Bonded IOB (360)	GLOBAL CLOCK BUFFERS (544)
ED_Top	3393	589	28	64	50	1
GF (GaussianFilter)	62	110	6	0	0	0
GIC (Gauss_imageControl)	1598	106	2	32	0	0
PC (PosCounters)	19	20	0	0	0	0
SF (SobelFilter)	113	247	18	0	0	0
SIC (Sobel_imageControl)	1601	106	2	32	0	0

Figure 51 : Edge Detection utilization before adding the ROM and address generator

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.230 ns	Worst Hold Slack (WHS): 0.025 ns	Worst Pulse Width Slack (WPWS): 0.718 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 15467	Total Number of Endpoints: 15467	Total Number of Endpoints: 2513

Figure 52 : Edge Detection max frequency before adding the ROM and address generator



```
1 create_clock -period 2.5 -name clk -waveform {0.000 1.250} [get_ports i_clk]
```

Figure 53 : Edge Detection timing constrains before adding the ROM and address generator

#### 8.1.2. Edge detection utilization and frequency after adding the ROM

After adding the ROM and the address generator the frequency was degraded from 400 MHz to 333 MHz as shown in figure 55 and 56. That was an accepted degradation as even when operating at this frequency a 512\*512 image processing is done in 0.7865 ms, which is equivalent to 1270 frame per second if operating on a stream of pictures. The utilization after adding the ROM and address generator is shown in figure 54.



Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	F7 Muxes (115200)	Block RAM Tile (312)	Bonded IOB (360)	GLOBAL CLOCK BUFFERS (544)
ED_Top	3458	652	31	72	64	50	1
GF (GaussianFilter)	62	110	6	0	0	0	0
GIC (Gauss_imageControl)	1598	106	2	32	0	0	0
PC (PosCounters)	19	20	0	0	0	0	0
ROM_Counter (fifo_counter)	49	49	3	0	0	0	0
SF (SobelFilter)	113	247	18	0	0	0	0
SIC (Sobel_imageControl)	1601	106	2	32	0	0	0

Figure 54: Edge Detection utilization after adding the ROM and address generator

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.199 ns	Worst Hold Slack (WHS): 0.019 ns	Worst Pulse Width Slack (WPWS): 0.968 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 15646	Total Number of Endpoints: 15646	Total Number of Endpoints: 2616

Figure 55 : Edge Detection max frequency after adding the ROM and address generator

```

1 create_clock -period 3 -name clk -waveform {0.000 1.5} [get_ports i_clk]

```

Figure 56 : Edge Detection timing constrains after adding the ROM and address generator

## 8.2. Theta Detection Module

Theta detection module as mentioned before take the edges from edge detection module and determine whether this edge is in the right region, left region or out of region of interest, here is the utilization shown in figure 57, Theta detection module can operate by maximum frequency 495 MHZ as shown in figure 58

Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	Bonded IOB (360)	GLOBAL CLOCK BUFFERS (544)
TopModule	452	430	62	69	1
Comparator1 (Comparator)	33	20	6	0	0
cordic1 (cordic)	317	290	52	0	0
Quadrant_Detector1 (Quadrant_Detector)	102	75	4	0	0

Figure 57: Theta Detection Module Utilization

### Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 2.982 ns	Worst Hold Slack (WHS): 0.030 ns	Worst Pulse Width Slack (WPWS): 1.968 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 408	Total Number of Endpoints: 408	Total Number of Endpoints: 456

```
create_clock -period 5.000 -name clk -waveform {0.000 2.500} -add [get_ports CLK]
```

Figure 58: Maximum frequency of theta detection module block

### 8.3. Hough Transform

As discussed in chapter 5, HT is the algorithm concerned with detecting the road lanes. The utilization for the implemented HT block is shown in figure 59. The block can operate on frequency up to 400 MHz with 10% slack as shown in figure 60 and figure 61.

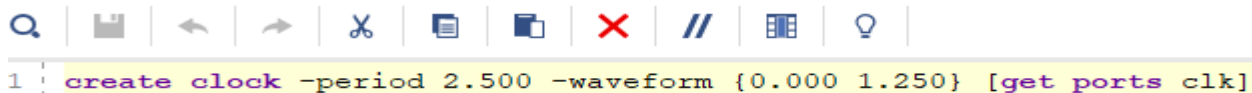
Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	F7 Muxes (115200)	Block RAM Tile (312)	DSPs (1728)	Bonded IOB (360)	GLOBAL CLOCK BUFFERS (544)
HT	237	389	16	14	5	3	113	1
accumulator (SynchMem)	21	0	0	0	5	0	0	0
Controller (control)	0	3	0	0	0	0	0	0
Counter (n_Counter)	54	10	0	14	0	0	0	0
Max_L (Max_Val_Cap)	31	92	3	0	0	0	0	0
Max_R (Max_Val_Cap_0)	30	92	3	0	0	0	0	0
mul (qmult)	10	0	2	0	0	2	0	0
Two_2_One_dimension (AddressConverter)	29	34	4	0	0	1	0	0

