19L820 – PROJECT WORK II

VLSI IMPLEMENTATION OF IMAGE RECTIFICATION FOR STEREO IMAGING IN COMPUTER VISION

SHEENA S (20L141)
TARUNISREE A V (20L149)
NATHEESHKUMAR B (21L410)
SENTHILNATHAN B (21L417)

Dissertation submitted in partial fulfilment of the requirements for the degree of

BACHELOR OF ENGINEERING
Branch: ELECTRONICS AND COMMUNICATION ENGINEERING
of Anna University

APRIL 2024

ELECTRONICS AND COMMUNICATION ENGINEERING
PSG COLLEGE OF TECHNOLOGY
(Autonomous Institution)

COIMBATORE – 641 004

PSG COLLEGE OF TECHNOLOGY

(Autonomous Institution)

COIMBATORE – 641004

VLSI IMPLEMENTATION OF IMAGE RECTIFICATION FOR STEREO IMAGING IN COMPUTER VISION

Bonafide record of work done by

SHEENA S	(20L141)
TARUNISREE A V	(20L149)
NATHEESHKUMAR B	(21L410)
SENTHILNATHAN B	(21L417)

Dissertation submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF ENGINEERING

Branch: ELECTRONICS AND COMMUNICATION ENGINEERING

of Anna University

……..…	..……..……
Dr. S. HEMA CHITRA	**Dr. V. KRISHNAVENI**
Faculty Guide	Head of the Department

Certified that the candidate was examined in the viva-voice examination held on

……

…….	……
(Internal Examiner)	(External Examiner)

CONTENTS

ACKNOWLEDGEMENT

We sincerely owe our gratitude to **Dr. K. Prakasan**, Principal, PSG College of Technology, Coimbatore, for providing us with the best teachers and facilities.

It is our privilege to place on record our sincere thanks to **Dr. V. Krishnaveni**, Professor and Head of the Department, Department of ECE, PSG College of Technology, Coimbatore, for her support and guidance.

We would like to place our everlasting gratitude and indebtedness to our Programme Coordinator **Dr.K.V.Anusuya**, Associate Professor, Department of ECE for her excellent advice, guidance and continued assistance throughout the course of this project.

We would like to express our sincere gratitude to our project guide, **Dr. S. Hema Chitra,** Associate Professor, Department of Electronics and Communication Engineering, for her constant motivation, direction and guidance through each step of our work.

We express our sincere gratitude to our Tutor **Dr. S. Hema Chitra**, Associate Professor, PSG College of Technology, Coimbatore, for her encouragement and support towards this project work.

We kindly acknowledge the timely help and valuable suggestions of our Professors, PG scholars and non-teaching staff of the department. Our sincere and heartfelt thanks to our classmates for their moral support and encouragement to complete this work.

SYNOPSIS

Computer Vision is an emerging field in recent times in the domains of artificial intelligence and autonomous systems. One of the challenges that are faced by today's industries is the increased development time of computer vision based 3d reconstruction systems. This project provides a simple solution by reducing the development time for Image preprocessing. Camera lenses can introduce distortions, such as barrel or pincushion distortions, which affect the accuracy of depth information. For stereo image pairs, it's essential to rectify the images and align their epipolar lines. The objective of the project is to develop Area and Power efficient image rectification algorithm for Stereo Imaging in Computer Vision.

The proposed work utilizes a special kind of interpolation technique called the bicubic interpolation rather than bilinear interpolation that is present in existing algorithm. Bicubic interpolation is more efficient than Bilinear interpolation, which uses 16 neighbouring pixel values compared with the 8 pixel values used by Bilinear for interpolation. The project follows the Image Rectification algorithm where left and right images captured by two different cameras are taken as input and they are combined together and then they are rectified using homographic transformation. After that they are checked for undistortion using the parameters that are obtained from Camera Calibration and the pixels are retained by interpolation. Thus the final output image is free of any kind of distortions like tangential and radial components introduced by the camera lens. Algorithm implementation is done using Matlab and Simulink. Each Simulink block contains the functionality of each stages of the Algorithm. The entire Simulink model is then converted into Verilog code using HDL Coder which is a toolbox present within Matlab HDL Toolbox. The code is then implemented in Xilinx Vivado Software for Area, Power and Synthesis Analysis.

Thus the proposed algorithm is Area and Power efficient having 61.764% decrease in LUT, 5.35% decrease in Power and 49.4% decrease in Average Mean Error. Some of the applications include Virtual Reality content creation, Object Detection, Gesture Recognition and Human Computer Interaction in Computer Vision. In future the algorithm can be implemented on FPGA for real time image processing.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Computer vision is a field of computer science that enables machines to interpret and make decisions based on visual data. It involves the development of algorithms and systems that allows computers to understand and extract information from images or videos [2]. Computer vision is the interdisciplinary field that enables computers to gain a high-level understanding of digital images or videos [2]. It seeks to automate tasks that the human visual system can do, such as image recognition, object detection, and scene understanding. Significant progress has been made in the domains of autonomous systems and artificial intelligence in recent times [2].

