University of Cincinnati

Date: 3/7/2014

I, Naren Ramesh , hereby submit this original work as part of the requirements for the degree of Master of Science in Electrical Engineering.

It is entitled:

A Hardware Implementation of Hough Transform Based on Parabolic Duality

Student's name: **Naren Ramesh**

This work and its defense approved by:

Committee chair: Carla Purdy, Ph.D.

Committee member: Wen Ben Jone, Ph.D.

Committee member: George Purdy, Ph.D.

9282

A Hardware Implementation of Hough Transform Based on Parabolic Duality

A Thesis submitted to the Graduate School

Of

The University of Cincinnati

In partial fulfillment of the requirements for the degree of

Master of Science

in the

Department of Electrical and Computer Engineering

of the

College of Engineering and Applied Sciences

March 2014

by

Naren Ramesh

B.E. Electrical and Electronics Engineering

Anna University, 2011

Committee Chair: Dr. Carla Purdy

Abstract

Hough Transform is a pattern recognition tool commonly used in many image processing algorithms. The detection of straight lines is the core to most of these applications. Over the years, many Hough Transform implementations have been proposed to reduce its computational overhead and also for detecting other patterns, i.e., curves and circles. The one problem that still remains fundamental to the original form of the Hough Transform with respect to straight line detection is its inability to detect vertical lines. This has forced the algorithm to use the parametric form rather than the parabolic form for straight lines in most cases. This thesis works on developing a method to overcome this drawback by incorporating two different forms of the slope intercept equation of a line, one using the regular form to detect lines with slopes less than 1 and the other by rotating the axis by 90 degrees to detect lines with a slope greater than 1 or infinity. We also consider a reflection of the input data points, which would mean the slope in the accumulator grid needs to be set only from 0 to 1. This makes sure the calculated values use relatively lesser memory for high precision. We maintain a [0,1] x [-1,1] accumulator grid to keep all our calculations within bounds.

Over the last decade, Field Programmable Gate Arrays (FPGAs) have become a widely popular hardware implementation tool due to their low cost and reconfigurable nature. FPGAs are proving to be the front runner as the most favored hardware implementation platform. Thus, in this thesis we try to design a hardware model of our new algorithm on an FPGA device to determine its basic functionality with respect to computational overhead and accuracy. The new algorithm overcomes the difficulties of hardware implementation of the parametric form of Hough Transform by neglecting the use of CORDIC algorithms and trigonometric lookup tables, which only add extra overhead and loss of precision. In this project, we design our algorithm by building a synthesizable model using Verilog and the Altera Quartus II tool, which is useful in giving us an idea on the memory and device usage of the proposed algorithm. We compare our design with some of the other similar hardware implementations of the Hough Transform.

Acknowledgements

Working on this thesis has been the most challenging and yet enjoyable phase of my academic life and I would like to take this moment to acknowledge everyone who has been a part of this journey.

Firstly, I would like to thank Dr. Carla Purdy for her time and patience during the course of this work. She has always been helpful by providing guidance and support for any query that I came to her with.

I would like to make a special mention thanking Dr. George Purdy, I could have never completed this thesis without his assistance. He brought his expertise in the field of Computational Geometry to the forefront by continuously providing tips and suggestions on ways with which I can improve the algorithm. He showed great patience in explaining simple concepts that I was unfamiliar of to start with. I would also like to thank him for participating as a defense committee member.

I thank Dr. Wen Ben Jone for taking time out of his busy schedule to participate as a defense committee member.

I would also like to thank Dr. Justin W. Smith for the time he spent in helping me through the initial phases of this work. He went out of his way to help me understand some of the fundamentals in relation to this topic.

I would like to thank the University of Cincinnati and the Department of Electrical and Computer Engineering in specific for providing me with a great platform to learn and develop myself as an Engineer over the last few years.

Lastly, I would like to thank all of my family, friends and roommates who have played a vital part in encouraging and supporting me through this arduous journey.

Naren Ramesh

TABLE OF CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1 The history of the Hough Transform

The Hough Transform has become one of the most widely used methods for detecting geometrical shapes in images. Based on its wide variety of applications, it is easily among the most widely used concepts in the field of computational mathematics and computer vision. Hough's procedure for determining straight lines and complex patterns was originally awarded a US patent in the year 1962 [1]. The original patent contained the basic idea of how one can use the correlation between point-line transformations to detect patterns in the geometrical space. It was later extended by A. Rosenfeld [2], but it was not until 1972 that the Hough Transform was introduced in a paper to the research community by Richard O. Duda and Peter E. Hart [3]. This paper described in detail how one can map points onto a parametric space so as to negate the difficulties of dealing with lines with a steep slope, i.e., vertical lines. Duda and Hart also extended their findings by incorporating the equation of a circle to detect curves using the same principle. Over the years, the Hough Transform has had many implementations, but the core of its applications still use the method developed by Duda and Hart.

1.2 The Standard Hough Transform (SHT)

The slope-intercept representation of a straight line can be expressed as

$$y = mx + b \tag{1}$$

As shown in Figure 1.1, x and y are Cartesian coordinates, m is the slope and b is the y-intercept of the straight line.

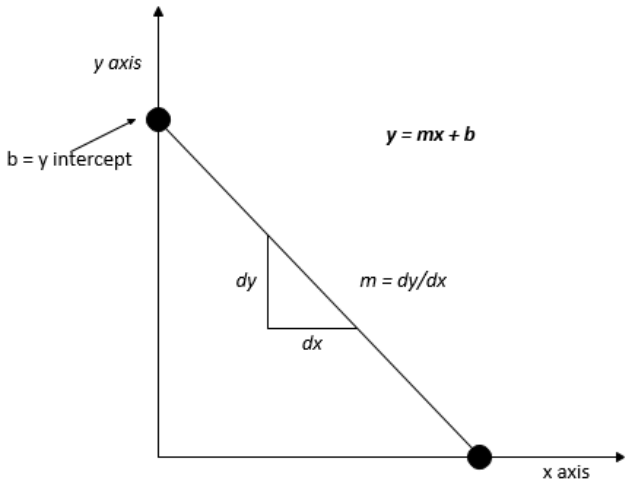

Figure 1.1: Slope-Intercept representation of a line

Duda and Hart [3] took into consideration the fact that Hough's original concept of using the slope intercept plane to detect straight lines did not work for lines having an infinite slope.

They used the fact that every point on a line in the Cartesian space can be represented using a parameterized equation.

The parameterized equation of a point on a line is given by

$$x(\cos\theta) + y(\sin\theta) = \rho \tag{2}$$

As shown in Figure 1.2, θ is the angle the normal drawn from the origin makes with the point and ρ is the algebraic distance from the origin. Assuming the line contains a set of n points $\{(x_1,y_1)...(x_n,y_n)\}$, each of these n points can be represented as a set of ρ and θ values given by $\{(\rho_1,\theta_1)...(\rho_n,\theta_n)\}$.

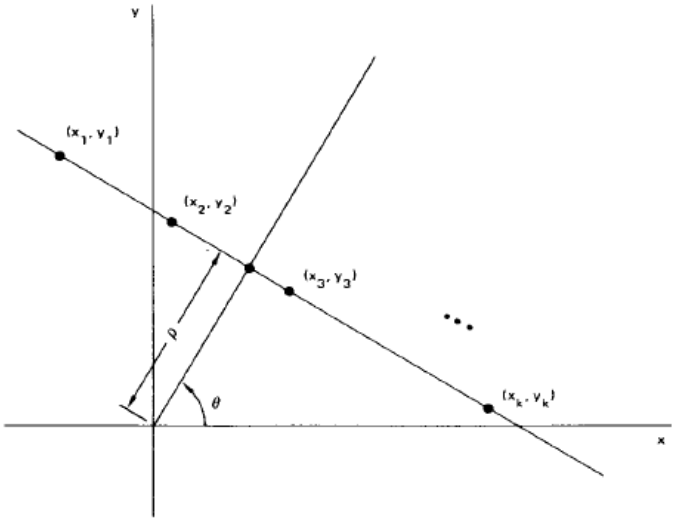

Figure 1.2: Projection of points on a line based on the parameterized equation [3]

Now, by varying θ from 0 to 180° in Eq. 2 and applying it on a set of approximately collinear points in the Cartesian space, each of these points would form a curve in the ρ - θ parameter space. Each point from the curve in the ρ-θ parameter space can be represented back as a line in the Cartesian space. Points that approximately lie in the same line will have intersecting curves at a common point in the parameter space. Thus, given a set of points, we can apply the above concept and determine a subset of these points that are approximately collinear. Thus, the traditional concept primarily uses point to curve transformations to convert collinear points into concurrent curves.

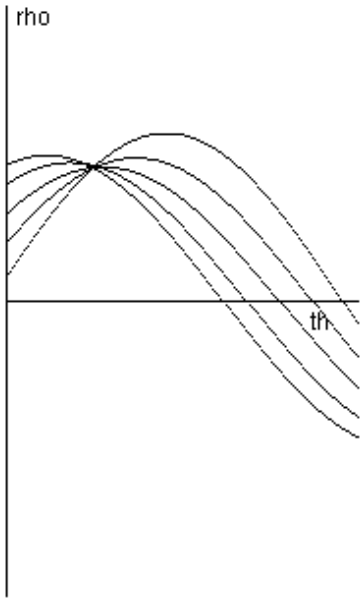

Figure 1.3: Example of intersecting curves in the Hough parameter space

Figure 1.3 shows an example of intersecting concurrent curves in the ρ - θ parameter space. The five curves would belong to five approximately collinear points as they all intersect at the same point. The intersecting point can then be used to redraw the line formed by the points.

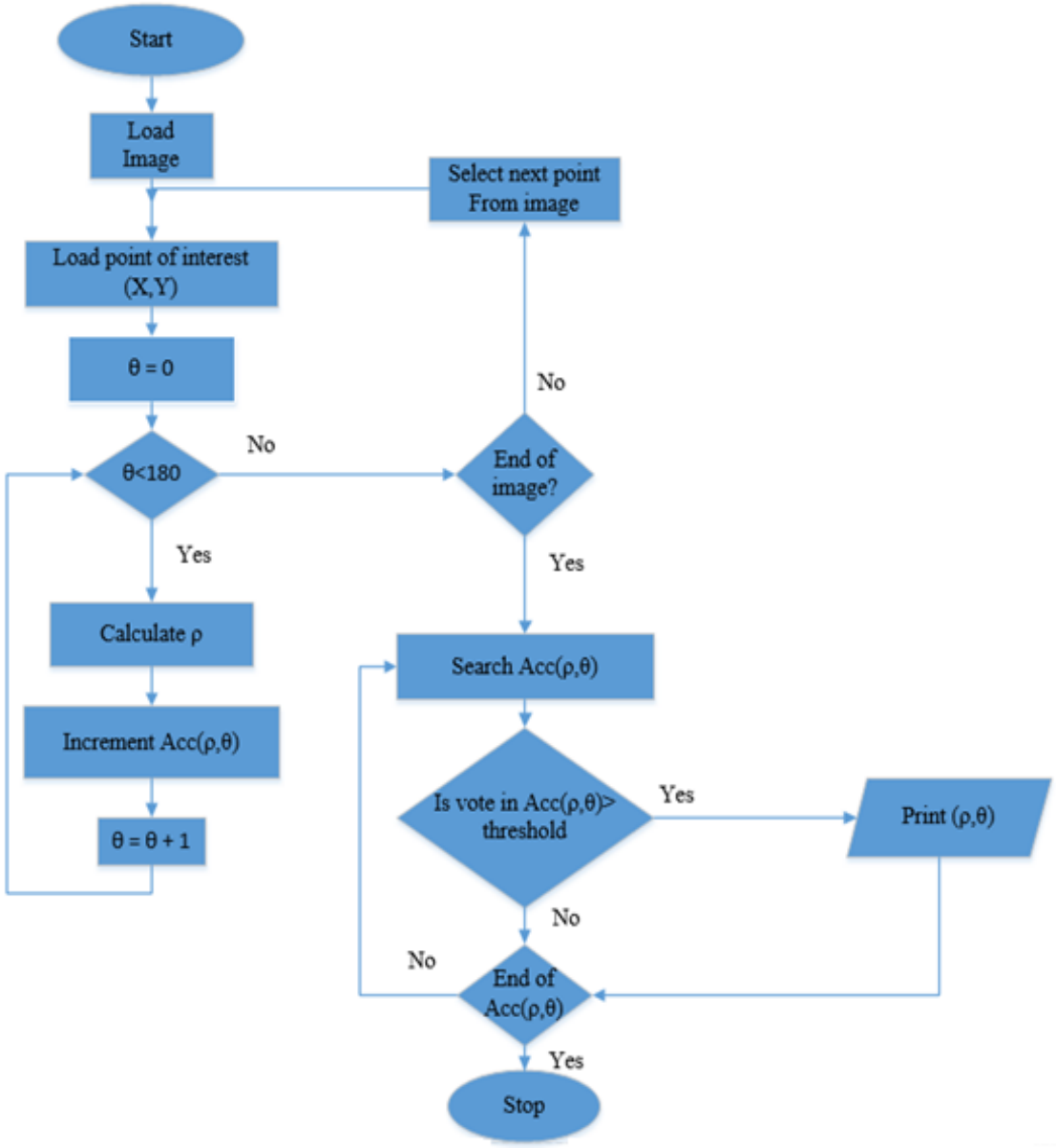

Figure 1.4: Flowchart of the Standard Hough Transform

Figure 1.4 shows the basic flow chart used to apply the Standard Hough Transform to an image. The algorithm is pretty simple to comprehend. Image processing plays an integral part in

the early stages of the algorithm. Every image must undergo filtering and edge detection. Using this new image, we must map it onto a domain space to determine a correlation between each pixel in the image and an (x,y) coordinate point. Once we have all the points from the image, we can start to apply Hough Transform. We select the first point of interest in the image, this is now applied in Eq. 2.

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

We now vary θ from 0 to 180° and increment the accumulator, the accumulator is a two dimensional grid collecting votes for different (ρ, θ) combinations. We continue this process until every point of interest in the image has been evaluated. Next, we search the accumulator for voting counts greater than the selected threshold. Selecting the threshold can be tricky as some true lines in the image might have a relatively smaller vote than other true lines due to the number of data points and other rounding errors. Keeping the threshold too low might cause the algorithm to give us spurious lines. Thus, there could be a scenario where one must vary the threshold to make sure all the lines are correctly detected. The algorithm finally comes to an end once we have searched the entire accumulator space. We can recreate lines from the original image using the (ρ, θ) combinations with high votes in the accumulator grid. This is done by substituting the values in the parameterized equation and varying either x or y.