Figure 59: Hough Transform Utilization

#### Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.223 ns	Worst Hold Slack (WHS): 0.021 ns	Worst Pulse Width Slack (WPWS): 0.708 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 838	Total Number of Endpoints: 838	Total Number of Endpoints: 440

Figure 60: Hough Transform Timing Report



```
1 create_clock -period 2.500 -waveform {0.000 1.250} [get_ports clk]
```

Figure 61: Hough Transform Timing Constrains

### 8.4. Lane Detection Reports:

Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	F7 Muxes (115200)	F8 Muxes (57600)	CLB (28800)	LUT as Logic (230400)	LUT as Memory (101760)	Block RAM Tile (312)	URAM (96)	Bonded IOB (360)
LD_Top	4154	1603	144	232	16	825	2125	2029	1603	71.5	3
ED_Top (ED_Top)	1192	652	31	200	0	312	1192	0	64	0	0
HT_Top (HT_Top)	1042	907	113	32	16	267	933	109	7.5	0	0

Figure 62: Total Lane Detection Utilization

#### Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.341 ns	Worst Hold Slack (WHS): 0.003 ns	Worst Pulse Width Slack (WPWS): 1.358 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 19736	Total Number of Endpoints: 19736	Total Number of Endpoints: 3723

**All user specified timing constraints are met.**

Figure 63: Lane Detection Timing Report

```
1 create_clock -period 3.800 -name clk -waveform {0.000 1.900} -add [get_ports i_clk]
```

Figure 64: Lane Detection Timing Constrains

Figure 62 shows the utilization for the implemented Lane Detection System. The block can operate on frequency up to 263 MHz with 10% slack as shown in figure 63 and figure 64.

## 8.5. Experimental Work and FPGA Results:

After presenting the algorithm, its implementation and hardware resources it's shown that the proposed FPGA-based HT architecture for straight lane detection presents several advantages which can be identified in three points:

- The simplification of the equation of mapping of HT.
- The reduction of the used on-chip memory for the accumulator implementation.
- The inverse HT to map polar to cartesian coordinates which can be performed with no further hardware cost.

The performance of the designed straight lane detection method is validated by an implementation on the Zynq Ultra Scale FPGA of Zynq UltraScale+ ZCU104 Evaluation Board and different types of simulation (Behavioral – Post-Synthesis – Post Implementation) using Vivado 2019.1 and results are visualized using MATLAB.

Some available online images are used as data set and they are chosen so that the vanishing point appears centered across the center of the image. Images with a resolution of 512 x 512 pixels were used for the test purpose. ROIs of Y-intercept and  $\theta$  were defined statistically by testing the algorithm on various images from the database. To be a pixel on the right lane boundaries, the edge orientation  $\theta$  of the pixel should be in the range of  $-70^\circ$  –  $-35^\circ$  while Y-intercept varies from -52 to 180. For the left lane boundaries, the edge orientation  $\theta$  varies between  $-145^\circ$  and  $-110^\circ$  while Y-intercept varies from 336 to 566. So, the ROI will be as shown in figure 65.

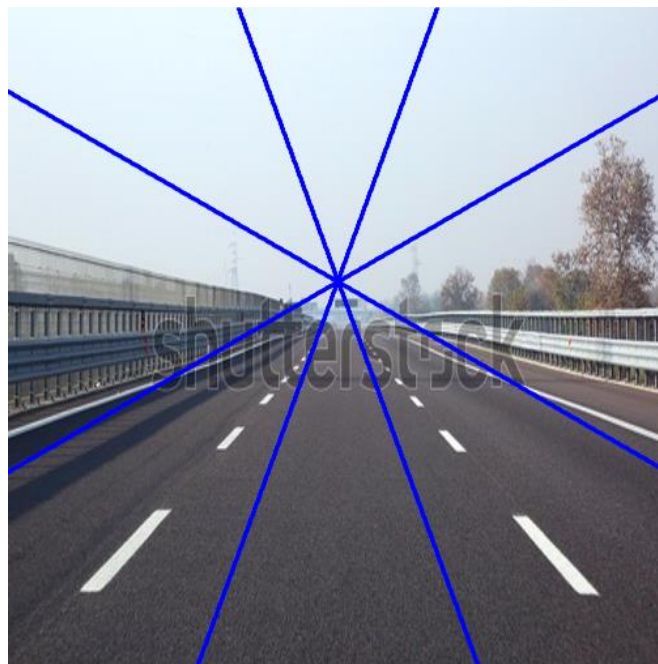


Figure 65: Shows the Chosen Region of Interest (ROI)