1.1 SIGNIFICANCE OF COMPUTER VISION

Computer vision is a field that has seen significant advancements in recent years, particularly in the context of artificial intelligence and autonomous systems [1]. It involves the use of mathematical principles and software or hardware algorithms to process and interpret visual data from the real world [3]. This data can come from sources such as images or videos, and computer vision techniques are used to extract meaningful information from this visual input [2]. One of the key applications of computer vision is in the development of systems for tasks such as object recognition, image classification, and scene understanding. The field also involves the use of technologies like FPGAs (Field Programmable Gate Arrays) for efficient and high-throughput processing of visual data [3]. Overall, computer vision plays a crucial role in enabling advancements in Areas such as robotics, autonomous vehicles, augmented reality, and more. Computer vision is one of the fields driving these advancements in technology.

In brief, computer vision allows an electronic system to "see" the environment in the same way that a human does [1]. The subset of computer vision that uses two or more cameras is the subject of this project. One advantage of employing a multi-camera system is that crucial environmental data, such as depth, can be obtained without the use of active sensing components, which are necessary for LIDAR and structured light techniques [2]. To develop a functional stereo camera system, however, a few technical challenges including camera calibration need to be addressed [3]. Determining the camera parameters required for image rectification and undistortion involves camera calibration procedure [1]. This study presents an efficient method for modeling, generating, targeting, and validating an FPGA-based stereo video preprocessing system using Model Based Design in conjunction with the MathWorks HDL, CoderTM tool flow.

1.2 APPLICATIONS OF COMPUTER VISION

Computer vision finds applications in various fields, offering solutions that range from enhancing industrial processes to enriching everyday experiences [2]. Here are some applications relevant to our project on image rectification and undistortion algorithms [2].

1.2.1 AUTONOMOUS VEHICLES

Computer vision processes images from cameras mounted on vehicles to detect and recognize objects such as pedestrians, vehicles, and road signs [1]. Fig1.1 shows application of the project in Autonomous Vehicle Stereo vision system that uses image rectification and undistortion algorithms to calculate depth information, vital for understanding the 3D structure of the environment. Accurate depth perception helps vehicles navigate safely by detecting obstacles and planning optimal paths [2].

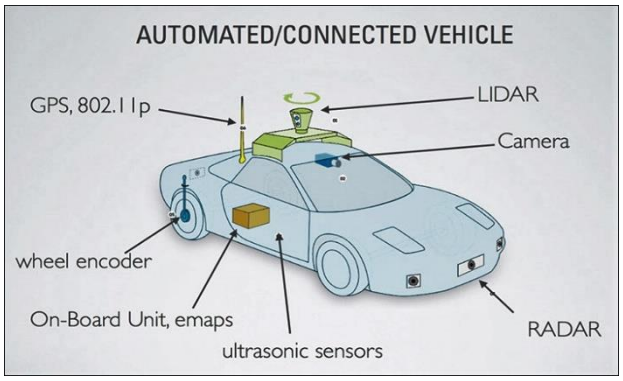


Fig 1.1 Autonomous Vehicle

1.2.2 AUGMENTED REALITY

Image rectification and undistortion ensure precise alignment of virtual objects with real-world scenes, enhancing the immersion and realism of augmented reality experiences [2]. Computer vision enables users to interact with virtual elements overlaid on the physical environment, such as selecting objects or manipulating virtual controls [2]. Fig 1.2 shows application of the project in Augmented Reality. By analyzing real-time images, augmented reality systems can recognize objects and surfaces, allowing for dynamic placement of virtual content.

Fig 1.2 Augmented Reality

1.2.3 MEDICAL IMAGING

Computer vision algorithms analyze medical images to identify abnormalities, assist in disease detection, and provide quantitative assessments of anatomical structures [2]. Image rectification and undistortion enhance the accuracy of preoperative imaging, aiding surgeons in planning procedures and minimizing risks [2]. Fig 1.3 shows how computer vision image processing is used in Medical Imaging. By tracking changes in medical images over time, computer vision helps evaluate treatment effectiveness and disease progression [1].