1.2.1 Shortcomings of the Standard Hough Transform

Even though the traditional Hough Transform is still a commonly used pattern recognition algorithm, there are a lot of shortcomings to it. The biggest drawbacks have been with respect to the hardware implementation of the algorithm because of the large memory it requires and the time it takes [20]. The parametric form of the Hough Transform has a computational complexity of $O(N^{2*}M)$. Where NxN is the size of the image and M is the resolution of θ. This can be very tedious for large images with large sets of data.

Also, to detect lines with high accuracy one needs to vary the value of θ with high resolution. This angular resolution is crucial in determining the right lines [3]. One must iterate the algorithm from 0 to 180° for all images to detect the right lines. The size of the image has no impact on this, making it computationally very heavy for small images or images with very few lines.

Lastly, and most importantly, the parameterized version incorporates trigonometric functions such as sine and cosine which adds to overheads for the hardware. Sine and cosine are primarily modeled on hardware devices such as FPGAs and ASICs using the CORDIC (Coordinate Rotational Digital Computer) Algorithm [4], which uses up a lot of system resources in terms of time and memory. One can also use trigonometric lookup tables, but in most applications where accuracy is the most important factor and it is not possible to get any sort of high precision using lookup tables. The above reason motivated us to look for a fresh approach, where the baggage of unnecessary hardware components on an FPGA can be removed and also higher accuracy can be achieved without the complexities of trigonometric functions.

1.3 Need for pattern recognition on FPGAs

Pattern recognition is a vital feature for many real world application oriented embedded systems. Pattern recognition not only deals with straight lines, it also involves detection of other shapes and sizes. But detection of straight lines is still very important as it can be used to build other shapes such as squares and triangles. Pattern recognition based on the Hough Transform was originally used to detect straight line paths within bubble chambers in the field of particle physics [5]. Pattern recognition has found a new lease of life in applications related to medical devices, where it can be used to detect harmful growths or shapes within the human body [6], [7]. Pattern recognition is also being used in high technology land mapping and satellite imagery devices [8]. In fact, good pattern recognition would be critical for applications that involve real time imagery. Hough Transform is also used in sharpening and correcting noisy images and videos [9]. In the field of robotics, the quality of pattern recognition is possibly what differentiates a successful model from an unsuccessful one. From aerial drones to robots working in an assembly line, recognizing a pattern is crucial to proper functioning. Pattern recognition is also fundamental to deciphering texts or any other printed data. Thus, it is very important to incorporate pattern recognition on a suitable hardware platform.

Keeping the cost of a hardware module low is important to its being used widely. FPGAs (Field Programmable Gate Arrays) have become one of the most common tools for hardware implementation. Xilinx was one the first companies to develop an FPGA in the late 1980s. The

first ever FPGA was designed to seem like a cross between a PLD (Programmable Logic Device) and an ASIC (Application Specific Integrated Circuit) [28].

FPGAs are designed with an array of logic blocks which are called configurable logic blocks (CLBs). These, along with the necessary I/O pads and routing channels, form the core of the device. The CLBs can be programmed to be anything from a single logic gate to a complex RAM (Random Access Memory). The CLB is generally built with a configurable switch matrix along with flip-flops and some circuitry for selection, i.e., MUX. Interconnects as the name suggests are used to connect the signals between various CLBs on the device. Most FPGAs also have an in built digital clock with high precision and memory blocks to store data.

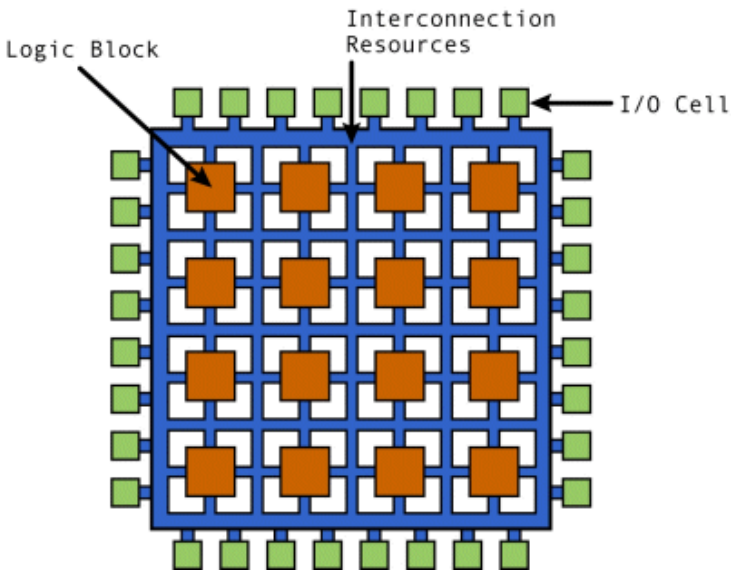

Figure 1.5: Internal architecture of an FPGA [10]

Over the last decade, a lot of other companies have realized the performance capabilities of FPGAs and have joined the bandwagon in manufacturing them. Nowadays, FPGAs are being sold with extra added intellectual property such as DSP blocks, memory blocks, USB ports and even ethernet capabilities. A high end FPGA can have as many as two to four million logic cells. One can also build softcore processors on the FPGA.

Most of the applications presented in the previous section are bound by not having a high investment cost, and a low cost FPGA would be an ideal platform for them. Also, one can manufacture FPGAs with other modules built-in for specific applications. For example, an FPGA for image processing would be better off with more built-in DSP modules. If we need a lot of memory we could get an FPGA with a bigger built-in RAM. Programmers and designers can quickly model the FPGA to function as they want using hardware description languages like VHDL or Verilog. Since the design is primarily built around the program we write, it makes it very easy to debug and modify even in the last stages of the design cycle. Thus, their reconfigurable nature along with its low cost make FPGAs highly sought after for large scale applications and quick programmable results when compared to ASICs.

1.4 Thesis organization

The thesis is organized as follows

Chapter 1 dealt with the basic history and helped us get an idea of the Standard Hough Transform. It also explained its drawbacks and gave us a brief explanation as to why FPGAs would be an ideal module on which one can build applications which use pattern recognition.

Chapter 2 is a background study of the different types of straight line detectors and also describes how different versions of the Hough Transform have been implemented in hardware.

Chapter 3 describes our approach in designing the line detection algorithm. It sheds light on the geometric aspects as well as how the hardware on the FPGA is modeled.

Chapter 4 discusses the simulation results of the algorithm and also compares it with some of the earlier straight line detectors.

Chapter 5 summarizes the work of this thesis and also gives us suggestions on how we can improve and extend our design.

CHAPTER 2

LITERATURE REVIEW

The literature review is divided into two sections; the first section deals with the different types of straight line detectors and the second one looks into the different hardware implementations of the Hough Transform.

2.1 Straight line detectors

As already mentioned, Duda and Hart [3] published the first research paper on what is now called the Standard Hough Transform. In the paper they test the method on the picture of a box to show the efficiency of the line detector. They acknowledge the fact that while the quantization of ρ and θ is critical for better resolution, it also adds to computation time.

Tuytelaars et al. [11] introduced a method they called the Cascaded Hough Transform (CHT), where they apply a series of Hough Transforms over each other The first one is to detect lines in the image, while a second one determines collinear peaks in the parameter space. A third transform then determines collinear vertices. This is shown in Figure 2.1. This was one of the first papers to introduce a bounded [-1,1] parameter subspace. However, their approach did require a lot of memory as each Hough Transform had to be done independently of the others. This method is still commonly applied to determine vanishing points in images.

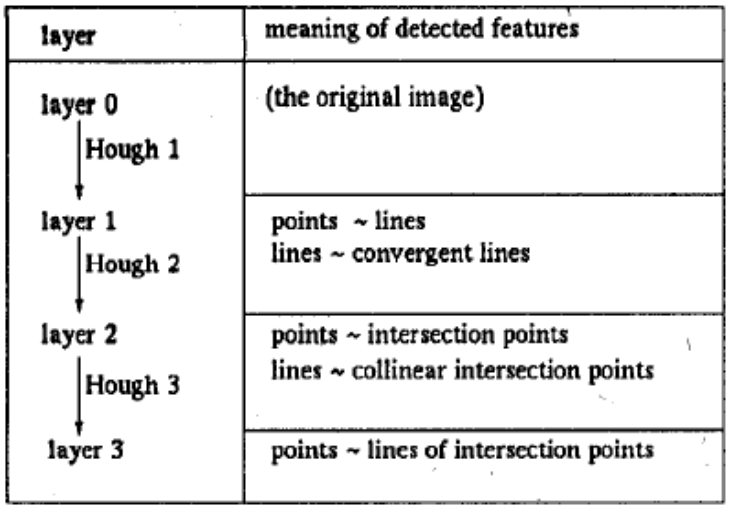

Figure 2.1: Flowchart showing the Cascaded Hough Transform [11]

Li et al. [12] came up with a new method called the Fast Hough Transform (FHT). They took into account the exponential growth in memory requirement for high resolution images. They dealt with this by representing the parameter space as a k-tree structure and recursively dividing it into hyper cubes, where areas with no votes were not considered in the parameter space. This was one of the more radical representations of the Hough Transform. The only possible drawback in this method was that the fast Hough Transform is limited to searching only one solution. Multiple instances were not considered in the paper.

More recently, Mejdani et al. [13] considered a set of different straight line detectors. They introduced three new methods called the Revisited Hough Transform (RHT), the Parallel-Axis Transform (PAT) and the Circle Transform (CT) for curves. The RHT uses a method

similar to ours based on the rotation of the x-y coordinate axis, but since the axis intercept b is not within any bounds we believe there will be lesser precision. The PAT was based on the method introduced by A. Inselberg [14], where the points on a line are represented in parallel coordinates to form an infinite family of lines and the distance between the two axes is used to represent it as a single point. In [13], PAT was deemed the superior amongst all the other line detectors.

Fernandes and Oliveira [15] introduce an interesting software approach to improve the voting scheme of the Hough Transform by clustering approximately collinear pixels in the image together as shown in Figure 2.2. An elliptical kernel is then computed from the line the cluster represents and then finally a vote for kernels with the bigger contributions is taken. Though this method shows great accuracy for images with high resolution of pixels, it is still only a real time software implementation.

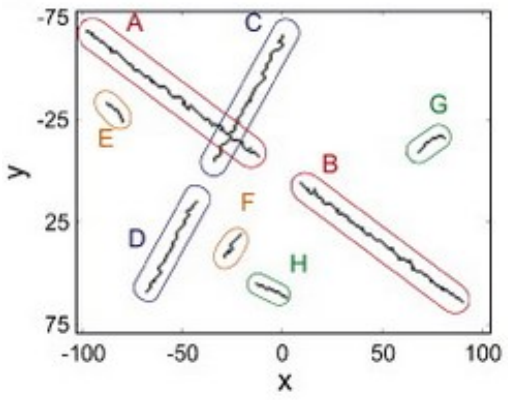

Figure 2.2: An example showing the clustering of pixels into segments [15]

Bhattacharya et al. [16] prove Duda and Hart's original hypothesis that "there is no Point to Line Mapping (PTLM) that maps all sets of collinear points into peaks that lie in a bounded region." They state that one can build a practical Hough Transform using two PTLMs, which is in a way the foundation of our thesis.

Due to the difficulties with respect to Hough Transform, other methods have also started to take the center stage with respect to line detection. Radon Transform has been successfully used to detect linear segments in images [17]. Radon Transform works by giving us a correspondence between the data on an image and its projection along some angles. We can explain this based on the parametric equation of a line,

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

A Radon Transform is applied to give,

$$R(\rho,\theta) = \iint_{-\infty}^{\infty} F(x,y)\,\delta(\rho - x\cos\theta - y\sin\theta)\,dx\,dy \text{ [17]}$$

Here δ is the Dirac delta function. The representation of Hough Transform is very much the same as Radon Transform in terms of their respective parameter spaces and Radon Transform can be considered a discretized form of the Hough Transform. Radon Transform is not as computationally heavy as Hough Transform and is widely used for applications with relation to Tomography [17].

Since Hough Transform has been around since the early 1960's, there have been numerous versions of it. But the main aim in almost every version was to find a way to improve

accuracy and reduce complexity in terms of the brute force method the Hough Transform utilizes.

2.2 Hardware implementations of Hough Transform

Though Hough Transform is computationally a very heavy algorithm, there has been a lot work done on implementing it on hardware. One of the first implementations of Hough Transform on an FPGA was by Tagzout et al. [18]. They based their design on a Fast Incremental Hough Transform 2 (FIHT2) model which requires no trigonometric operations except for initially setting up the values. But, they encountered errors due to the approximations of $\cos\theta$ and $\sin\theta$ which tend to add up as time goes on. This forced them to always keep track of the error to make sure it doesn't cross a tolerated amount. When it did cross the margin, they used the conventional Hough Transform equation to bring it back to acceptable limits. Thus, they eventually had to use trigonometric lookup tables for the correction process. They successfully tested a 256x256 size image, using only 324 CLB's and a clock speed of 27.1MHz on a Xilinx FPGA board.

Many other hardware implementations have been built based on the CORDIC algorithm, which provides an iterative method of performing vector rotations by arbitrary angles using only shifters and adders [4]. The iterations can be used to decipher the value of a specific trigonometric function. CORDIC algorithm uses two methods namely, rotation mode and vector mode. In the rotation mode, we can set the angle to a desired value based on subtracting or adding such that the current angle is as close as possible to the required one. In vector mode, the

CORDIC algorithm would rotate the input vector to the desired angle by aligning it with the x axis.