The resolution of the parameters of the accumulator  $\theta$  and Y-intercept are defined as  $1^\circ$  and 2-pixel units, respectively. The proposed implementation was tested on a variety of images with various illumination and conditions of the road scene, such as road type (urban street, highway), road condition, difficult weather conditions, shadowing, poor line paintings, day and night. From table 3, It can be shown visually that the implemented architecture detects effectively the straight lane lines in most of the cases (shadows, objects, cars). In this architecture, the lines which do not belong to the left or right lane boundaries, are filtered and not processed at the HT stage. The time between starting of processing of the input picture until the slopes and intersections of the lanes are obtained are approximately 262405 cycles which means if the LD operates on 263 MHz, a 1002 frame per second can be achieved.

## 8.6. Error Calculations

The error resulting from the fixed-point precision used in the introduced hardware implementation compared with software model (Golden Reference) which uses floating point is calculated by comparing every point (pixel) position of the detected lane with the software model output shown in the following equations.

$$y = b - x \cdot \cot(\theta) \quad \text{where } x = 1:512$$

$$\text{pixel\_error} = \frac{|y_{HW} - y_{SW}|}{y_{SW}} * 100 \%$$

$$\text{image\_error} = \frac{1}{512} \left( \sum_{x=1}^{512} \text{pixel\_error}(x) \right)$$

Table 2: The average error of hardware model

Image number	Image error %
1	1.0665
2	1.1366
3	1.3371
4	0.2840
5	2.2905
6	3.4664
7	1.3029
8	1.0121
9	0.8077
<b>Average Error</b>	1.41153

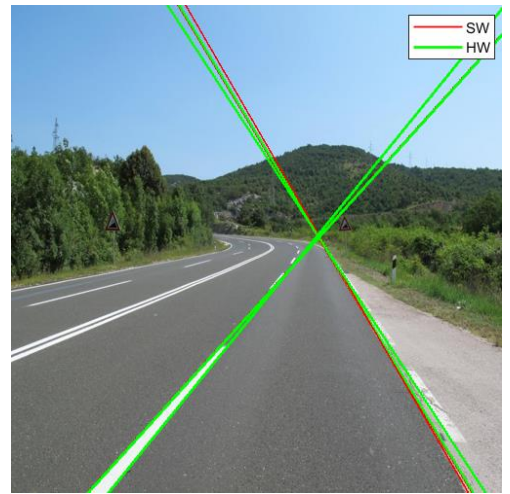
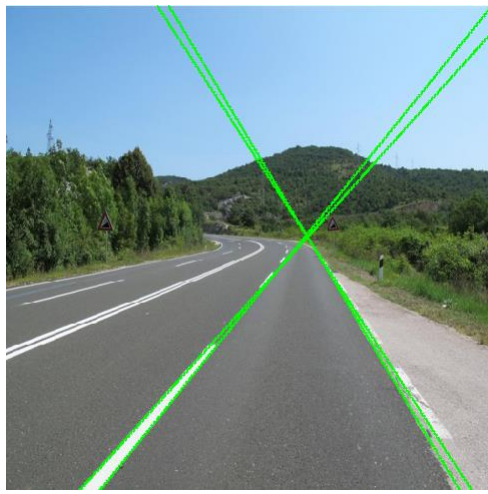
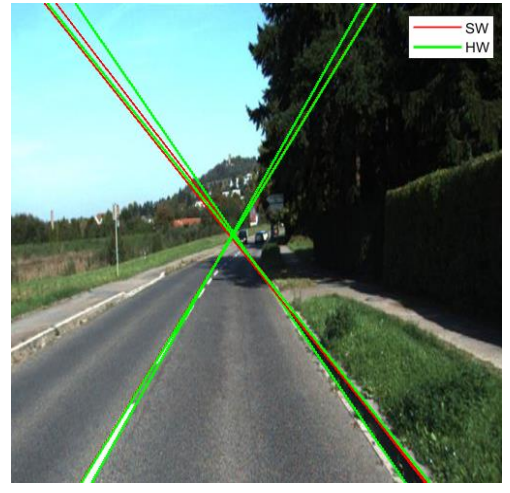
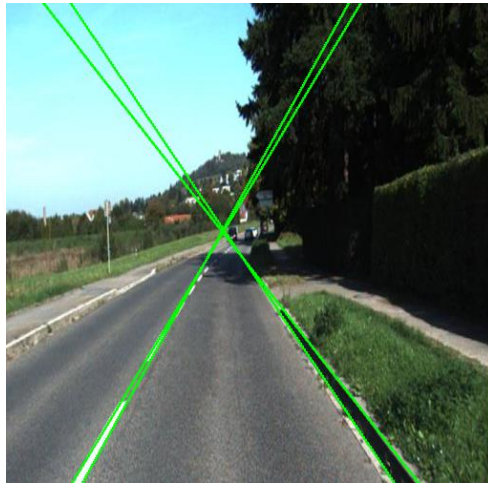
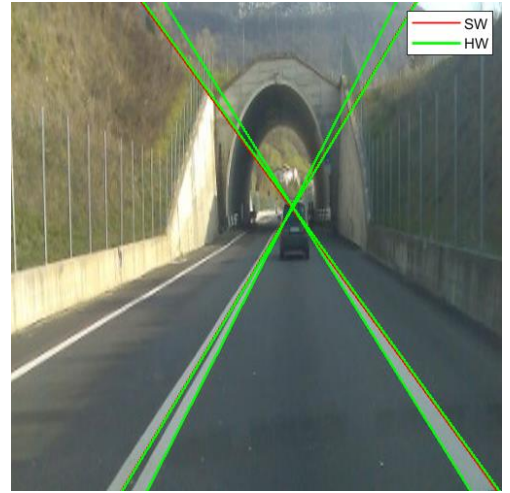
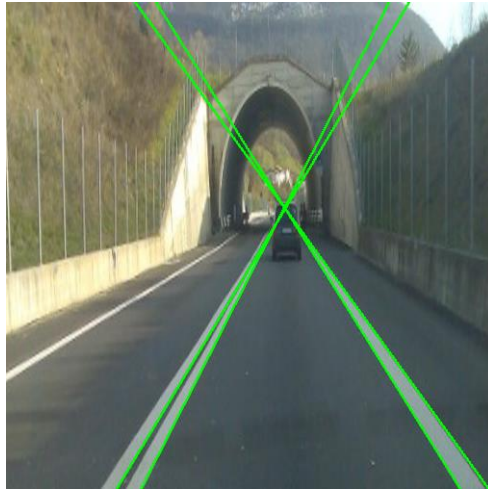
And this is what ensures the robustness and high performance of Hough transform algorithm in different road conditions.