Fig 1.3 Medical Imaging

1.3 STEREO IMAGING

Stereo imaging, a crucial aspect of computer vision, simulates human binocular vision to perceive depth in images [2]. It relies on a pair of cameras, mimicking the human eyes, to capture two slightly offset images of the same scene [1]. These images, when processed, enable the system to calculate disparities between corresponding points, which are used to reconstruct a 3D representation of the scene. Image rectification and undistortion algorithms play a vital role in stereo vision by aligning and correcting images to ensure accurate depth estimation [2]. These algorithms rectify the images to a common reference frame, removing distortions caused by camera lenses and enabling the precise matching of image features for depth calculation [1].

1.3.1 APPLICATIONS OF STEREO IMAGING

Stereo imaging facilitates the creation of immersive virtual reality content by capturing scenes from multiple viewpoints, allowing for the generation of realistic 3D environments and experiences [2]. Stereo vision systems capture stereoscopic 360-degree video footage, enhancing the depth perception and immersion of virtual reality experiences, from gaming and entertainment to training and education [1].

1.3.1.1 3D RECONSTRUCTION

In depth Mapping, Stereo vision systems calculate the disparity between corresponding points in stereo image pairs, enabling the estimation of depth information and the reconstruction of 3D scenes [2]. In Modelling by capturing multiple viewpoints, stereo vision systems create detailed 3D models of

environments, valuable for virtual reality simulations, urban planning, and for archaeological documentation [1].

1.3.1.2 ROBOTICS AND AUTONOMOUS SYSTEMS

In obstacle Avoidance, Stereo vision enables robots and autonomous vehicles to perceive their surroundings in 3D, facilitating obstacle detection and navigation in dynamic environments [3]. In manipulation, Stereo vision systems provide accurate depth perception for robotic arms, assisting in tasks such as grasping, manipulation, and assembly in industrial and service robotics [2].

1.3.1.3 AUGMENTED REALITY AND MIXED REALITY

In spatial anchoring, Stereo vision aligns virtual content with real-world scenes, enhancing the realism and accuracy of augmented reality experiences by ensuring precise registration and interaction with the physical environment. In immersive gaming, Stereo vision technologies enable immersive gaming experiences by overlaying virtual objects onto the real world, allowing users to interact with digital elements in a 3D space.

1.3.1.4 GESTURE RECOGNITION AND HUMAN-COMPUTER INTERACTION

In natural Interaction, Stereo vision enables intuitive gesture-based interfaces by accurately capturing hand movements and gestures in 3D space, facilitating natural interactions with computers, gaming consoles, and smart devices [2]. In biometric authentication, Stereo vision systems analyse facial depth information for biometric authentication purposes, offering robust and secure identity verification in access control systems and electronic devices [2].

1.3.1.5 QUALITY CONTROL AND INSPECTION

In dimensional Inspection, Stereo vision systems measure object dimensions and geometrical features with high accuracy, ensuring quality control in manufacturing processes by detecting deviations and defects [2]. In surface defect detection, Stereo vision algorithms detect surface irregularities and defects on manufactured parts or products, enabling automated inspection and quality assurance in industrial settings [2].

1.4 IMAGE DISTORTIONS

Image distortions can arise as a result of various defects in camera lenses, influencing the quality and fidelity of captured images [2]. Lens imperfections can manifest in several forms, impacting the accuracy and clarity of the visual information [1]. One prevalent type of distortion is geometric distortion, which includes barrel distortion and pincushion distortion [1]. Barrel distortion causes straight lines to curve outward towards the edges of the image, resembling the shape of a barrel, while pincushion distortion results in lines curving inward, creating a pinched appearance [3]. Chromatic aberration is another common issue, arising from the lens's inability to focus different wavelengths of light at the same point, leading to color fringing along edges [2]. Spherical aberration, caused by variations in the lens's curvature, can result in blurring or smearing of image details [2]. Additionally, lens vignetting may occur, causing a darkening of

image corners due to uneven light distribution [1]. These distortions not only compromise the visual quality but also impact the accuracy of computer vision algorithms that rely on precise image information [1].

To mitigate these effects, advanced lens technologies, such as aspherical lens elements and low-dispersion glass, are employed in modern camera systems, supplemented by post-processing techniques and calibration methods to correct distortions and enhance overall image quality. Rectification and undistortion are crucial preprocessing steps in preparing images for computer vision applications, as they play a pivotal role in ensuring the accuracy and reliability of subsequent algorithms [2]. The distortions introduced by camera lenses, such as geometric distortions (barrel and pincushion distortions), chromatic aberration, and vignetting, can significantly impact the geometric and photometric properties of captured images [2]. Fig 1.4 shows the different types of distortions. In computer vision tasks, where precise spatial relationships and accurate feature extraction are essential, rectifying and undistorting images become imperative. Geometric distortions can distort the shapes and proportions of objects, hindering object recognition and localization algorithms [2].