In the basic form as described in [19], each iteration can be defined as,

$$x_{i+1} = x_i - m.\ \mu_i y_i . 2^{-Sm,I}$$

$$y_{i+1} = y_i + \mu_i x_i . 2^{-Sm,I}$$

$$z_{i+1} = z_i - \mu_i . a^{-Sm,I}$$

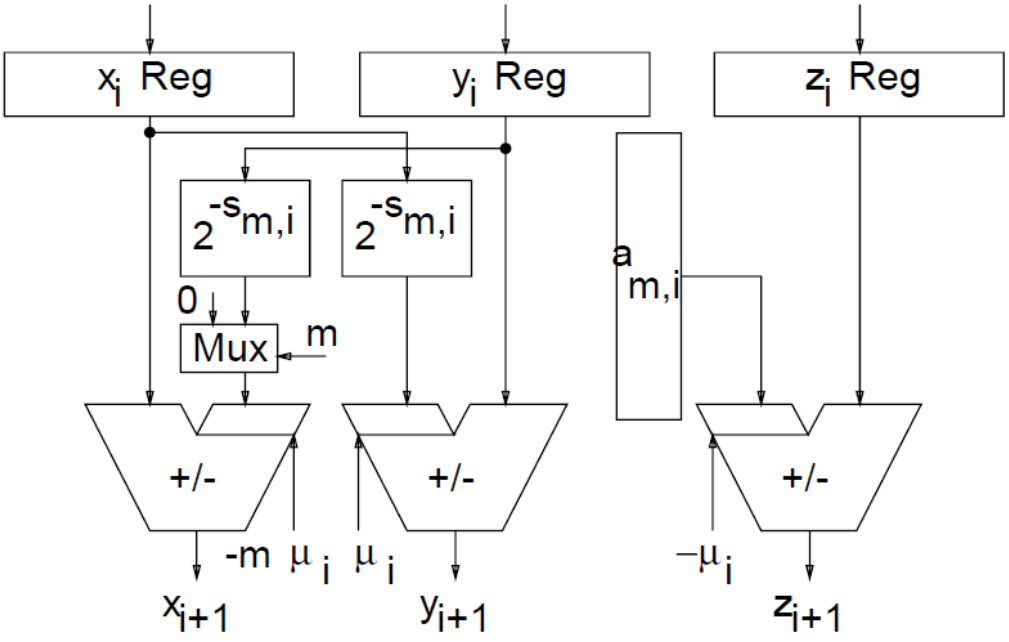

Figure 2.3: Basic structure of a CORDIC processing unit [19]

Figure 2.3 shows the basic CORDIC architecture for one iteration in a parallel iterative mode, here x_i and y_i are the initial vectors, z_i is used to control the rotation angle $a_{m,I}$, the rotation

direction is controlled by μ_i. The $2^{-Sm,I}$ is used to define the shift sequence in a radix 2 number system. This can be changed if required. As one can clearly see, for high precision or a large number of bits the size of the design would grow exponentially and become too big or account for too much time to be feasible on an FPGA.

Karabernou and Terranti [20] have proposed a real time FPGA model for Hough Transform. In their version they use the gradient method to reduce the amount of calculations along with the CORDIC algorithm to make it less intensive computationally.

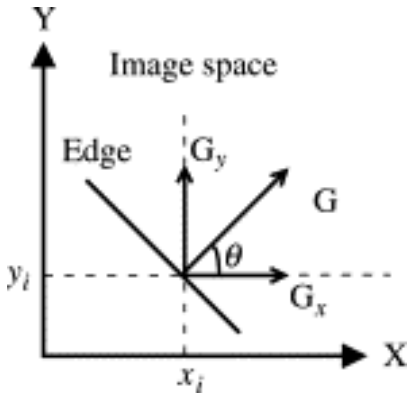

Figure 2.4: Relationship between an edge and gradient [20]

As shown in Figure 2.4, gradient based methods are used to find the orientation of the line with respect to a point. This would constrict the computation to only a single value of θ, making it easier to design the trigonometric functionality. G_y and G_x are the vertical and horizontal gradients respectively. These must be obtained in the preprocessing stages. Their straight line detection architecture is composed of two CORDIC units which are used simultaneously to compute ρ and θ. They implement the parameter estimation architectures to

estimate time and complexity. They also propose a prototype design of the global architecture with edge detection, parameter space calculation and straight line image rebuilding.

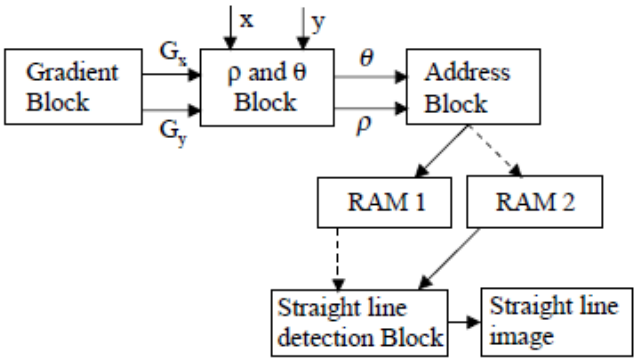

Figure 2.5: FPGA based Hough Transform architecture by Karabernou and Terranti [20]

The prototype of their proposed global design uses two RAMs. One RAM is used to maintain the current voting counts of the present frame while the other saves the parameter space of the previous frame. At the end of every frame the RAM roles are swapped. Thus, the line detection and rebuilding can be done in a parallel manner. The straight line rebuilding architecture is far more complex, with four CORDIC units for the estimation of sinθ, cosθ, multiplication and division.

Kalomiros and Lygouras [21] incorporate a mixed hardware/software co-design for image processing using an Altera FPGA board along with the National Instrument (NI) LabVIEW^{TM} software. They designed their architecture based on DSP techniques using FIR filters and Gaussian filters. They use the Altera NIOS-II software to configure the design as a System on a Programmable Chip (SOPC).

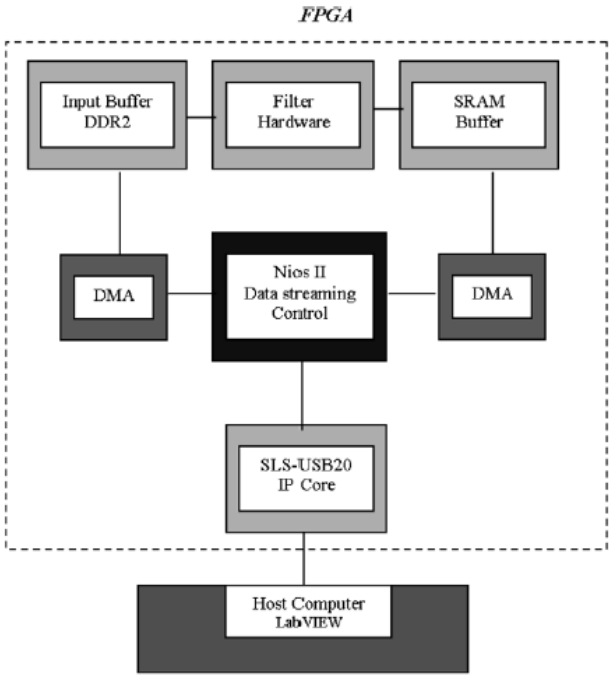

Figure 2.6: Basic hardware architecture by Kalomiros and Lygouras [21]

As shown in Figure 2.6, the NIOS-II data controller is the integral part of the design, acting as a peripheral between the host computer and the FPGA device. Their work is restricted to only edge detection and not any specific shape or segment in the image, but it gives us a good example of the performance capabilities of a line detector within a larger system which might include an image processing unit. This shows that we can include a line detection algorithm like the one we are going to propose here along with an image filtering system on the same FPGA board.

More recently, Zhou et al. [22] designed another implementation of Hough Transform on a Xilinx FPGA device using embedded DSP blocks and block RAMs. This is shown in

Figure 2.7. They incorporated a dual port RAM for reading and writing operations on the memory. The DSP blocks are used to preload values of cosθ and sinθ and compute values for xcosθ and ysinθ in parallel. They show that it is possible to split $0 \leq \theta \leq 180$ into two parts [0,89] and [90,179]. Thus they require DSP blocks only for $0 \leq \theta \leq 90$ instead of $0 \leq \theta \leq 180$.

The equations in [22] were described as,

$$\rho = x(\cos\theta) + y(\sin\theta) \text{ where } \theta \text{ is varied from 0 to 89}$$

$$\rho = -x(\cos\theta) + y(\sin\theta) \text{ where } \theta \text{ is varied from 1 to 90}$$

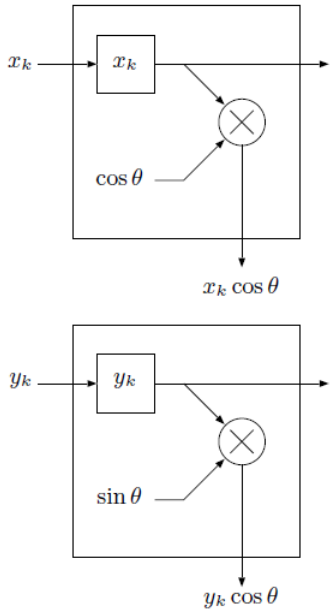

Figure 2.7: Example DSP blocks [22]

Their approach reduced the computational time of the algorithm from 180m voting cycles to 97+m clock cycles, where m is the number of edge points. The design seems ideal for real time applications but uses 14493 slices of logic blocks along with 178 DSP slices, which is a lot of resources.

CHAPTER 3

METHODOLOGY

3.1 Parabolic duality

The principle of duality allows one to convert points to lines and lines to points within a given space. “The duality transform ($\mathbb{D}$) is a function that provides a mapping between points and lines (in $\mathbb{R}^2$) such that their incident relationships are preserved” [23]. This means the incident property of a point lying on a line or a line containing a point is translated to their respective dual planes.

As illustrated in Figure 3.1, the primal plane or non-dual plane is the plane in which the original point P(p_x,p_y) is present, the dual of P is denoted as $\mathbb{D}$ (P) and is a line in the dual plane defined as

$$\mathbb{D}\ (P)\ :=\ y=p_x*x-p_y$$

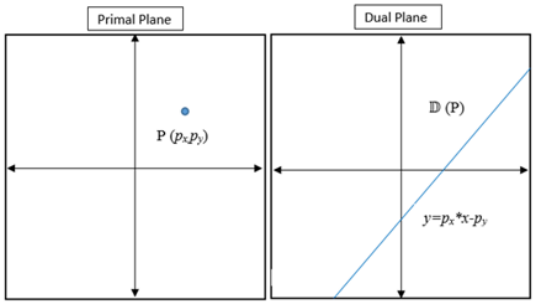

Figure 3.1: Relation between P and $\mathbb{D}$ (P)

As illustrated in Figure 3.2, if the primal plane contains a line L, expressed as L(y=mx+b). The dual of the line L is a point $\mathbb{D}$ (L) representing the slope and negative y-intercept of the line in the dual plane. We could also say the dual of line L, $\mathbb{D}$ (L) can be defined as

$$\mathbb{D}\ (L)\ := (m,-b)$$

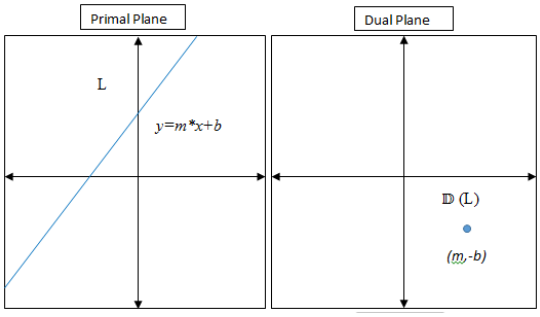

Figure 3.2: Relation between L and $\mathbb{D}$ (L)

We can summarize this fact as

Property 3.1.1: $\mathbb{D} = \mathbb{D}^{-1}$

This signifies that the duality transform function is an inverse of itself, meaning,

$$\mathbb{D}(\mathbb{D}(P)) = P \text{ or } \mathbb{D}(\mathbb{D}(L)) = L$$

Property 3.1.2: Point P ∈ L if and only if $\mathbb{D}$ (L)∈ $\mathbb{D}$ (P)

Using these properties it can be proven that the points {p1,p2…pn} are collinear, if and only if the lines $\mathbb{D}(\{p1, p2..pn\})$ intersect at a common point and vice versa [23].

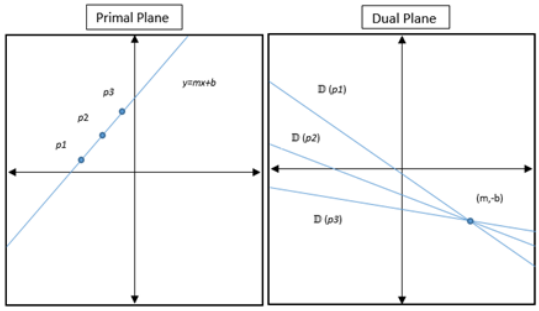

Figure 3.3: Relationship between collinear points and intersecting lines

Given a system of approximately collinear points, by applying the duality transform function we can determine the cells in the dual plane that have a large number of lines passing through them. Such a cell in the dual plane is characterized by the slope and y-intercept of the line formed by the input points. It is important to note that the above properties hold true only for non-vertical lines. Thus, the point-line duality gives us a new dimension to work with where the problem of finding collinear points is converted into a problem of finding intersecting lines.

3.2 The algorithm design approach

We must consider how to deal with lines that have a steep or infinite slope. This can be simply done by tilting the axis of the plane by 90°. The original difficulty with respect to Hough

Transform not being able to detect vertical lines was because of the fact that vertical lines cannot be characterized with respect to the slope and y-intercept on applying the duality transform function. This is because the slope of a vertical line is infinite. By tilting the axis, we are indirectly swapping the x axis with y axis, causing the vertical lines to become horizontal lines and vice versa. The tilting of the primal plane detects lines with $|m| \geq 1$. The equation of such a line in the tilted subspace would become

$$x = my + b \tag{3}$$

Thus, it is possible to detect all the lines for a given set of points using Eq. 1 and Eq. 3 which use the original and rotated forms of the planes. In our design, given a set of approximately collinear points in the form (x,y) we apply the two types of duality transform functions. We extract the cells with large numbers of votes or lines passing through them and substitute the values for *b* and *m* in Eq. 4 and Eq. 5 below to reconstruct the lines. Again it is necessary that we maintain our inputs $(x,y) \in [-1,1]$. This makes sure that our y-intercept is always within bounds.

Case 1: For detecting lines with slopes $|m| \leq 1$ originally,

$$b = y - mx \text{ where } (x,y,m,b) \in [-1,1] \tag{4}$$

Case 2: For detecting lines with slopes $|m| \geq 1$ originally,

$$b = x - my \text{ where } (x,y,m,b) \in [-1,1] \tag{5}$$

We also modify our algorithm so that it is sufficient to maintain m ϵ [0,1]. This is done by reflecting the input points. Given a line L($y=mx+b$) consisting of n points $\{(x_1,y_1)...(x_n,y_n)\}$, a reflection of these n points would be given as $\{(-x_1,y_1)...(-x_n,y_n)\}$, which in turn transforms the equation of the line to L($y=-mx+b$). This reflection of the input points would mean that it is sufficient to maintain a parameter space for m ϵ [0,1] and b ϵ [-1,1]. In terms of hardware this could be very advantageous as the size of the accumulator grid we would need to design is reduced by half and could potentially save a lot of memory in the hardware implementation of the algorithm.

To sum up, our algorithm is now primarily defined by 4 different states, each state representing the equation of the line we are trying to detect. State 1 and 2 work with the regular inputs whereas state 3 and 4 work with the reflected inputs.