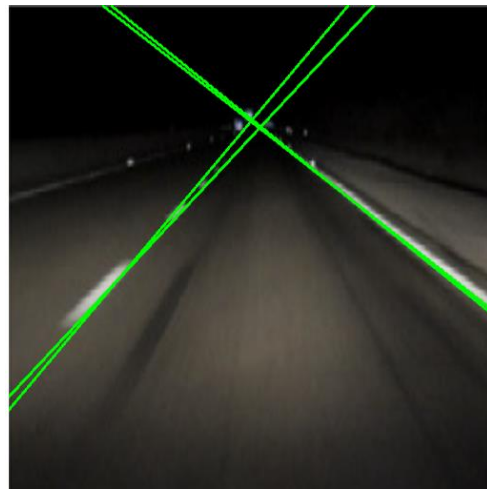
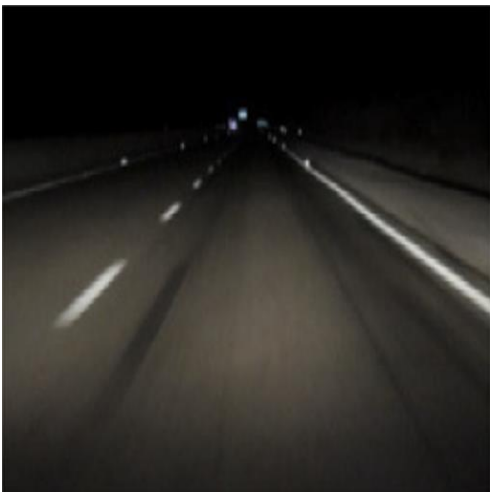
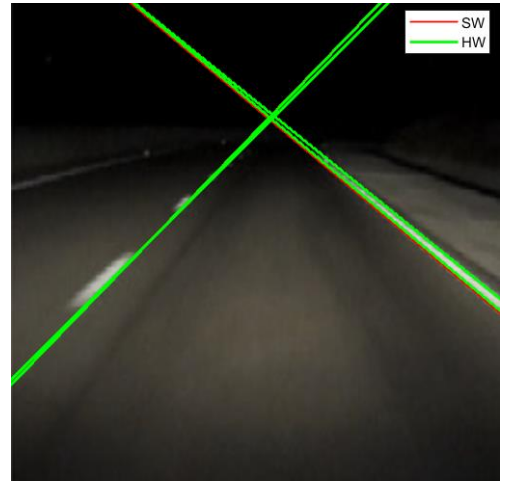
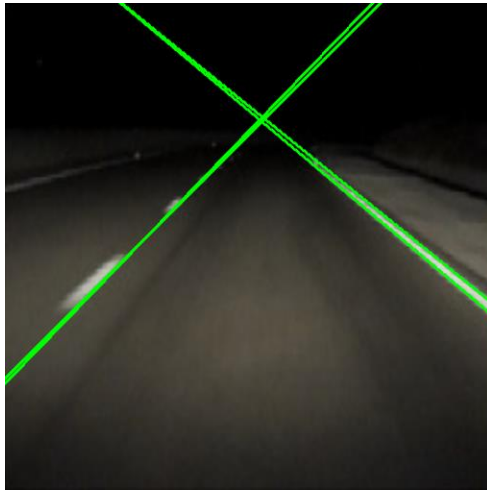
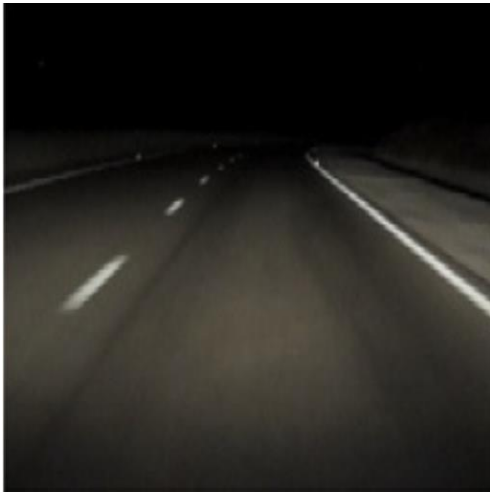
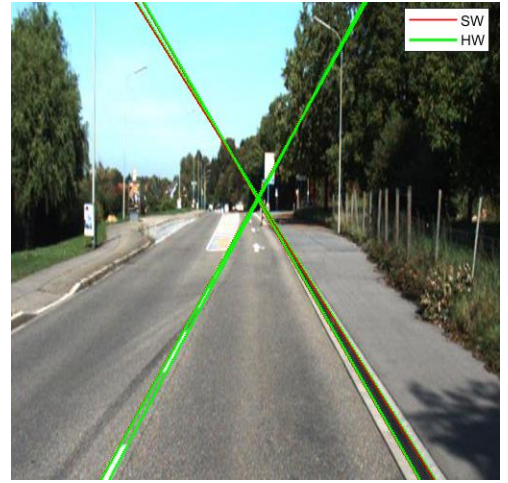
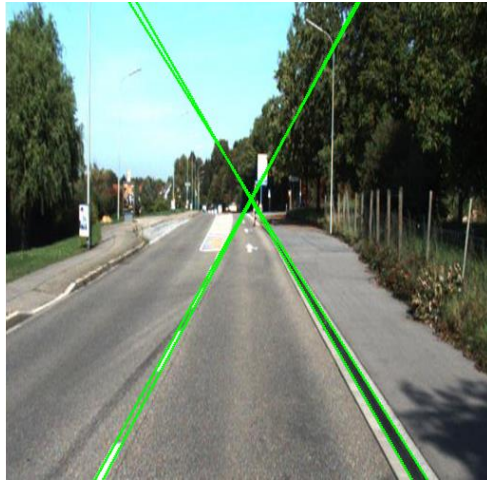
Table 3: Lane detection results in various conditions