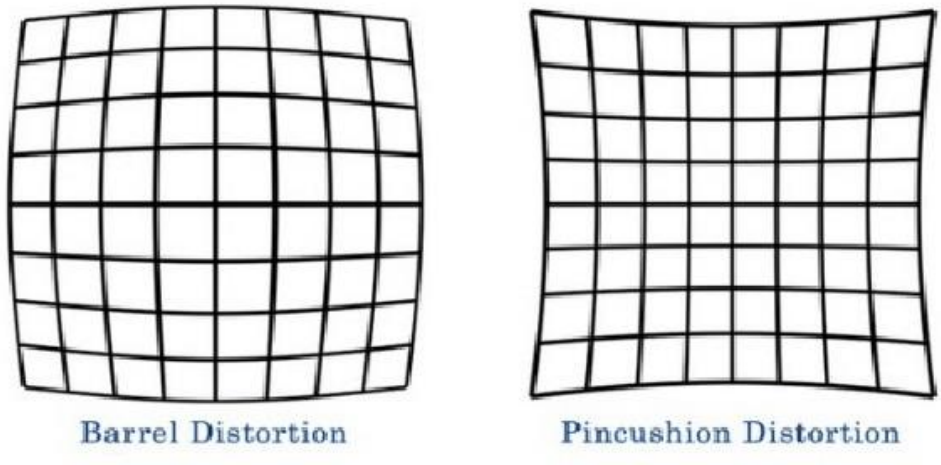


Fig 1.4 Types of Distortion

1.4.1 NEED FOR IMAGE RECTIFICATION

The need for image rectification arises from the relative offsets of two image sensors as well as the radial and tangential effects caused by the camera lenses [2]. These distortions can lead to inaccuracies in the captured images, affecting the quality and reliability of computer vision systems [2]. Image rectification aims to overcome these distortions by creating new images that align the two input images and correct for the radial and tangential effects, ultimately improving the accuracy and reliability of computer vision algorithms [2]. Fig 1.5 shows the rectification of the images through the alignment of epipolar lines obtained from the images [2].

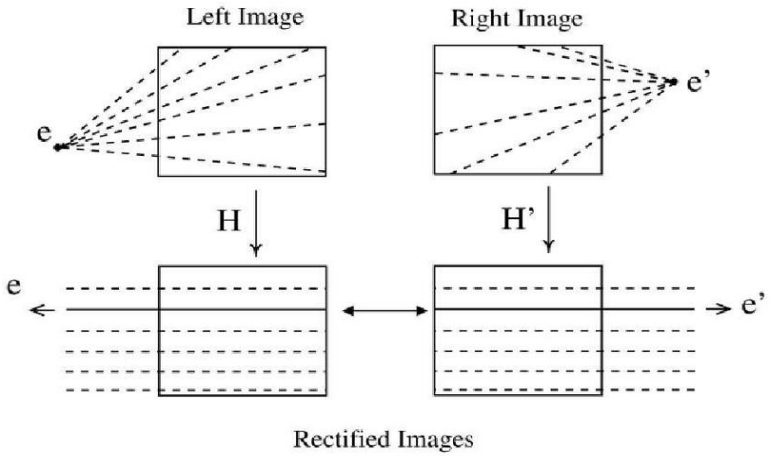


Fig 1.5 Image Rectification Process

1.4.2 NEED FOR IMAGE UNDISTORTION

Undistortion is the process of removing lens distortion from images [2]. Lens distortion is an optical aberration [1]. If the camera is positioned perpendicular to the light sheet and uses a high quality non-distortion lens, there is no need for undistortion [1]. However, if a lower quality lens is used or the camera cannot be positioned perpendicular to the light sheet, a picture of a calibration target can be taken and positioned at the location of the light sheet. Fig 1.6 shows different types of distortions like pincushion distortion and barrel distortion that occurs due to the spherical shape of the lens

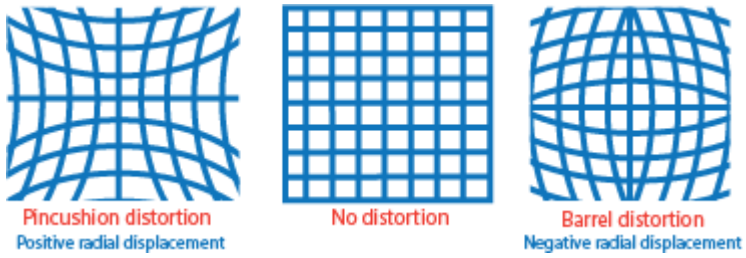


Fig 1.6 Image Distortion due to radial displacement

1.4.3 NEED FOR CAMERA CALIBRATION

Camera calibration, also known as camera resectioning, determines the relationship between a 3D point in the real world and its corresponding 2D projection in the image. This process is used for 3D reconstruction and stereo vision, which is the process of finding depth based on two images [2]. The calibration process determines the distortion factor in the camera imaging process and the quality of the 3D reconstruction [1]. The goal of stereo calibration is to determine the exterior and interior parameters of all cameras of the system to be able to compute 3D object points based on corresponding image points [2].