State 1: For detecting lines with slopes, $0 \leq m \leq 1$ originally,

$$b = y - mx \text{ where } m \epsilon [0,1], (x,y,b) \epsilon [-1,1] \quad (6)$$

State 2: For detecting lines with slopes, $1 \leq m \leq \infty$ originally,

$$b = x - my \text{ where } m \epsilon [0,1], (x,y,b) \epsilon [-1,1] \quad (7)$$

State 3: For detecting lines with slopes, $-1 \leq m \leq 0$ originally,

$$b = y + mx \text{ where } m \epsilon [0,1], (x,y,b) \epsilon [-1,1] \quad (8)$$

State 4: For detecting lines with slopes, $-\infty \leq m \leq -1$ originally,

$$b = -x - my \text{ where } m \in [0,1], (x,y,b) \in [-1,1] \qquad (9)$$

Thus, it is possible to detect all the different orientations of a line for a given set of points. By applying the four variations of the duality transform function and by mapping these lines on an accumulator we can determine *b* and *m* values of the lines formed by the input points. The accumulator is a grid consisting of a set of subdivided small cells. Each cell takes into account every line that crosses through it in the dual plane. We maintain a constant subspace for the parameters in all the four states. This makes it relatively easy to implement the algorithm in terms of the hardware design. The idea of building an accumulator MxB [0,1] x [-1,1] and keeping (x,y) ϵ [-1,1] is crucial to the design as one can maintain a high range of accuracy without any of the calculations going out of bounds.

Costa and Sandler [24] earlier introduced a similar idea called the Binary Hough Transform, but in their approach they limit only the value of slope to $m \in [0,1]$ and other parameters are still greater than 0. This forced them to add extra rounding equations to keep the y-intercept within limits. With the advances in image processing one can now easily map all the pixels in an image to a domain space in the range [-1,1] x [-1,1] which would give each pixel a coordinate value(x,y) ϵ [-1,1]. By doing this, we are not only reducing the complexity of the algorithm, but also improving the overall accuracy of the algorithm. Also, the Binary Hough Transform [24] has never been implemented on an FPGA.

3.2.1 Pseudo-code of the algorithm

The algorithm can be described as follows,

```
Begin
    Input data points p=(px,py) from image such that (px,py) ϵ [-1,1]
    Select input state for the system of lines you wish to detect
    Initialize accumulator[B][M] to 0; where B=M*2

        Case: State 1
        for each data point p=(px,py) in the image do
        for m= 0 to 1, where m=m+m_resolution do
        compute b= (-px*m)+ py.
        determine cell in accumulator[i][j] of point (m,b)
        increment count(accumulator[i][j]) by 1
        end of loop on m
        end of loop on p
        call output(accumulator[i][j],state)

        Case: State 2
        for each data point p=(px,py) in the image do
        for m= 0 to 1, where m=m+m_resolution do
        compute b= (-py*m)+ px
        determine cell in accumulator[i][j] of point (m,b)
        increment count(accumulator[i][j]) by 1
        end of loop on m
        end of loop on p
        call output(accumulator[i][j],state)

        Case: State 3
        for each data point p=(-px,py) in the image do
        for m= 0 to 1, where m=m+m_resolution do
        compute b= (-px*m)+ py
        determine cell in accumulator[i][j] of point (m,b)
        increment count(accumulator[i][j]) by 1
        end of loop on m
        end of loop on p
        call output(accumulator[i][j],state)

        Case: State 4
        for each data point p=(-px,py) in the image do
        for m= 0 to 1, where m=m+m_resolution do
        compute b= (-py*m)+ px
        determine cell in accumulator[i][j] of point (m,b)
        increment count(accumulator[i][j]) by 1
        end of loop on m
        end of loop on p
        call output(accumulator[][],state)

        end case

end
```

```
Function output(accumulator[][],state)

        for i=0 to B do
        for j=0 to M do
        if count(accumulator[i][j]) > threshold)

            Case: State 1

            determine point (m,b) that is represented by cell [i][j]
            print("The line is in domain 1 and can be represented
            using the equation")
            print( b = - (m*x) + y ), where m & b are known.

            Case: State 2

            determine point (m,b) that is represented by cell [i][j]
            print("The line is in domain 2 and can be represented
            using the equation")
            print( b = - (m*y) + x ), where m & b are known

            Case: State 3

            determine point (m,b) that is represented by cell [i][j]
            print("The line is in domain 3 and can be represented
            using the equation")
            print( b =  (m*x) + y ), where m & b are known

            Case: State 4

            determine point (m,b) that is represented by cell [i][j]
            print("The line is in domain 4 and can be represented
            using the equation")
            print( b = - (m*y) - x ), where m & b are known

            end case

        end if
        end of loop on j
        end of loop on i

end of function
```

The pseudo-code presents us with a generic implementation of the algorithm. One of the primary advantages of our algorithm is that it is not required to go through all the states. If need be, we can detect the lines of the specific orientation we wish to by just selecting the specific state. But, for an unknown set of points or image it is necessary that we traverse through all the four states to make sure all the lines are detected. Selecting the parameters of [B][M] for the accumulator is integral in making sure the resultant lines are of high precision. We must select B and M such that M=B/2, so that we can maintain symmetry in the accumulator design. The larger the value of B and M, the higher is the accuracy of our design. Selecting the threshold is again a tricky part in the design. One might have to vary the threshold within certain parameters to identify the right lines. In terms of the algorithm the threshold can be set as a user defined parameter so that he/she can vary it as they wish or we could design a heuristic to set the threshold based on the application to which the algorithm would be applied to. The final output prints the (m,b) coordinates with the most number of lines passing through them in the dual plane. We can reconstruct the line in the primal plane with the *m* and *b* values, the equation of the line and then determine *x* by varying *y* or vice versa.

The algorithm is independent of the approach for the filtering of the image, edge detection and point extraction. The algorithm is well suited for hardware implementation as it is reconfigurable and easily modifiable for desired precision.

3.3 Hardware implementation

As stated in chapter 1, FPGAs have become a powerful tool for hardware design implementations. Their low cost, reconfigurable nature and ability to work in parallel with other systems within a larger design are the main reasons for their wide usage nowadays. FPGAs can be designed using Hardware Description Languages (HDLs) such as VHDL or Verilog. While VHDL is more structural, Verilog is more like the C language. Our implementation is restricted to the programming of a synthesizable model using Verilog which is then simulated on ModelSim™ to verify the correct functioning of the program. We do not port our design to an FPGA, but we compile it on an FPGA simulator to check its compatibility.

Designing an algorithm in hardware requires some care. Since the algorithm largely deals with decimal numbers between -1 and 1, we have to make sure the number representation system we use is coherent and can maintain an acceptable level of accuracy. Timing in the design is also an important factor, as reading the data from the image is much faster than the time it takes to apply the Hough Transform to it. We must make sure that the timing in inputting the new data points is done correctly. Timing is again an issue with respect to the memory, as we have to read and write to the same address of the memory block during the voting process. We must find a way to do this as quickly and as efficiently as possible. The algorithm has four different cases, thus in terms of the hardware design one would need to build a state machine.

In our design we are able to overcome all of the questions raised in the previous section. First, since we deal with only numbers in the range [-1,1]. A floating point architecture would be

an unnecessary overload in the design. So we restrict ourselves to a fixed point architecture. We use 32 bit numbers with 8 digits to represent the integer part and 24 for the fraction. We make sure that we assign a lot of bits to represent the fractional part to maintain precision. Arithmetically the algorithm requires adders, multipliers and dividers. These are easily implemented by the logic blocks of the FPGA and Quartus II makes sure that minimal resources and routing are used.

Timing issues are the biggest bottleneck in almost every hardware design, the rate of transfer of bits in the RTL level can vary in every part of the design. In our design, once an input data point is selected it takes some amount of time for the Hough Transform algorithm to be applied by varying m from 0 to 1. We must carefully calculate the time it takes for this process and input the next data point only after all the calculations and voting have been done for the current set of points. In a situation where we input the next point too early, we get faulty votes in the accumulator. In the design, some calculations are performed faster than others forcing some units of the architecture to be in an idle state waiting for other results. This is not favorable in most hardware designs and good design practice entails the use of pipelining. Pipelining is the process of breaking up one large monolithic task into various subtasks done by individual resources. The breaking up of the task is done so that resources in the system are used in parallel in completion of the subtasks. Pipelining essentially improves the throughput of the design. We insert a few registers along with non-blocking assignments in our design to implement pipelining. We add latency bubbles in the program when and where necessary to make sure there is enough time to make sure all calculations are accurately completed.

The biggest overhead in our design in terms of system resources is the memory requirements. We would need a large memory block for the accumulator. The larger the size of this block, the higher the accuracy of our algorithm. Since we are writing to the same address we are reading from, we design a dual port RAM. We use one set of ports specifically for reading and one set of ports specifically for writing to make the process more efficient. More details in regards to the working of the dual port RAM will be discussed in the sections following.

Since our algorithm revolves around four different cases, we build a synchronous state machine where the state can be selected by the user input for the orientation of lines he/she wishes to detect. Once all the computations in one state are completed the program automatically prints the results and resets the accumulator for the next state or set of inputs.

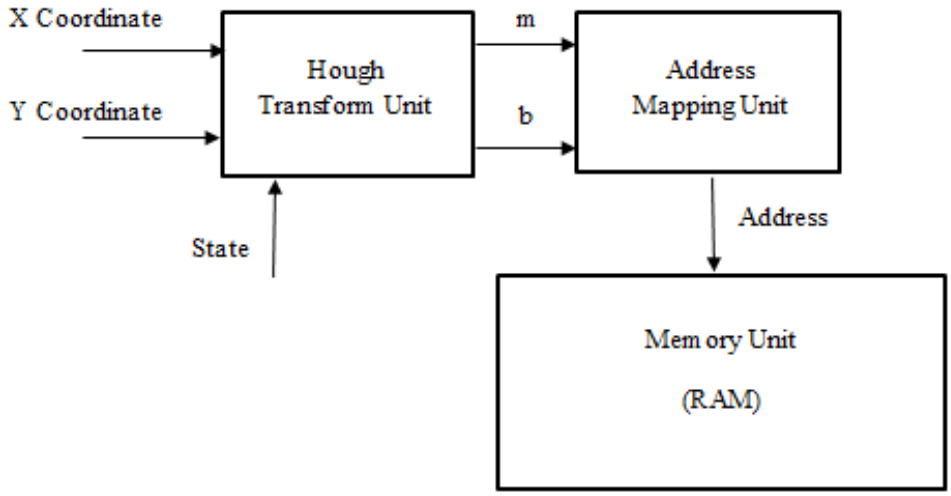

Figure 3.4: Block diagram of the hardware architecture

Figure 3.4 represents a basic block diagram of the designed architecture. The design is primarily split into three different units. The Hough Transform unit calculates *m* and *b* by applying the dual transfer function on the input coordinates based on the input state. The address mapping unit is responsible for two tasks, one is the rounding and conversion of the decimal data to an integer number which represents the two dimensional addresses of the (m,b) pairs. Second, it converts the two addresses for *m* and *b* into one single address using row-major order. Lastly, the memory unit is a dual port ram used as the accumulator to count the votes for various (m,b) pairs. The Hough Transform unit and address mapping units were programmed behaviorally as one unit whereas the memory unit was a separate entity.

3.3.1 The Hough Transform unit

The basic block diagram of the Hough Transform unit is depicted in Figure 3.5. Based on the algorithm, the Hough Transform unit would require basic adders and multipliers for the slope intercept equations. We also need the design to interpret the slope intercept equation it is supposed to use based on the state given by the user input. We do this by designing two, 2-to-1 multiplexers, where the x coordinate and y coordinate are the two inputs and the state is the select line. The input coordinates are sent to the opposite ends in each of the multiplexers. We use two registers “m” and “m_res” for the slope. The goal is to vary “m” from 0 to 1, it is ideal that the resolution with which we vary “m” be as small as possible. This “m” is then multiplied with the output “p” from the first MUX. We then store this multiplied value in a register, where

we multiply it by -1 using 2's complement method. This value "-p*m" is now added to "q" which is the output of the second multiplexer, which then finally gives us the value of "b".

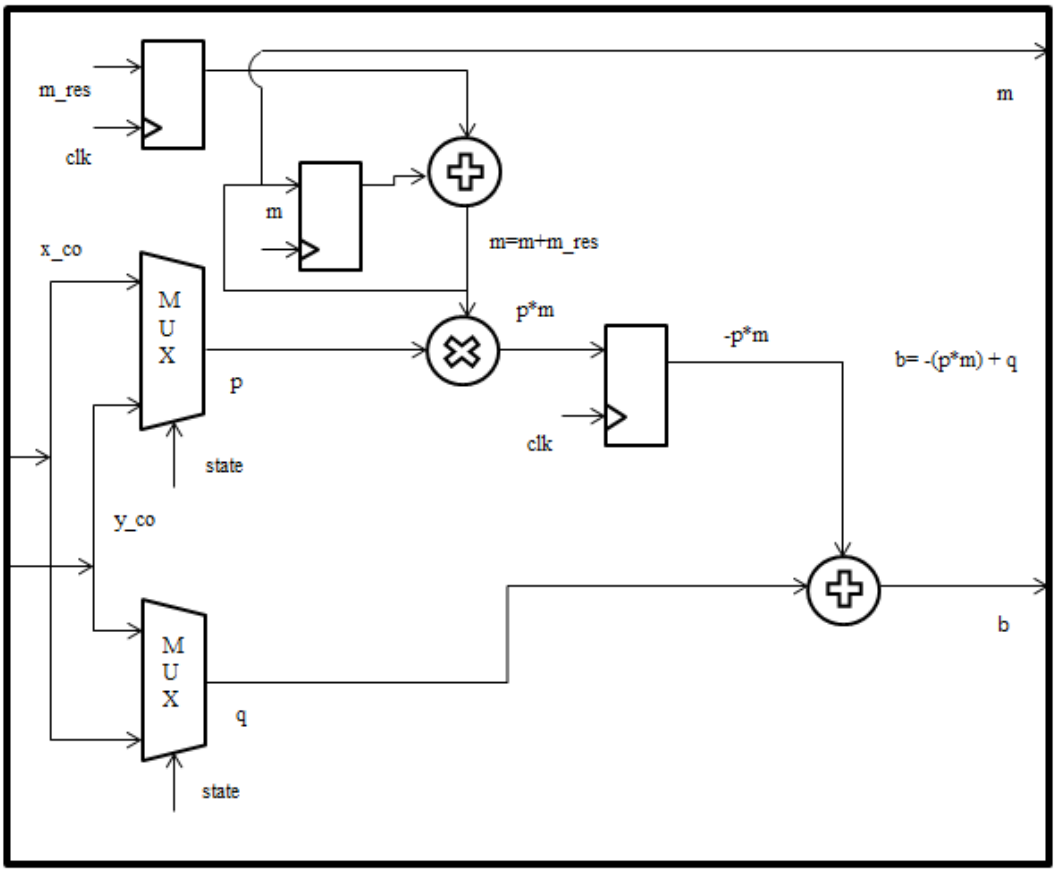

Figure 3.5: Internal architecture of the Hough Transform unit

Thus, the Hough Transform unit calculates the values of "m" and "b" which can now be converted to a ram address for mapping to the accumulator. The design is completely pipelined and synchronous.