Original Image	Hardware Results	Hardware VS Software









## Chapter 09: Functional Safety

### 9.1. Introduction

There are many ways through which a device could fail or stop working, this includes human errors, systematic error, hardware failures and operational/environmental stress. The probability of occurrence of these failures increases as time moves on and as we move towards the life-time of the device.

In safety critical systems, the event of failure will most likely put people's life at risk, safety critical systems include automobiles, planes and medical devices. It is clear that if one of these systems fail it may cause health hazards to the users of these systems.

Functional Safety (FuSa) is a way to be able to allow the device to work through those failures, as it provides safety mechanisms which make the device immune to failures. These safety mechanisms include duplication, triplication, parity or memory ECC. The goal that these safety mechanisms try to achieve is to discover that an error has occurred and even recover from it.

This is why functional safety is an important step for safety critical systems, and this is why it was important to implement it in the system, as a failure in the lane detection system can lead to more risk of drivers leaving their lanes unintentionally which could lead to car accidents that threaten the lives of the drivers and the passengers.

There are many different safety standards for functional safety. For the lane detection system, the safety standard followed was the ISO 26262 standard. This is the safety standard of the automotive industry, as it covers electric and electronic systems in production vehicles. This standard uses Automotive Safety Integrity Levels (ASIL A to D) to measure the safety level of the design. ASIL measurements depend on the probability of exposure (denoted by E), controllability by driver (denoted by C) and severity of failure (denoted by S).

Table 4 shows the ASIL level corresponding to a hazardous event depending on the previously mentioned parameters, QM refers to Quality Management, which means that the risk of the corresponding hazardous event does not require safety measures. Initially, the targeted ASIL level for lane detection is C.