1.4.4 NEED FOR INTERPOLATION

The need for interpolation in the proposed algorithm arises from the fact that the rectification and undistortion process does not neatly map the output image back to the input image on pixel boundaries [2]. As a result, the final interpolation step is essential to determine the correct address of the Block RAM (BRAM) to read out. This is crucial for accurately compensating for lens distortion and ensuring that the rectified and undistorted images are represented accurately. The interpolation process involves splitting the mapping to the surrounding pixels using bilinear interpolation, which allows for the accurate representation of points in the rectified and undistorted images [2]. Ultimately, interpolation plays a critical role in producing high-quality, undistorted, and rectified images, ensuring the accuracy and reliability of the algorithm's output.

1.5 NEED FOR THE PROJECT

Several important aspects that affect the precision and dependability of visual data processing in computer vision necessitate lens undistortion and image rectification [1]. Here are a few main justifications for carrying out these procedures:

1.5.1 CAMERA CALIBRATION

Camera calibration involves both intrinsic and extrinsic parameters [2]. Two essential components of camera calibration are picture rectification and lens undistortion [1]. The conversion of pixel coordinates into real-world coordinates is made possible by calibrating the camera's intrinsic and extrinsic properties [2]. Accurate measurements and practical uses, such robotics or augmented reality, depend on this calibration [1].

1.5.2 OBJECT RECOGNITION

Consistent Representation of Objects in Object Recognition, affected things may be seen differently due to distortions, making identification more difficult. Redirecting and removing the distortions in the images helps computer vision systems to recognize and classify objects more reliably by promoting consistent object representation [1].

1.5.3 STEREO VISION

Accurate Depth Estimation, Accurate rectification is essential in applications involving the extraction of depth information from differences between corresponding spots in stereo pictures [2]. By guaranteeing that matching points align along epipolar lines, it increases the accuracy of depth maps and makes the computation of disparities simpler.

1.5.4 ELIMINATING COLOR ABERRATIONS

Correction for Chromatic Aberration, Images can be made more visually sharp by removing chromatic aberrations, which can cause color fringing. Maintaining accurate color information is crucial,

particularly in applications that depend on color-based analysis, like object tracking and image segmentation [1].

1.5.5 ENHANCING OVERALL IMAGE QUALITY

Improving the Overall Quality of the Image, Correcting vignetting, or the darkening of image corners, is another aspect of rectification [1]. By ensuring consistent illumination throughout the image, this adjustment improves overall image quality and helps computer vision algorithms that depend on steady intensity values [2]. Enhancing image quality through image rectification and undistortion is imperative for various computer vision applications [2]. The rectification process corrects geometric distortions, such as barrel or pincushion distortions, ensuring that object shapes and spatial relationships are faithfully represented [3]. This contributes to accurate feature extraction, enabling precise matching of points in tasks like stereo vision and 3D reconstruction [1]. Simultaneously, undistortion eliminates lens-induced aberrations like chromatic aberration and vignetting, resulting in images with improved clarity and color accuracy.