3.3.2 Address mapping unit

The address mapping unit is used to convert the calculated *b* and *m* values into an address for the RAM. This is done in two steps.

First, we need to convert the decimal number into an integer. For this we need to multiply the values with the respective size of the accumulator grid. The accumulator grid is a two dimensional array mirroring a Cartesian space. The dimensions of our grid are in the form [2M][M] where *M* is the number of cells in each dimension, the values *b* and *m* must be multiplied with *M*. Since *b* ranges from -1 to 1 and *m* only from 0 to 1 in the grid, we have to further reduce the value *b*M* by *M* to map it accordingly. We add 0.5 to the final value to round it to the nearest integer. Using bit slicing we discard the fractional bits as the integer bits are now sufficient to represent the cell to its nearest value. B_map and M_map refer to the two dimensional addresses of the grid. The calculations are summed up below

For an accumulator grid of dimensions [2M][M],

$$B\_map = (M - (b*M + 0.5))$$

$$M\_map = (m*M + 0.5)$$

In the second step of the address mapping unit we need to convert these two dimensional addresses into a single dimension. This is because two dimensional addressing though available for registers in Verilog is not ideal for synthesis, as FPGA tools tend to build the design using logic rather than the inbuilt memory blocks. We convert the two dimensional addresses into one

single address using row-major order. The size of our RAM remains the same, the only difference being we can linearly address all the data present.

Assuming [i][j] are the two address indices,

$$Address = (j*width) + i$$

The advantage of using this method is that one can convert the one dimensional address back into two dimensions if need be. Thus, the address mapping unit primarily consists of adders, subtractors and multipliers. Again this unit is designed in such a manner that all the data is synchronous and pipelined. We add latency bubbles to some parts to make sure all the data in available in the same clock cycle.

3.3.3 Memory unit

The memory unit is the most crucial part of the design, as all the data we calculate is mapped onto the memory unit. We design a RAM to function as the accumulator grid. This might be time consuming as we have to read from a memory address location and write back into the same memory location. We have to make sure we do this in a fast process, avoiding any timing hazards.

As shown in Figure 3.6, to improve the overall performance and to keep the writing and reading of the memory locations independent of each other, we design a dual port RAM using the Altera megafunction tool. This makes sure that the designed RAM is implemented correctly

in the memory of the board rather than being built through logic. We use memory initialization files to initialize the RAM to all zeros on start up.

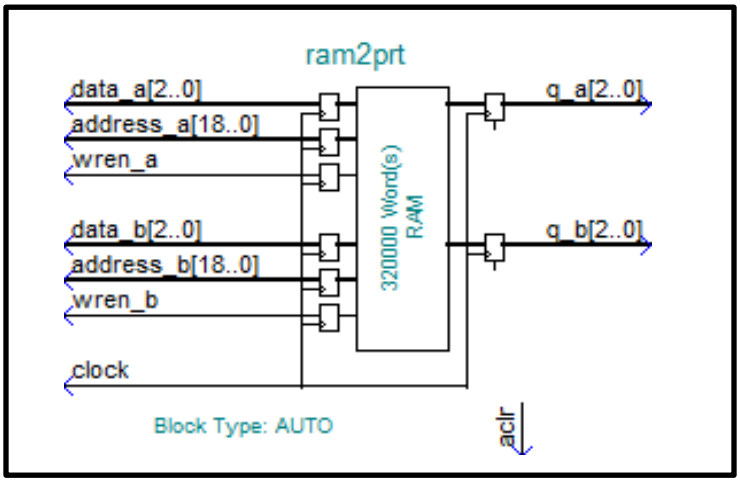

Figure 3.6: Block diagram of the dual port RAM

In our design, we build a [800] x [400] grid, with b ranging from -1 to 1 with a precision of 0.0025 and m ranging from 0 to 1 with a precision of 0.0025. Thus, we would need 320,000 cells to store the vote counts. We assign 3 bits for each word, which can be increased if need be. Writing and reading to the same address can be done using a single port RAM as well, but this would take more time to perform with only one set of ports. So, we prefer to use dual ports. In read during write using dual port memory, data is read in the first rising edge of the clock from one of the ports that is dedicated to reading. The new data is written on the rising edge of the next clock cycle from the other set of ports which are dedicated for writing. The timing diagram in Figure 3.7 clearly shows this.

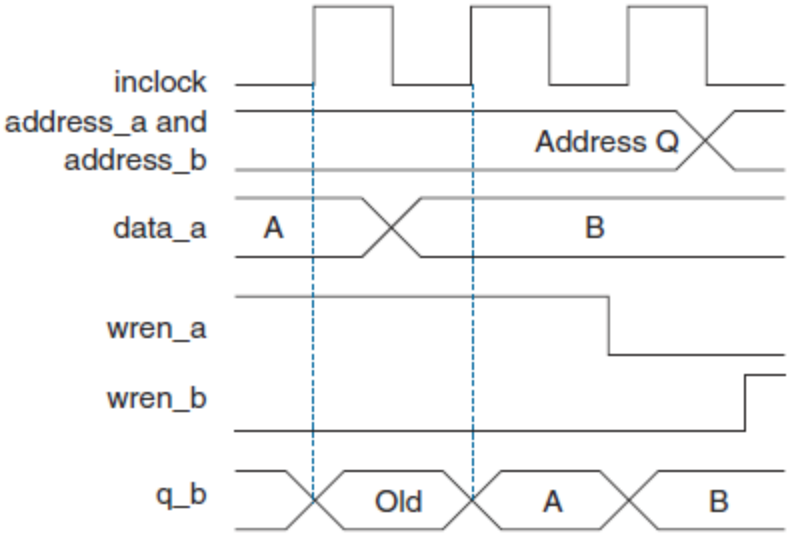

Figure 3.7: Cyclone II mixed port read during write [25]

CHAPTER 4

RESULTS AND ANALYSIS

We designed two versions of the program. Prior to the hardware implementation, we designed the algorithm in the C language. This was primarily done to functionally verify the working of the algorithm and to determine the ideal parameters with which it could detect all the lines to some level of accuracy. Once this is done, we design the hardware in Verilog HDL. We use ModelSim™, which is a HDL simulation tool available freely online for students. We first simulate the algorithm with normal input points. We then add noise by perturbing the input points with errors and then also add other random data points to verify the quality of the line detector. We simulate our design using Altera Quartus II™ which is another tool meant for HDL simulation, with Altera FPGA boards. This tool gives us an idea on the number of logic elements and memory bits the design utilizes. It also does a complete timing and power analysis for the design in the selected model of the FPGA board. We simulate our design on an Altera DSP development kit containing a Cyclone II EP2C70F672C6 chip.

4.1 HDL simulation and results

We first simulate our design to detect six different lines, each from different states, with a couple of states having two lines each. This data is meant to mirror the behavior within a bubble chamber where the movement of a particle is represented by points. Obviously, the same concept

could be extended to line detection with respect to any image. We use 32 input coordinates with 5~6 points for each line. The input points are 32 bit signed fixed point numbers. We also need to specify the domain in which we would like to detect the specified lines.

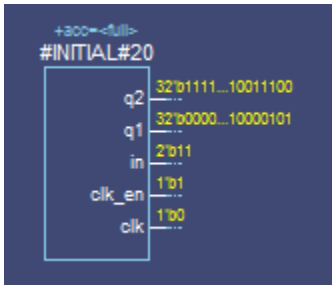

Figure 4.1: Blackbox of the testbench module

Figure 4.1 represents the block diagram of the instantiated test bench module. Here, q1 and q2 are the input coordinates for x and y respectively, in refers to the state or domain in which we want the design to detect lines, and clk and clk_en are the respective clock signals. Figure 4.2 shows the output waveform diagram of all the signals in the Hough Transform unit and address mapping unit of the system. All the important signals have been annotated in the figure. It is important to note that we must know the number of data points we would be reading and add that to the signal count which keeps track of the number of inputs that have changed. This shouldn't be a problem as in most cases we would know this beforehand. For example, if we were reading an image, we would know exactly the number of pixels that are present in the image. The figure also clearly shows the advantages of pipelining the design, it takes 13 clock cycles for the first

pair of “m” and “b” to be computed, whereas after that every pair of “m” and “b” takes only 4 clock cycles. The design is thus almost three times faster thanks to pipelining.

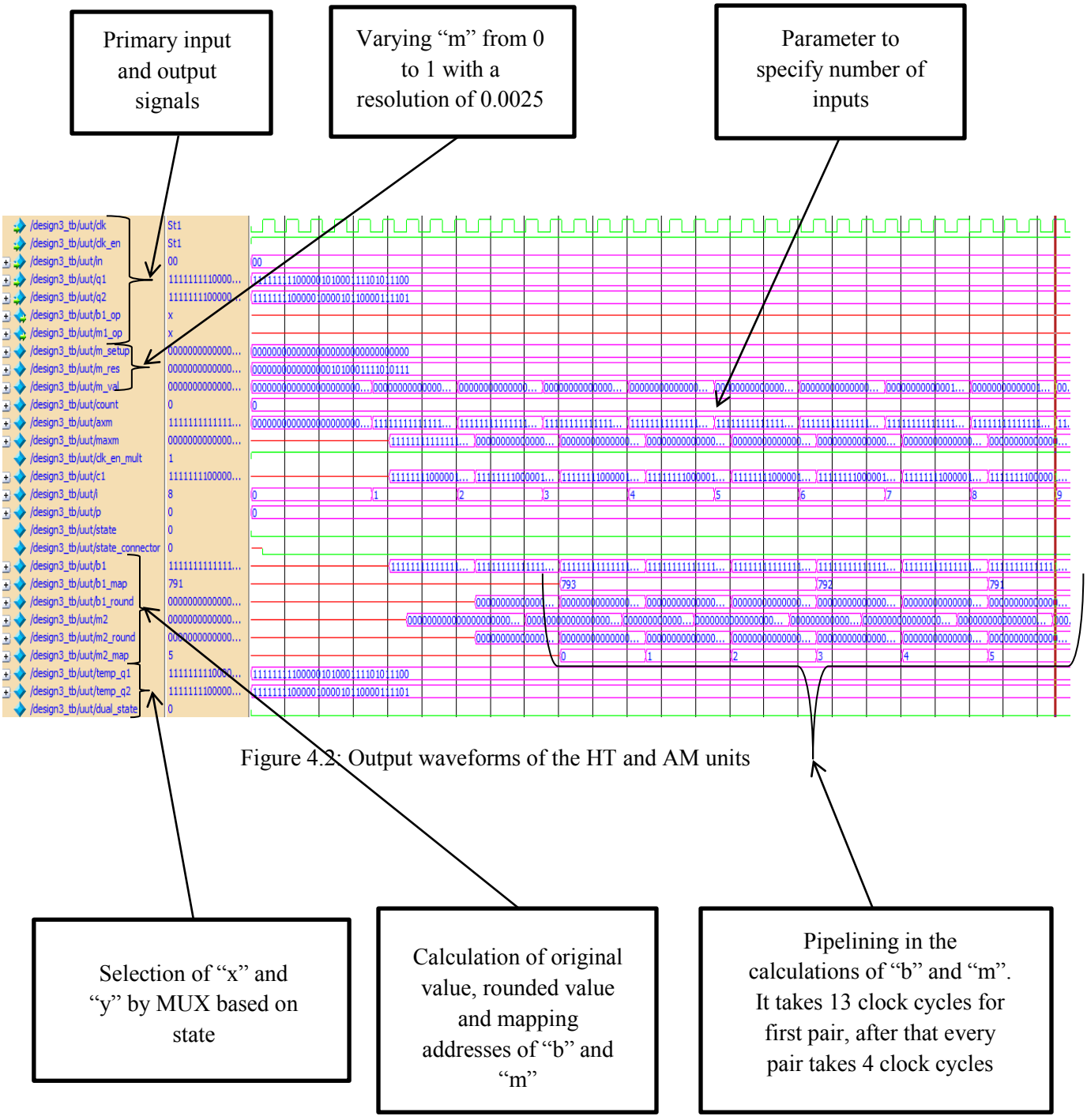

Figure 4.2: Output waveforms of the HT and AM units

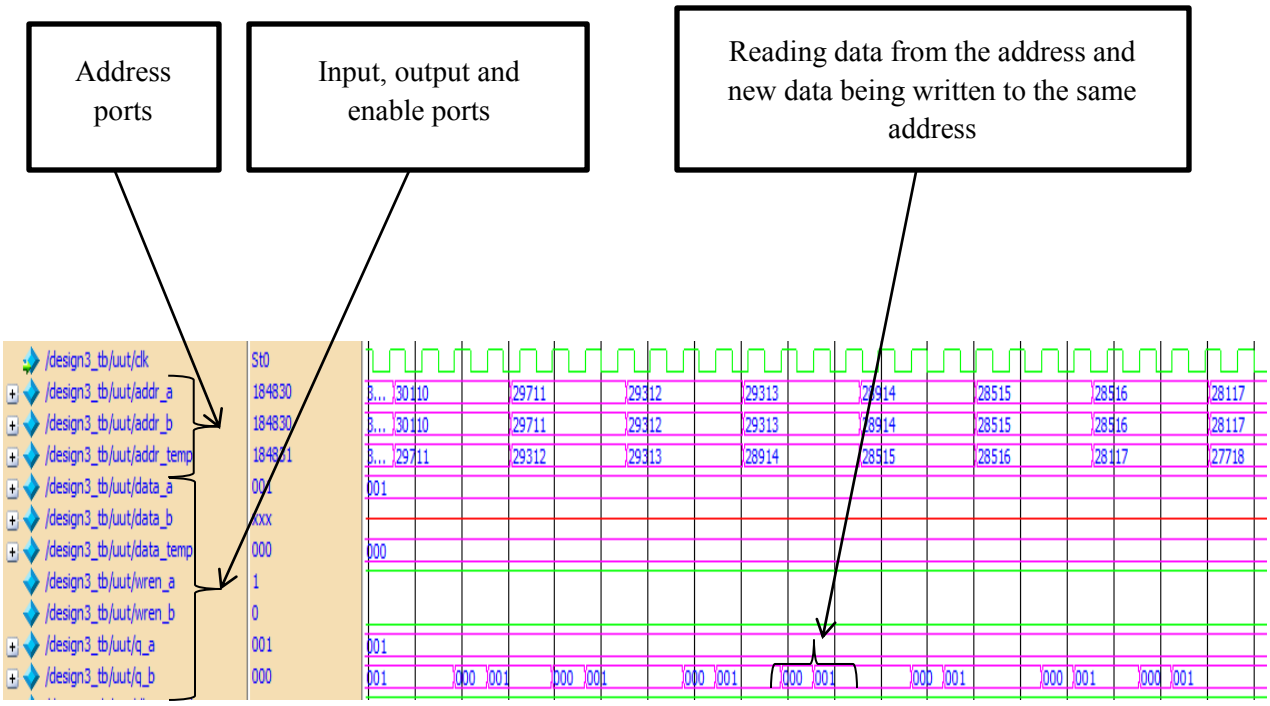

Figure 4.3: Output waveforms of the dual port RAM

Figure 4.3 represents the output waveform signals from the dual port RAM. There are three registers for the addresses, addr_a, addr_b and addr_temp. Addr_a and addr_b are the primary address ports, whereas addr_temp is the address calculated from the “b” and “m” values. If the calculated address is not within bounds, then the value is not passed onto the primary address ports. Port A is strictly used for writing and Port B is strictly used for reading. If the calculated address is within limits, q_b reads the data in the address and the data is passed to data_temp which in turn is added to 3’b001 and passed to data_a. Thus, voting is done by reading and writing to the same memory address. Data is written on the rising edge of the clock.