Table 4: The ASIL levels corresponding to certain hazardous events

Severity	Probability	C1	C2	C3
S1	E0	QM	QM	QM
	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E0	QM	QM	QM
	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E0	QM	QM	QM
	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D



There are multiple steps that have to be made to be able to implement safety mechanisms onto the design. First, there are some safety metrics that have to be calculated by checking the coverage provided by inserting different safety mechanisms onto different blocks. After the safety metrics are calculated and finding out the proper methods that help in achieving the safety targets, the next step is to insert the safety mechanisms in the design and check if the device is still working correctly. After inserting the safety mechanisms, the safety metrics have to be recalculated to make sure that after the insertions the design still meets the safety target, if it is found out that the safety targets are no longer met, then the step should be repeated with additional safety mechanisms or the safety mechanisms should be modified. The last step involves performing a fault campaign onto the device, which means subjecting the device to a great number of failures to test the effectiveness of the safety mechanisms that were implemented. If the final step is passed this means that the functional safety phase has been completed successfully.

## 9.2. Functional Safety Tools

To be able to implement functional safety, Siemens EDA provided the necessary tools to be used. Four tools were used in serial, the tools used were SafetyScope, RadioScope, Annealer and KaleidoScope.

### 9.2.1. SafetyScope

This tool is the first step in the functional safety phase, here we have to specify the safety mechanisms for each block so we can calculate certain variables that are important to the functional safety phase, the tool takes the RTL as input, analyzes it and gives outputs that determine what safety mechanisms are required to meet the safety targets.

The parameters calculated are FIT, which stands for Failure In Time, and DC, which stands for Diagnostic Coverage. FIT measures the hardware random defect rate of the permanent and transient errors in design, which means it measures the susceptibility of the silicon and package to incur random hardware faults, while DC measures the numbers for the permanent and transient faults, in other words it is the effectiveness of the Safety Mechanism (SM) to detect a fault.

First step is to estimate DC using proposed safety mechanisms, once a suitable number is obtained for the DC, the next step is to proceed to the next tool to synthesize the safety mechanisms, insert them into the design, then come back to SafetyScope to calculate FIT and make sure the required safety targets are met.

1 FIT amounts to 1 failure/billion hours. Calculating FIT without inserting the safety mechanisms shows the failure rate without safety mechanisms, so it can then be compared to the failure rate after inserting safety mechanisms so their effect can be measured.

The design was separated into 3 major blocks which are Edge Detection, Cordic Block and Hough Transform Block, safety analysis was performed on each block individually, and for each block these safety mechanisms were used:

- 1- Edge Detection Block: Address Generator, Gauss Control Unit, Sobel Control Unit and Positions Counter are critical blocks, any failure in any of these modules will result in failure of the whole edge detection process, so a Duplication Safety Mechanism is used.
- 2- Cordic Block: The whole block is very critical and also utilizes low resources, so Duplication Safety Mechanism is used.
- 3- Hough Transform Block: There are critical blocks in HT such as; Accumulator, Address Converter, Maximum Detector (Right and Left Lanes) and LUT counter which are chosen to be duplicated. Also, it's thought that the most critical block in HT is the controller which handles the reading and processing of the edges so triplication is chosen as a Safety Mechanism.

The calculated DC resulting from the proposed safety mechanisms was 96%. FIT was not calculated, as it is needed to synthesize the safety mechanisms first, insert them into the design, then insert the new RTL into SafetyScope again to determine the FIT. To synthesize the safety mechanisms, the next tool, Annealer, is used.

### 9.2.2. Annealer

In this tool, a command is used to insert specific safety mechanisms for specific blocks. This tool inserts safety mechanisms at macro level (instance level). The tool not only duplicates or triplicates the specified blocks, it also creates control and observe blocks to manage the flow of data between the duplicated or triplicated blocks. These blocks generate error outputs that show if an error has occurred.

For duplication, Annealer tool creates 2 instances of the top design, it then makes appropriate connections for all the outputs and inputs between the first and second instance along with the checker block. So, if the inputs to the checker blocks are different it means that an error has occurred.

For triplication, 3 instances are created of the design, but a majority vote is used to determine the correct output. This is done by comparing the 3 instances output, and consider the output of at least 2 instances.

After the safety mechanisms from the previous section are synthesized using the tool, the output RTL from the tool had to be inserted into the design. The new RTL for each block contained new inputs and outputs depending on the used safety mechanism associated with it. The design RTL had to be changed to accommodate for the new signals that appeared in the new blocks.

When using the new RTL, testbenches were made to make sure that the functionality of the design blocks didn't change, and the outputs showed that the functionality of the system has remained unchanged which means that the safety synthesis step was completed without errors. At first, the testbenches were performed for each block alone, then a testbench was performed for the whole system.