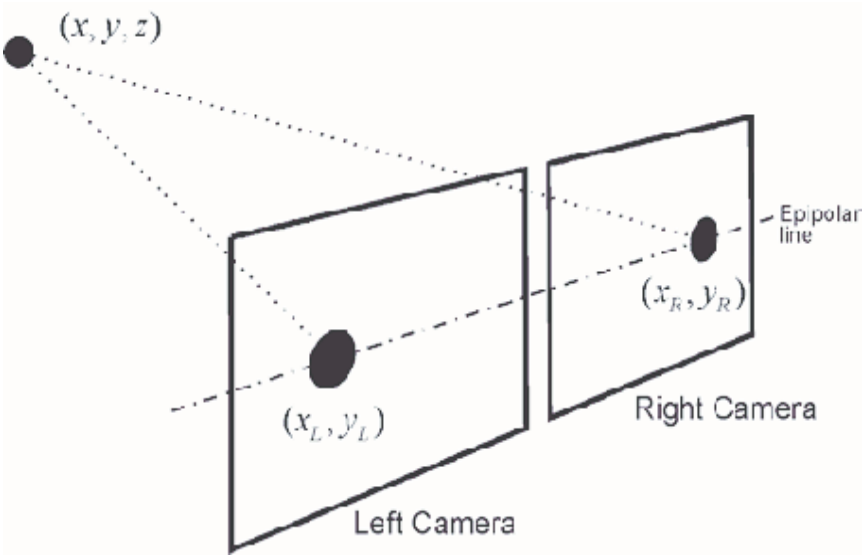


Fig 1.7 Dual Camera for Stereo Vision

1.6 OBJECTIVE OF THE PROJECT

The objective of the project is to develop Area and Power efficient image rectification algorithm for Stereo Imaging in Computer Vision [1]. Thus the proposed algorithm leads to 61.764% decrease in LUT, 5.35% decrease in Power and 49.4% decrease in Average Mean Error. The left and right images are taken and combined with necessary amount of gray channel and then undistortion and interpolation is performed. These functionalities are modelled as separate blocks in Simulink for Software Analysis. Performance Metrices are calculated by collecting ample amount of dataset both for error analysis and for calculation of distortion deviations [2]. The algorithm is also converted to Verilog code inorder to perform Area, Power and synthesis analysis [2].

1.7 ORGANIZATION OF THE REPORT

The report is structured as follows. Chapter 2 presents the literature review that outlines the various approaches used in image rectification and undistortion algorithm. The suggested algorithm for the same is explained in Chapter 3. The system requirements for the hardware implementation of the algorithm as well as the implementation details are discussed in Chapter 4. The simulation results and the results from hardware implementation of the algorithm are covered in Chapter 5. The outcome of the algorithm and the applications of the work are reported in Chapter 6. Finally, the references that we have referred to are all given in Chapter 7.

CHAPTER 2

LITERATURE SURVEY

This paper presents a revolutionary embedded stereo vision system based on the heterogeneous SoC FPGA [1]. There are two finished implementations that are described and shown [1]. Both systems use a specialized hardware module that is specifically made to calculate disparity maps, which allows for extremely fast performance of auxiliary calculations like image rectification and disparity map refinement. Configuration and communication are managed by the on-chip general-purpose processor's software routine. The obtained findings demonstrate that the suggested system achieves improved performances, competitive accuracy, lesser complexity, and higher flexibility when compared to numerous existing hardware solutions. For many contemporary computer vision applications, including gesture detection, navigation by robots, and driverless cars, stereo vision is essential. In essence, stereo vision is the capacity to infer depth information about an observed scene from two-dimensional (2D) images that were taken from distinct points of view by two cameras. Things that are visible to both cameras are projected at various locations into the pictures that are taken [1].

In the realm of 3D computer vision and robot vision, stereo image processing is one of the most demanding tasks that calls for high-performance computing skills within embedded programs [2]. Hardware acceleration is a result of real-time limitations for autonomous vehicles, such as humanoid robots. Techniques for high-resolution stereo imaging in systems that resemble human vision, where FPGA devices are frequently used to manage extremely high rates of sensor data. This paper describes the creation of a real-time smart stereo camera system that looks like the entire stereo processing chain contained within a single FPGA module. We present "Sparse Retina Census Correlation, " a novel memory-optimized stereo processing algorithm that combines two well-known window-based stereo matching techniques [2].

This paper describes a real-time multicamera cylindrical panorama video system using a field-programmable gate array (FPGA) [3]. A novel method of simultaneous picture alteration has been studied in this system. Using this technique, cylindrical projection, barrel correction, and perspective transformation were carried out simultaneously for a multicamera system, improving performance overall. A high-speed real-time panoramic video system with minimal latency and good performance has been developed using an affordable parallel FPGA architecture. This video system can produce 5 × 1280(H) × 720(V) panoramic video at 15 frames per second or 5 × 1280(H) × 360(V) at 30 frames per second on a Xilinx Spartan6-150T FPGA in real time. Furthermore, a hardware display structure capable of outputting the panoramic video in slide mode has been built in order to display the multimedia [3].

This paper proposes the construction of a real-time FPGA-based rectification algorithm in conjunction with a stereo camera [4]. Rectification, which includes radial distortion correction, is a prerequisite for stereo matching in order to align the left and right pictures. In our experiment, we used a stereo camera and an FPGA rectification module to test in a video environment. As a result, Zynq7020

FPGA processes HD picture rectification at 45 frames per second. Prior to stereo matching, rectification is a crucial step in aligning the epipolar lines of the two side pictures. Furthermore, autonomous fields frequently use fish-eye lenses in stereo vision. As a result, the rectification algorithm corrects lens distortion by combining the alignment and calibration processes. Zicari suggested an FPGA-based rectification architecture for stereo vision, whereas several papers suggested rectification and calibration methods. A comprehensive system design is required to efficiently construct hardware in order to evaluate stereo vision in video scenarios [4].