Thus, by using the dual port RAM we do the reading and writing in an independent and efficient manner.

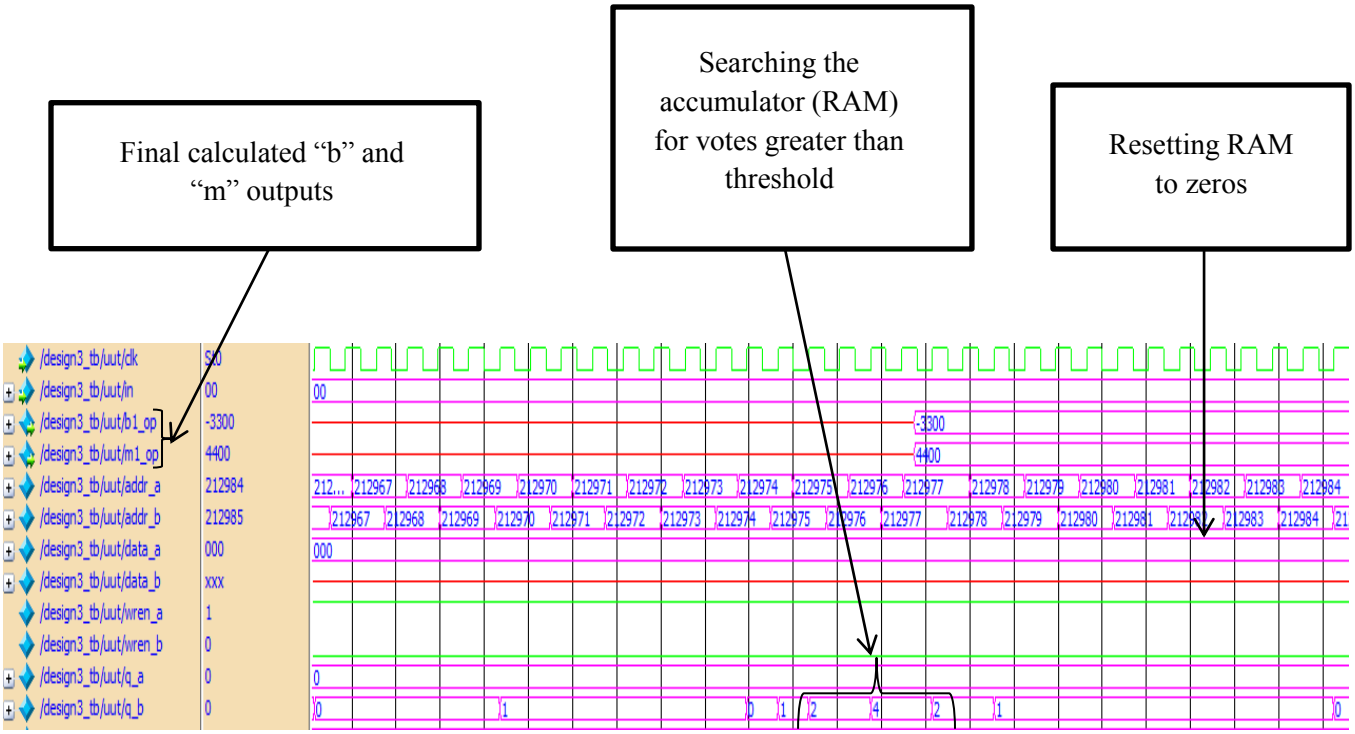

Figure 4.4: Output waveforms showing the searching of the parameter space

Figure 4.4 shows the last few steps of the algorithm process where we search the dual port RAM which is our Hough parameter space. The outputs b1_op and m1_op are represented as whole integer round numbers where the original value is multiplied by 1000. As the figure shows, we use port B to read the data, addr_b is varied and the respective q_b values are checked to see if they are above or below the selected threshold value. We set the threshold value to be 3, thus when q_b reads the value 4 for the specific address, it immediately passes that address to the

output where it is converted back into two dimensions and the respective values of “b” and “m” are output. In this case “b” has the value -0.3300 and “m” is 0.4400. Since the state is 2’b00 which represents state 1 ($0 \leq m \leq 1$), the program displays the equation of the line in correspondence to the values.

We once again take advantage of the dual ports in the RAM, by using the dedicated write port A to reset the all the memories to zeros. We write the zeros to the cells simultaneously during the searching process. This saves a lot of time in the overall algorithm. Once the RAM is completely searched and reset, we can input the data again to detect the lines for another domain by changing the state represented by the signal “in” in the code. Thus, we detect all the lines in the system after traversing through all the 4 states. We could also do the above for only one specific state in which we wish to detect lines for a given set of points. Figure 4.5 shows the final output displayed on the transcript after going through all the states. We can reconstruct the lines by varying either X or Y from -1 to 1.

```
VSIM 121> run
# The equation is in domain 1
# The equation of the line is (0.  -3300 = -(0.   4400 * X) + Y )
# Number of votes=  4
# The equation is in domain 2
# The equation of the line is (0.   1150  = -(0.    6150 * Y)  + X )
# Number of votes=  6
# The equation is in domain 2
# The equation of the line is (0.   1100  = -(0.    1400 * Y)  + X )
# Number of votes=  4
# The equation is in domain 3
# The equation of the line is (0.   6800 = -(0.   2200 * -X) + Y )
# Number of votes=  5
# The equation is in domain 3
# The equation of the line is (0.   5000 = -(0.   2000 * -X) + Y )
# Number of votes=  6
# The equation is in domain 4
# The equation of the line is (0.  -1100 = -(0.   1375 * Y) - X )
# Number of votes=  4
# The equation is in domain 4
# The equation of the line is (0.  -1100 = -(0.   1400 * Y) - X )
# Number of votes=  5
```

Figure 4.5: Output from transcript displaying all the equations detected

We used the above equations and plotted the results with the data points we gave as inputs using Matlab™. In Figure 4.6, we see the plot of the detected lines and the input data points. We can clearly notice that all the lines formed by the input points are detected and no spurious lines are detected. Ideally we would like the line detector to work in situations where the input data is noisy. Thus, we decided to perturb the input data points to replicate real world errors.

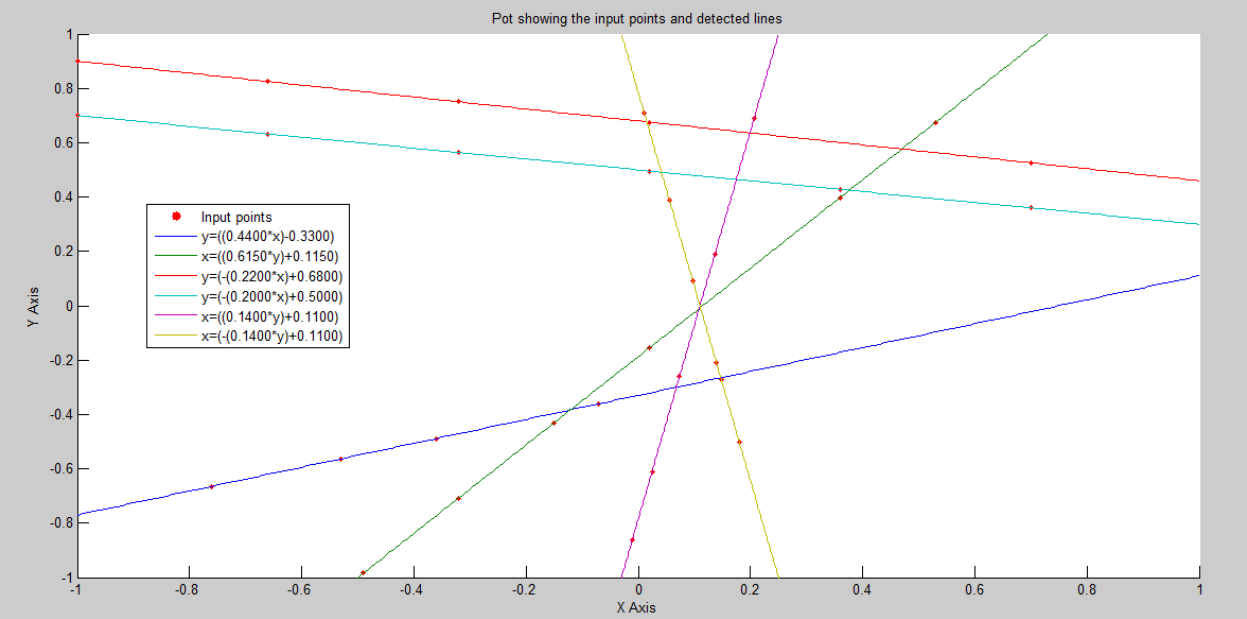

Figure 4.6: Plot showing input points and detected lines

Figure 4.7 shows the plot with respect to perturbed points. The points are randomly perturbed using a random number generator. A few of the detected lines have changed by a cell or two due to the perturbing. Figure 4.8 shows us a zoomed in version of the plot, giving us a

closer look and showing the errors added to the input points. On addition of larger errors, the line detector starts to detect spurious lines or not detect any lines at all.

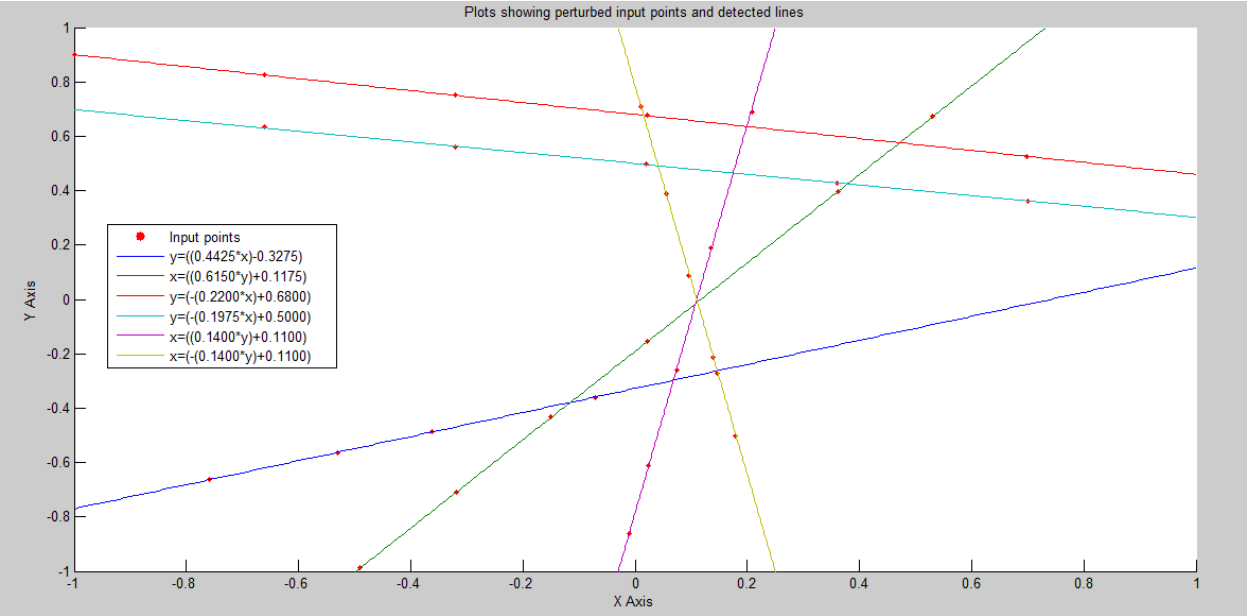

Figure 4.7: Plot showing perturbed input points and detected lines

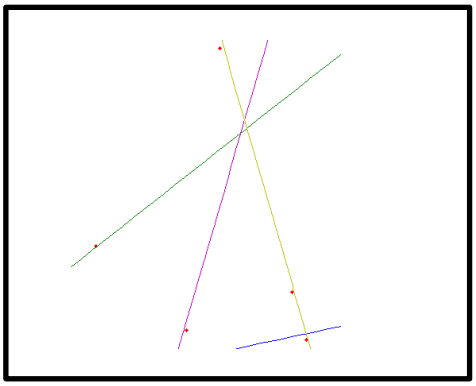

Figure 4.8: Zoomed in version showing the perturbed points

Next, we add random data to the image to verify the functionality of the line detector when other data is present as in the case of most images. This is shown in Figure 4.9.