As an example, Figure 66 shows Gauss Control Unit (*Gauss\_CU*) wrapper implemented by Annealer after applying duplication safety mechanism. As shown, another instance *u1\_Gauss\_CU* is added by Annealer) and the block *ATD\_MOD\_fn\_safte\_chk\_Gauss\_CU* is inserted to control safety signals.

### 9.2.3. RadioScope

This tool function is the same as Annealer, but it operates on Flip-Flop level. So, it can also apply parity safety mechanism which is usually used with memories to assure that the output data is the same as the input data and no faults occurred during propagation through the Flip-Flop.

### 9.2.4. KaleidoScope

This tool is used to inject faults into the design, it manages the fault campaigns that test the circuit. This step is referred to as Safety Verification. The tool injects faults into the safety-critical nodes in the design to determine if the safety mechanisms detect the faults.

## 9.3. Results

Tables 5 and 6 shows the full utilization, frequency and power of the 3 main blocks (Edge Detection with ROM, Cordic and Hough Transform) before and after applying safety mechanisms discussed in section 9.2.1.

From these results, it is obvious that there is some tradeoff between functional safety and the performance of the design which is expected due to the added modules which utilize more resources. That increases the power consumption and degrades the maximum frequency. But the degradation is tolerable in front of the huge benefits of functional safety, specially in a safety critical application as Lane Detection.

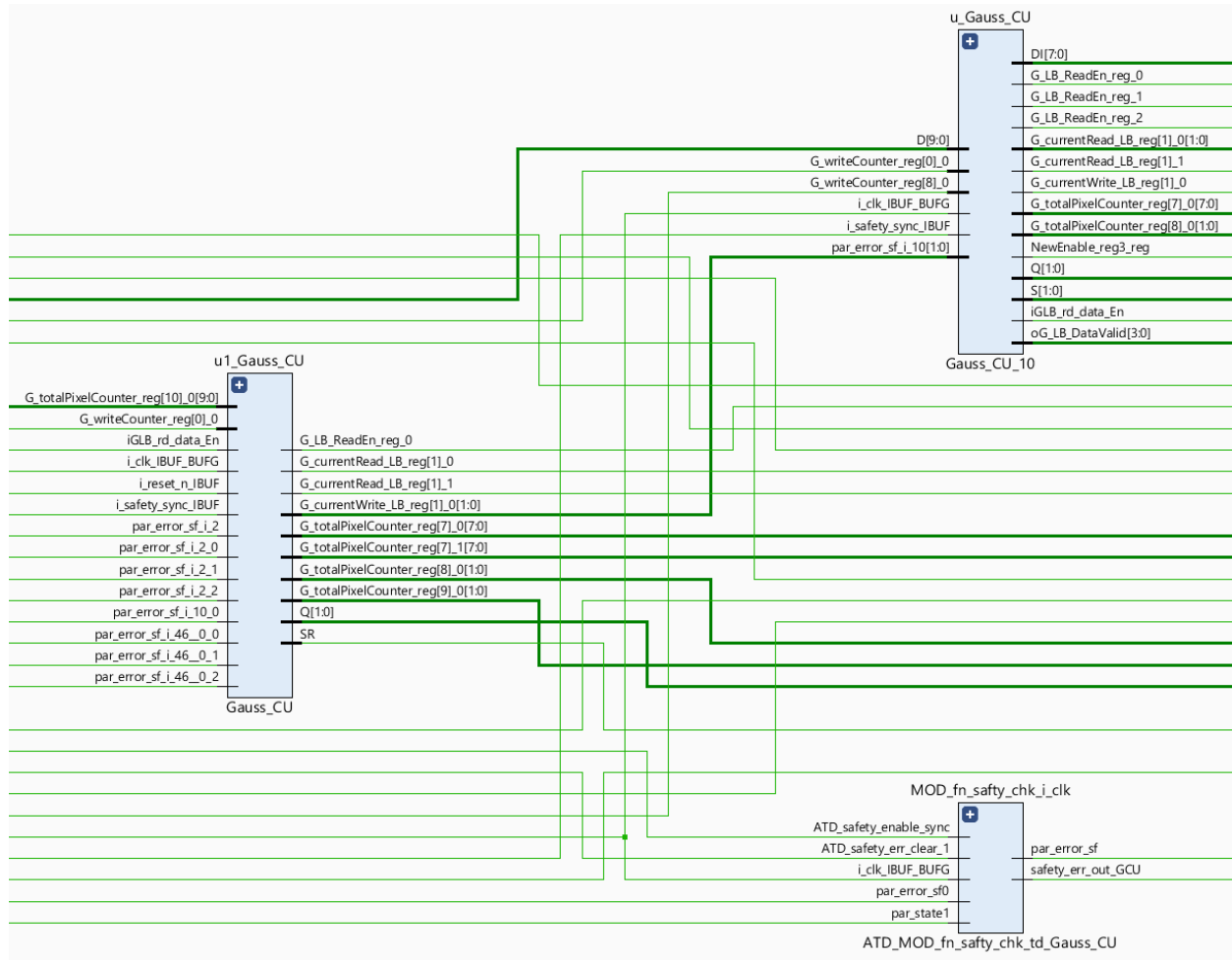


Figure 66: Gauss\_CU module after applying duplication safety mechanism using Annealer