One Area of computer vision research is stereo image matching. Technological advancements in stereo image matching move the field from Area-based matching techniques to feature-based matching techniques [5]. We describe a Harris corner identification technique for matching features in stereo images in this study. With the use of a threshold value, this intensity-based feature matching method manages the strong and weak corners. Typically, software is used to simulate image processing algorithms; however, in this case, Xilinx System Generator is used to perform hardware co-simulation using a model-based design. Moreover, the Xilinx Virtex-5 FPGA is used to synthesize the architecture. This study presents simulation results to validate the suggested system's performance [5].

The FPGA acceleration community has recently paid close attention to stereo-matching methods [6]. The solutions that are offered range from straightforward, very resource-efficient systems for tiny embedded devices with low matching quality to complex algorithms implemented on large FPGAs that require multiple processing steps. To attain a high rate of throughput, The majority of implementations place a lot of emphasis on data reuse and pipelining between various computational stages. Although this method achieves high efficiency, it restricts the computing patterns that may be used, and because of the high implementation integration, it is challenging to modify the algorithm. This study, offers an implementation of stereo-matching that begins with the CPU offloading individual kernels to the FPGA. Data is kept off-chip in the FPGA accelerator card's on-board memory in between consecutive compute stages. For the first time, without requiring computational changes, and for up to full HD image dimensions, this allows us to speed up the AD-census algorithm using cross-based aggregation and scanline optimization. We present various trade-offs associated with this method in comparison to a tighter integration of many kernel loops into a single design by analyzing performance and bandwidth needs [6].

This work describes the hardware system plan for DSP-based multi-channel image processing [7]. FPGA serves as a coprocessor to gather the original picture data from CMOS, while DSP acts as the primary control of both FPGA and CMOS, handling the entire scheduling system. In order to meet the requirements for both stereo and panoramic vision, this study integrates the fisheye lens. The SRAM cache controller based on the ping-pong technique is designed using FPGA. The DSP's EDMA model completes the data exchange between the DSP and FPGA. Cross clock domain is a problem that is solved by applying asynchronous FIFO. These days, popular Areas of visual processing include stereo vision and panoramic vision. With a panoramic vision system, we can obtain picture information over a large field of view, and with a stereo vision system, we can determine the object's depth information. However, each of the two visual systems has drawbacks of its own. For example, a stereo visual system's field of view is limited, while a panoramic visual system's picture processing will lose one-dimensional depth information.

However, there are situations when both the image's depth information and a broader field of view are required simultaneously, such as in military tracking and field detection [7].

Lens undistortion and image rectification is a commonly used pre-processing, e. g. for active or passive stereo vision to reduce the complexity of the search for matching points [8]. The undistortion and rectification is implemented in a field programmable gate array (FPGA). The algorithm is performed pixel by pixel. The challenges of the implementation are the synchronisation of the data streams and the limited memory bandwidth. Due to the memory constraints, the algorithm utilises a pre-computed lossy compression of the rectification maps by a ratio of eight. The compressed maps occupy less space by ignoring the pixel indexes, sub-sampling both maps, and reducing repeated information in a row by forming differences to adjacent pixels [8].

CHAPTER 3

IMAGE RECTIFICATION ALGORITHM

Image rectification is a crucial pre-processing step in computer vision and stereo vision systems that plays a pivotal role in ensuring accurate and meaningful analysis of images [2]. The primary objective of image rectification is to remove perspective distortions and disparities caused by variations in camera positions and orientations [2]. Through a series of geometric transformations, rectification transforms images into a common perspective, where corresponding points align along the same epipolar lines [2]. This process is particularly valuable in stereo vision applications, as it simplifies subsequent depth estimation algorithms and facilitates easier comparison of image features [2]. By rectifying images, the system minimizes geometric distortions, thus improving the reliability and precision of subsequent computer vision tasks such as object recognition, tracking, and three-dimensional reconstruction [1]. Overall, image rectification is a critical technique that enhances the consistency and accuracy of visual data for more effective and reliable image analysis

3.1 IMAGE RECTIFICATION AND UNDISTORTION ALGORITHM

Fig 3.1 shows the left and the right image taken separately and then combined along with required amount of gray channel. The combined image is then passed on to rectification, Undistortion and interpolation block to get the image rectified [3]. The algorithm's key components and characteristics can be summarized as follows: The algorithm takes as input the homography matrix and distortion coefficients, which are determined during the calibration process [2]. These parameters are essential for rectifying and undistorting images captured by the left and right cameras [2]. The processing for both left and right cameras follows the same pipeline, differing only in the input parameters [2]. This streamlined approach ensures consistency and simplifies hardware implementation [1]. Additionally, the synchronization of cameras in hardware ensures uniformity in image processing. The algorithm incorporates a final interpolation step to handle the discrepancies between pixel mappings from the output image back to the input image. This step is necessary for achieving smooth and accurate rectification and undistortion results.