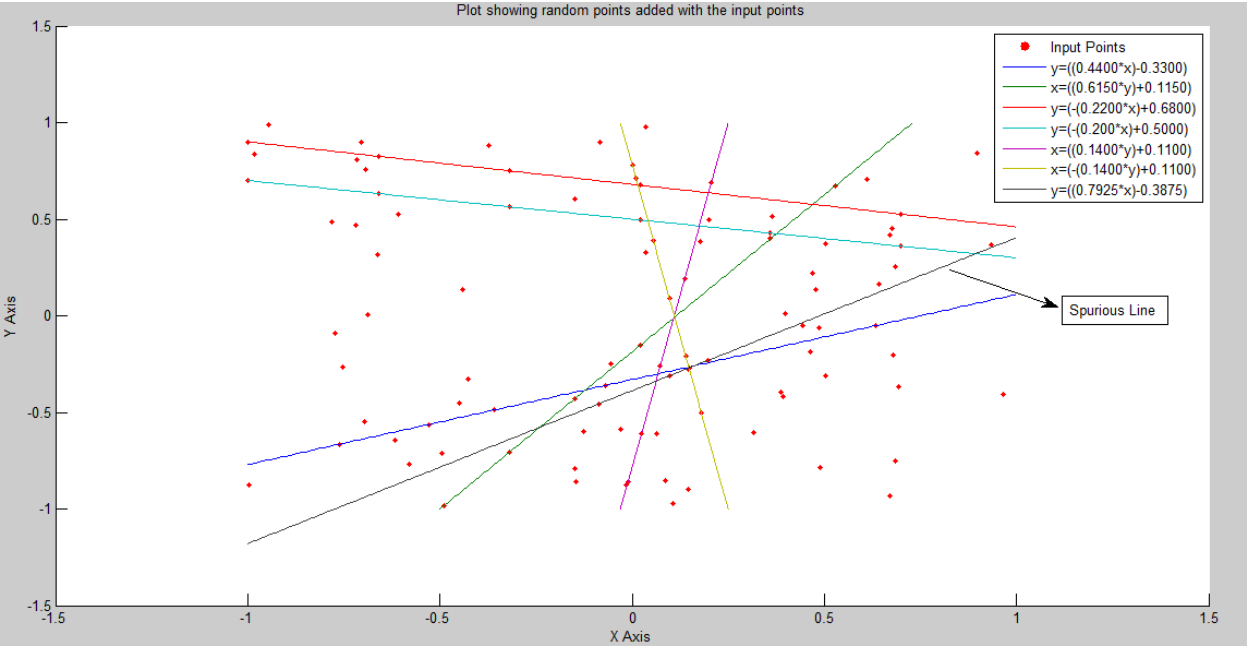

Figure 4.9: Plot with random points added to the input points

On added about 100 random points along with the input points and the line detector detected all the lines correctly. But along with that it also detected a spurious line which was formed by the random data. This problem arises mainly because of lack of enough correct input points. In modern line detection, especially with reference to images, there are many more input points that are extracted from the edges. Since our test case only assumes 4~6 points for the lines, the probability of spurious lines being detected is higher. Thus, in our next test case we add more data points along with a much higher number of error points to show the functionality of the line detector with respect to noise.

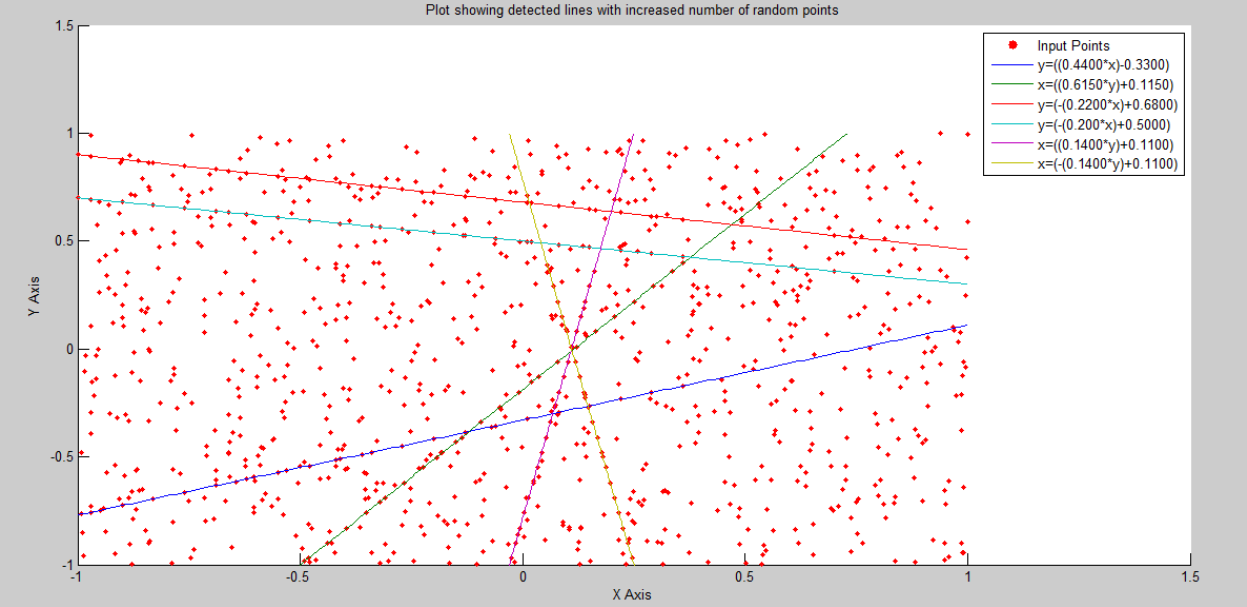

Figure 4.10: Plot with increased number of random input points

We increase the number of data points to around 20 for each line and add 1000 random input points as shown in Figure 4.10. We also perturb the input points by adding small errors. We notice that the line detector detects all the lines correctly and does not detect any spurious lines as it did in the previous scenario. Thus, we can safely say that the line detector wouldn't detect any false lines when used in real world applications, as the detector would have more true data available to it.

4.2 FPGA resource usage

Now that the HDL code is proven to be working successfully, we can determine the feasibility of porting such a design onto an FPGA. The Quartus II tool can directly compile Verilog or VHDL codes. It compiles the code to check if the design is a synthesizable one. Designs that work correctly during simulation do not necessarily work on an FPGA. There are a lot of differences between a simulation and a synthesizable model. The tool also gives warnings with respect to timing analysis and possible logic errors like inferred latches during compilation.

Once all the errors are corrected, the Quartus II tool gives a detailed summary of the design with respect to resource usage, memory usage, timing analysis and power analysis. The tool also builds a schematic view of the RTL level design. Figure 4.11 shows the RTL level diagram of the HT unit. The schematic shows the multiplexers which select the input coordinates based on the input state, this is then sent to adders and multipliers, one of which is pipelined by adding delays with respect to clock cycles. Figure 4.12 shows the RTL level schematic of the memory unit which is primarily made of the dual port RAM. The addresses for reading and writing data are input through the pipelined registers. Finally, Figure 4.13 shows the output of the final “m” and “b” values of the detected lines. The RTL viewer is a very useful tool, as one can make changes to their design by identifying unnecessary logic that is being added inadvertently due to the style of a programmer’s coding.