Table 5: Full Utilization of the 3 main blocks before and after applying safety mechanisms

Block	Before FuSa			After FuSa		
	LUTs	Registers	BRAMs	LUTs	Registers	BRAMs
Edge Detection	3458	652	64	3976	774	64
Cordic	452	430	0	728	447	0
Hough Transform	237	389	5	436	646	9

Table 6: Max frequency (with 10% slack) and dynamic power before and after applying safety mechanisms

Block	Before FuSa		After FuSa	
	Frequency	Dynamic Power	Frequency	Dynamic Power
Edge Detection	333 MHz	0.1 W (14%)	300 MHz	0.105 W (15%)
Cordic	611 MHz	0.090 W (13%)	581 MHz	0.111 W (16%)
Hough Transform	400 MHz	0.092 W (13%)	274 MHz	0.096 W (14%)

## Conclusion

This document has discussed why the lane detection system was selected as the safety-critical automotive system to be designed. The document also discussed why the Hough Transform algorithm was chosen and the architecture used for it. After that the Edge Detection and Cordic algorithms were discussed.

After the algorithms were explained, the software model created on MATLAB was explained in detail, the software model was created to obtain a golden reference for the output as well as to be used for parameter calibration. After the software model was completed, the next step was to start designing the hardware blocks to begin the hardware implementation of the algorithms. The first block to be discussed was the Edge Detection block, which included line buffers, Gaussian and Sobel filters and other control blocks. The next block was the Theta Detection block, which included the quadrant detector, the cordic unit and a comparator. The last block was the Hough Transform block, which included a controller, a counter, a look up table for the values of  $\cot(\theta)$ , a fixed-point multiplier, an address flattener, an accumulator and a maximum detector block. Each block was explained in detail in its designated section.

After the design was completed, the next step was verification, both UVM and Formal Verification were used to verify the behavior of the design. Both these methods were used to be able to cover as much bugs as possible. After the design was verified, the next step was to implement the design onto FPGA, in this section the utilization and the timing report of each major block was mentioned as well as the report for the complete lane detection system, this section also contains the results when using different pictures of roads as inputs. From the obtained results it is seen that there is very small or no difference between the software model results and the results from the hardware, even when using different pictures of different angles and different qualities, the lane detection system was always able to detect the lanes in the images. After that, the next step was to implement functional safety on the system to increase the life-time of the system by detecting and recovering from failures.

## References

- [1] J.-S. Sheu, C.-H. Chang, T.-L. Huang and H.-J. Lin “Lane departure warning system using front-view and two mirror-view cameras” , Scientia Iranica Transactions B: Mechanical Engineering [www.scientiairanica.com](http://www.scientiairanica.com) , 2015.
- [2] I.El-Hajjouji, S.Mars, Z.Asrih and A.El-Mourabit , “A novel FPGA implementation of Hough Transform for straight lane detection” Article on *Engineering Science and Technology, an International Journal*,2019
- [3] V.Devane, G.Sahane, H.Khairmode and G.Datkhile, “Lane Detection Techniques using Image Processing,” Article on *ITM Web of Conferences* 40, 03011,2021.[Online Serial].
- [4] GurjyotKaur and Gagandeop Singh, “A Review of Lane Detection Techniques”, International Research Journal of Engineering and Technology (IRJET), Vol-02, Issue-03, 2015.
- [5] G.T. Shrivakshan and Dr.C. Chandrasekar, “A Comparison of various Edge Detection Techniques used in Image Processing”, International Journal of Computer Science Issues, Vol. 9, Issue 5, No 1, September 2012.
- [6] Aggarwal, Himanshu. “Study and Comparison of Various Image Edge Detection Techniques.”, International Journal of Image Processing, 2009.
- [7] Nazma Nausheen, Ayan Seal, Pritee Khanna, Santanu Haldar, “A FPGA based Implementation of Sobel Edge Detection”, Microprocessors and Microsystems (2017), doi: 10.1016/j.micpro.2017.10.011
- [8] Anuvab Sahoo and Dr. Mamata Panigrahy KIIT, Bhubaneswar, “Hardware Implementation of CORDIC Algorithm”, International Conference on Applied Electromagnetics, Signal Processing and Communication (AESPC) 2018.