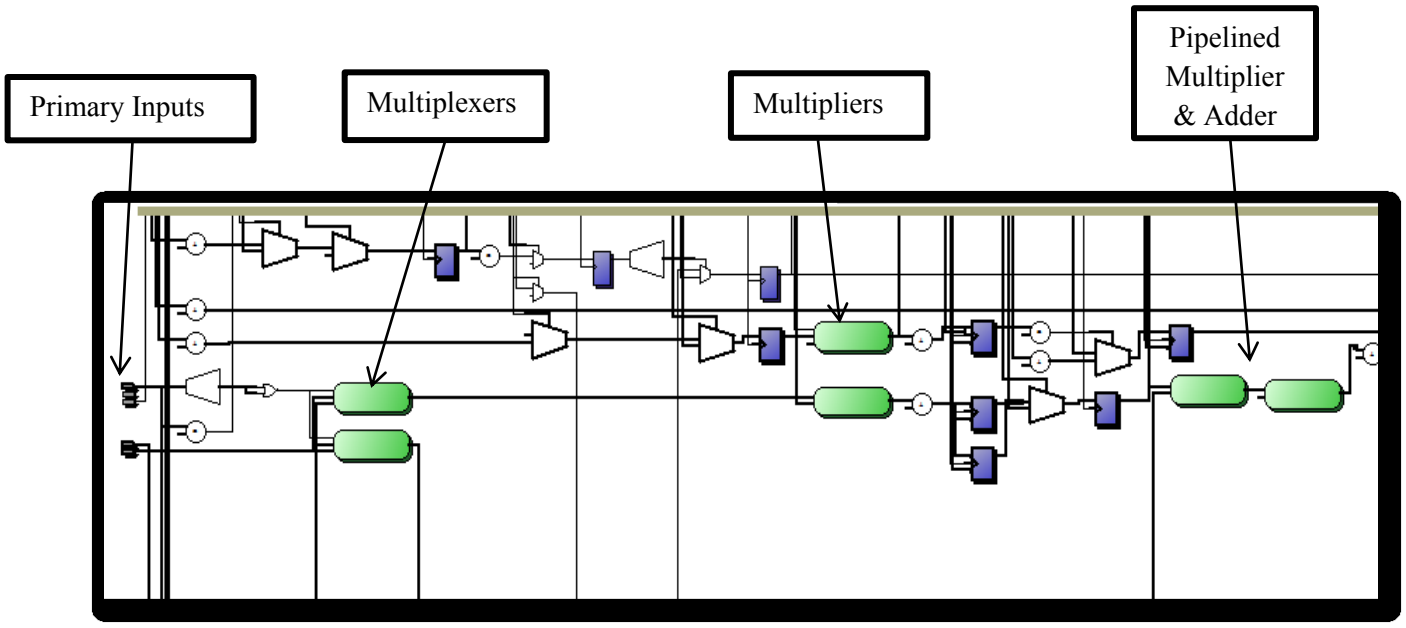

Figure 4.11: Generated RTL design 1

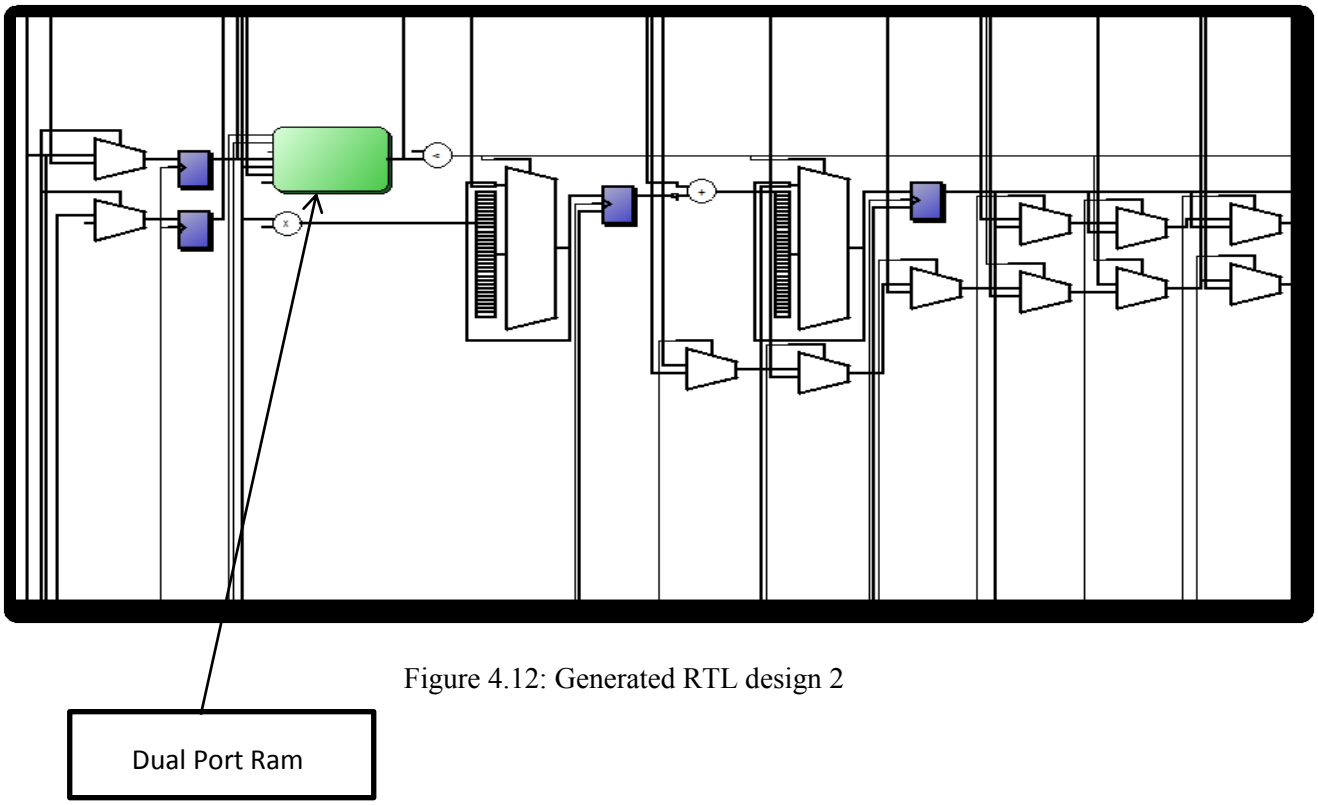

Figure 4.12: Generated RTL design 2

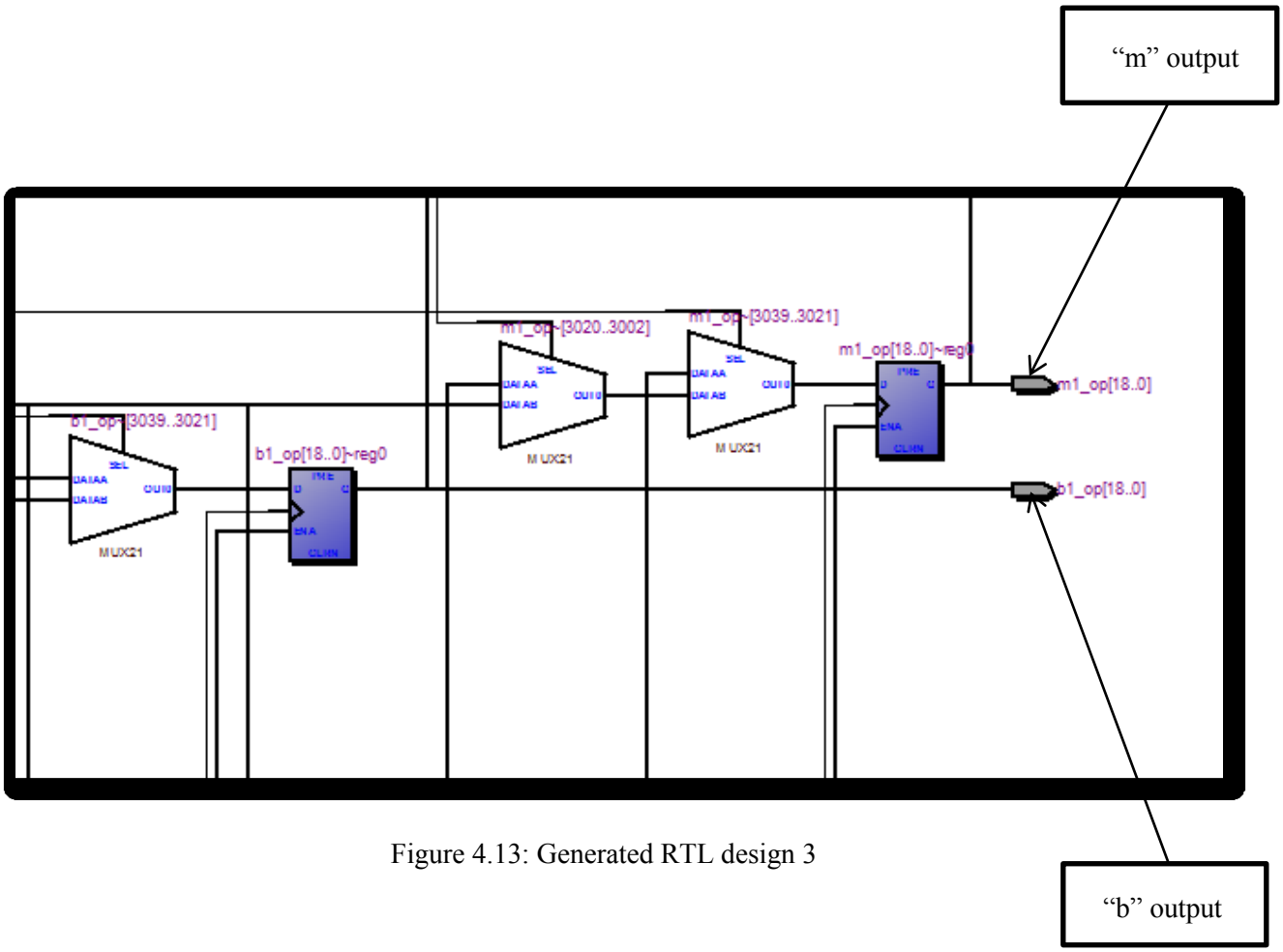

Figure 4.13: Generated RTL design 3

Figure 4.14 shows the flow summary in regards to the FPGA resource usage. As it shows, the design only uses 1093 total logic elements, out of which 978 are combinational functions. This is about 2% of the total available logic blocks on the board. It also uses 960,081 memory bits, which is 83% of the memory available and around 117 kB. The timing analysis tool also determines the maximum frequency of the design to be 62.82 MHz.

Flow Summary	
Flow Status	Successful - Wed Jan 22 18:08:52 2014
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Web Edition
Revision Name	design3
Top-level Entity Name	design3
Family	Cyclone II
Device	EP2C70F672C6
Timing Models	Final
Total logic elements	1,093 / 68,416 (2 %)
Total combinational functions	978 / 68,416 (1 %)
Dedicated logic registers	430 / 68,416 (< 1 %)
Total registers	430
Total pins	106 / 422 (25 %)
Total virtual pins	0
Total memory bits	960,081 / 1,152,000 (83 %)
Embedded Multiplier 9-bit elements	30 / 300 (10 %)
Total PLLs	0 / 4 (0 %)

Figure 4.14: Quartus II flow summary of the design

Based on the frequency, we can estimate the time it takes for completion of Hough Transform for all the states, given an image ported on to the board. The parameter calculation for each input point takes 20.06μs and for completely searching and resetting the accumulator it takes 8ms approximately. Thus, for 32 points the line detector takes about 34.56ms approximately to detect all possible lines in every state which includes all the functions with regards to the memory unit. The line detector spends most of the time reading and resetting the RAM. On increasing the number of input points the time taken for reading and resetting the RAM will remain the same, so it wouldn't increase quite as much. Though, the design is

relatively slow, it is a hardware design meant to perform only one specific task, so it should be acceptable.

4.3 Comparison with other implementations

Since our design is not meant for any singular application but is designed as a generic line detector, the comparative study is based on hardware design parameters. We take into consideration the hardware implementations previously stated in chapter 2. The comparative study is based on four basic criteria, namely, the type of device, number of logic blocks, memory usage and frequency of the design. As seen in Table 4.1, we compare our work with the three implementations previously discussed. The implementation by Karabernou and Terranti [20] was the previously stated gradient methodology and CORDIC structure. Though their approach used very few logic blocks, it has a lower frequency. Due to this the implementation of the real time model did not meet the timing constraints for some image transition standards. Also, the number of logic blocks refers to only the parameter extraction and not the complete architecture. That being said, their proposed architecture could process one pixel of data in less than 88.25ns. In terms of the memory, 8MB refers to the total memory in the device as the authors fail to mention the exact memory their proposed global architecture would require. But since they eventually go on to design two RAMs in the design, we believe it would use much more memory than ours. That being said, it wouldn't be fair to compare our design with a real time architecture in terms of memory. Also, their design requires a lot of extra preprocessing in terms of extracting the horizontal and vertical gradients.

Tagzout et al. [18] used an architecture based on the incremental Hough Transform model. It is important to note that their model is constrained to only determining the value of ρ and θ. Since there is no need to map the values onto a memory, the number of logic blocks used is very small. Again, the frequency of their design is relatively slower than the one we have designed. Also, since their design incorporates Sine and Cosine look up tables, making it an approximated approach, we believe our approach would be much more accurate.

	Karabernou and Terranti [20]	**Tagzout et al. [18]**	**Zhou et al. [22]**	**This work**
Device	XC4010EPC84	XC4010XL	XC6VLX240T-1	EP2C70F672C6
Logic Blocks	205 CLBs	324 CLBs	14493 Slices	1093 CLBs
Memory	8 MB (Available memory in device)	-	180, 18 kbit block RAMs	118 kB – 196 kB
Frequency	23.116 MHz	27.1 MHz	245.519 MHz	62.82 MHz

Table 4.1: Comparison with other implementations

Zhou et al. [22] implemented the algorithm using DSP blocks and block RAMs. This is by far the best implementation in terms of speed and throughput of the design. Their implementation uses a Virtex-6 board which is a medium to high end FPGA board in terms of cost. Their design performed Hough Transform on 33232 edge points in 135.75μs. This is definitely impressive in terms of speed, but their design uses 14493 slices of logic. For a Virtex 6 FPGA, a slice is made out of 4 LUTs, multiplexers and a arithmetic carry logic. Two such slices form a CLB [26]. This would mean their design uses approximately 7247 CLBs, which is much larger than ours. Also, their design incorporates 180, 18Kbit RAM blocks which is almost 405 kB. In our approach we use only 118 kB and even in the design with a higher number of input points like the example shown in Figure 4.10, the maximum memory would only be 196 kB. Thus, though our design is not as fast, it performs the Hough Transform, making best use of memory without using too many logic elements.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this chapter we will briefly summarize the work done and look into areas in the research which we could develop or improve upon in future work.

5.1 Summary of work

When we started this thesis our main aim was to look for a simplified line detection algorithm that is easy to implement in hardware because the Standard Hough Transform comes with added complexities and computational overhead. We went about doing this by using the parabolic form of the Hough Transform. The parabolic form of the Hough Transform was not used earlier due to its inability to detect vertical lines. We solve this conundrum by doing a rotation of the axis and incorporating two forms of slope intercept equation. By using this parabolic form, we successfully avoid the use of trigonometric functions which makes our line detector far more accurate. We also add another alteration to our algorithm by reflecting the input points to (-x,y). This ensures the value of the slope is always positive. This means that our accumulator memory needs to be only half the size it was previously. Our final design was made of four different cases, to detect four different slope orientations of the line. We incorporated a synchronous pipelined design. Our design consisted of three parts, the Hough Transform unit, address mapping unit and memory unit. We verified the functionality of the line detector through

experiments. We added random noise and perturbed the points to make sure the design could withstand such real world factors. We also compared our design with other implementations. It was noticed that this implementation made the best use of memory in the board. Though it isn't the fastest implementation of the Hough Transform, it finds a middle ground in terms of speed and resource usage.

One of the other biggest advantages of our design would be with respect to its modularity. It can easily be changed for the required application. If the target application is time critical we could compensate on memory and build four units to work simultaneously to detect lines for all the 4 states. We could also, if need be, increase or decrease the size of memory unit for the required accuracy in detecting the lines.

5.2 Future work

There are some areas in the design that can still be improved and there are also some advances that one can look at. Some of these are presented below

1. In terms of the design itself, we notice that a large part of the memory is made up of zeros during searching. This is wasted memory. In software we can easily go around this by building a sparse matrix. In hardware though it is not so simple. Exploring different ways to build a dynamically allocated memory unit could definitely improve the algorithm, as it would also reduce our parameter search space.

2. It is important to note that the frequency of the design is estimated based on the placement and routing of the Altera Quartus II tool. This frequency could be increased. For example, we could look at creating a custom design which could give us a higher frequency.
3. While searching the accumulator for possible cells with high votes, we notice that some votes are divided across neighbors of the main cell and this might cause additional lines to be created. We could look into a more intelligent way to extract the local maxima by smoothing the accumulator space. In cases where the neighboring cells also have high values we would be better off taking an average value for the final coordinates. In terms of reducing the errors in the parameter space, we could design a process where, once we find a global maximum, we could remove all the votes associated with that point before searching for the other solutions. Guo et al. [27] have also recently introduced a method incorporating weights for edge points. These weights are later used to accumulate votes in the parameter space. This method is shown to remove the possibility of detecting false peaks.
4. Pre-processing in terms of image filtering and quality of edge detection is a large area of research by itself. It would be interesting to see how this line detector works when used along with a larger system that involves image processing.

5. In terms of the concept, we could improve on this approach to detect line segments rather than lines. This is usually done by keeping track of the points contributing votes to their respective cells.

REFERENCES

[1] P. V. C. Hough, “Method and means for recognizing complex patterns,” U.S. Patent 3 069 654, December 18, 1962.

[2] A. Rosenfeld, *Picture Processing by Computer*. New York: Academic Press, 1969, pp.335-336.

[3] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Commun ACM,* vol. 15, pp. 11-15, 1972.

[4] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays,* 1998, pp. 191-200.

[5] J. Illingworth and J. Kittler, "A survey of the Hough Transform," *Computer Vision, Graphics, and Image Processing,* vol. 44, pp. 87-116, 1988.

[6] P. T. Kuruganti and H. Qi, "Asymmetry analysis in breast cancer detection using thermal infrared images," in *Engineering in Medicine and Biology, 2002. 24th Annual Conference and the Annual Fall Meeting of the Biomedical Engineering Society EMBS/BMES Conference, 2002. Proceedings of the Second Joint,* 2002, pp. 1155-1156.

[7] S. Golemati, J. Stoitsis, E. G. Sifakis, T. Balkizas and K. S. Nikita, "Using the Hough Transform to segment ultrasound images of longitudinal and transverse sections of the carotid artery," *Ultrasound Med. Biol.,* vol. 33, pp. 1918-1932, 2007.

[8] N. Fitton and S. Cox, "Optimising the application of the Hough Transform for automatic feature extraction from geoscientific images," *Comput. Geosci.,* vol. 24, pp. 933-951, 1998.

[9] J. Yuan, M. Li, H. Qingqing and G. Yan, "Research on object shape detection from image with high-level noise based on fuzzy generalized Hough Transform," in *Multimedia and Signal Processing (CMSP), 2011 International Conference on,* 2011, pp. 209-212.

[10] http://www.eetimes.com/document.asp?doc_id=1274496, accessed 12/23/2013.

[11] T. Tuytelaars, L. Van Gool, M. Proesmans and T. Moons, "The cascaded Hough Transform as an aid in aerial image interpretation," in *Computer Vision, 1998. Sixth International Conference on,* 1998, pp. 67-72.

[12] H. Li, M. A. Lavin and R. J. Le Master, "Fast Hough Transform: A hierarchical approach," *Computer Vision, Graphics, and Image Processing,* vol. 36, pp. 139-161, 1986.

[13] S. El Mejdani, R. Egli and F. Dubeau, "Old and new straight-line detectors: Description and comparison," *Pattern Recognit,* vol. 41, pp. 1845-1866, 2008.

[14] A. Inselberg, "The plane with parallel coordinates," *The Visual Computer,* vol. 1, pp. 69-91, 1985.

[15] L. A. Fernandes and M. M. Oliveira, "Real-time line detection through an improved Hough Transform voting scheme," *Pattern Recognit,* vol. 41, pp. 299-314, 2008.

[16] P. Bhattacharya, A. Rosenfeld and I. Weiss, "Point-to-line mappings as Hough Transforms," *Pattern Recog. Lett.,* vol. 23, pp. 1705-1710, 2002.

[17] L. M. Murphy, "Linear feature detection and enhancement in noisy images via the Radon Transform," *Pattern Recog. Lett.,* vol. 4, pp. 279-284, 1986.

[18] S. Tagzout, K. Achour and O. Djekoune, "Hough Transform algorithm for FPGA implementation," *Signal Process,* vol. 81, pp. 1295-1301, 2001.

[19] H. Dawid and H. Meyr, "CORDIC algorithms and architectures," *Digital Signal Processing for Multimedia Systems,* pp. 623-655, 1999.

[20] S. M. Karabernou and F. Terranti, "Real-time FPGA implementation of Hough Transform using gradient and CORDIC algorithm," *Image Vision Comput.,* vol. 23, pp. 1009-1017, 2005.

[21] J. A. Kalomiros and J. Lygouras, "Design and evaluation of a hardware/software FPGA-based system for fast image processing," *Microprocessors and Microsystems,* vol. 32, pp. 95-106, 2008.

[22] X. Zhou, N. Tomagou, Y. Ito and K. Nakano, "Efficient Hough Transform on the FPGA using DSP slices and block RAMs," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International,* 2013, pp. 771-778.

[23] J.W. Smith, "Points and Lines in the Plane," M.S Thesis, Dept. Computer Science, University of Cincinnati, OH, 2010.

[24] L. Da Fontoura Costa and M. B. Sandler, "A binary Hough Transform and its efficient implementation in a systolic array architecture," *Pattern Recog. Lett.,* vol. 10, pp. 329-334, 1989.

[25] Altera, "Cyclone II Memory Blocks", February 2008. Available: http://www.altera.com/literature/hb/cyc2/cyc2_cii51008.pdf, accessed 1/12/2014.

[26] Xilinx, "Virtex-6 Family Overview", January 19th 2012. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, accessed 1/18/2014.

[27] S. Guo, T. Pridmore, Y. Kong and X. Zhang, "An improved Hough Transform voting scheme utilizing surround suppression," *Pattern Recog. Lett.,* vol. 30, pp. 1241-1252, 2009.

[28] https://www.semiwiki.com/forum/content/1596-brief-history-field-programmable-devices-fpgas.html, accessed 12/23/2013.