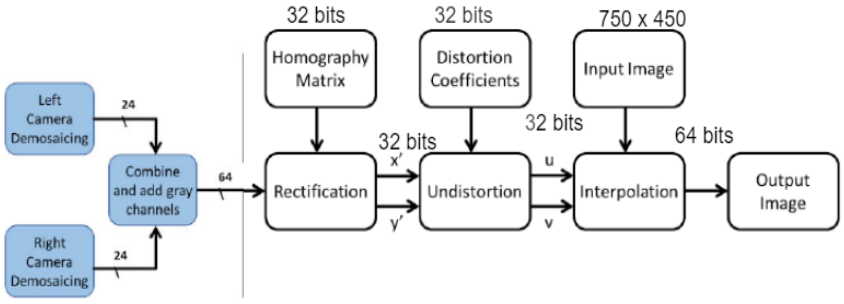


Fig 3.1 Image Rectification and Undistortion Algorithm

The algorithm used in the project employs a mathematical approach to determine the rectification and undistortion inverse mapping, as opposed to the lookup method [3]. The entire algorithm exists within the

Simulink development environment, and the Vision HDL toolbox is used to aid in the simulation by allowing the conversion of images back and forth between logical arrays and pixel streams [2]. Fig3.2 illustrates how a human binocular vision works [2].

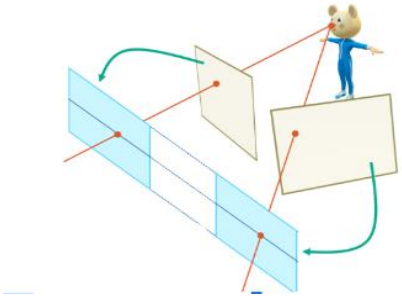


Fig 3.2 Human Binocular Vision

3.2 CAMERA MODEL

In the project work, a robust stereo depth perception system was implemented using a dual-camera model. The stereo camera pair, comprising two individual cameras with their focal lengths and optical centers, provided a Powerful platform for capturing and analyzing depth information in real-time. The coordination of these cameras allowed for the creation of a three-dimensional representation of the environment, enabling precise depth estimation for objects within the field of view. Utilizing intrinsic and extrinsic matrices derived through MATLAB's stereo camera calibration tool, the system demonstrated effective calibration for accurate depth mapping. The dual-camera configuration not only enhanced the accuracy of depth perception but also facilitated the implementation of closed-loop techniques and virtual cameras for thorough debugging and algorithm verification [1]. This approach not only ensures a comprehensive understanding of the spatial relationships between objects but also offers flexibility in adapting to various scenarios, contributing to the advancement of stereo vision applications [2]. Fig 3.3 shows the mathematical matrix calculations to be performed both for Intrinsic and Extrinsic parameters calculation [1]. Fig 3. 4 illustrates how extrinsic parameters differs from intrinsic parameters in a real world scenario.

$$\begin{bmatrix} f_x & skew & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = IntrinsicMatrix \quad \text{..... Eqn 3.1}$$

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \end{bmatrix} = ExtrinsicMatrix \quad \text{..... Eqn 3.2}$$

Fig 3.3 Camera Parameters

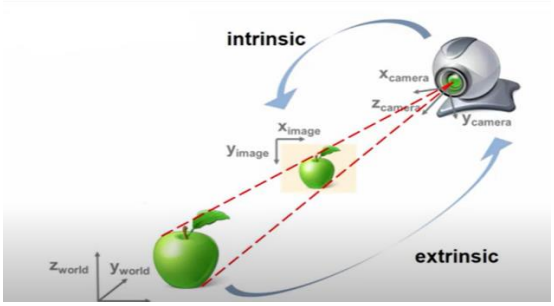


Fig 3.4 Intrinsic and Extrinsic Camera Parameters

3.3 MODELLING OF EXISTING IMAGE RECTIFICATION ALGORITHM

Fig 3.5 shows that the left and the right image each has 8 bits allocated for red , green and blue colour components where demosaicing is to reconstruct a full-color image from the incomplete color information captured by a single sensor with a color filter array (CFA). It is carried out separately for both the blocks, on adding both we get a total of 48 bit channel. In the combined image block, desired 16-bit gray channel is also added and finally 64 bit is been obtained [3]. Thus the image is converted from frame to pixel stream for the subsequent process to take place efficiently. The combined image is then passed onto the image rectification block and the (x', y') pixel coordinates are passed onto the undistortion block. The (u, v) coordinates obtained from undistortion block is then passed onto interpolation block and finally the output is obtained [3].

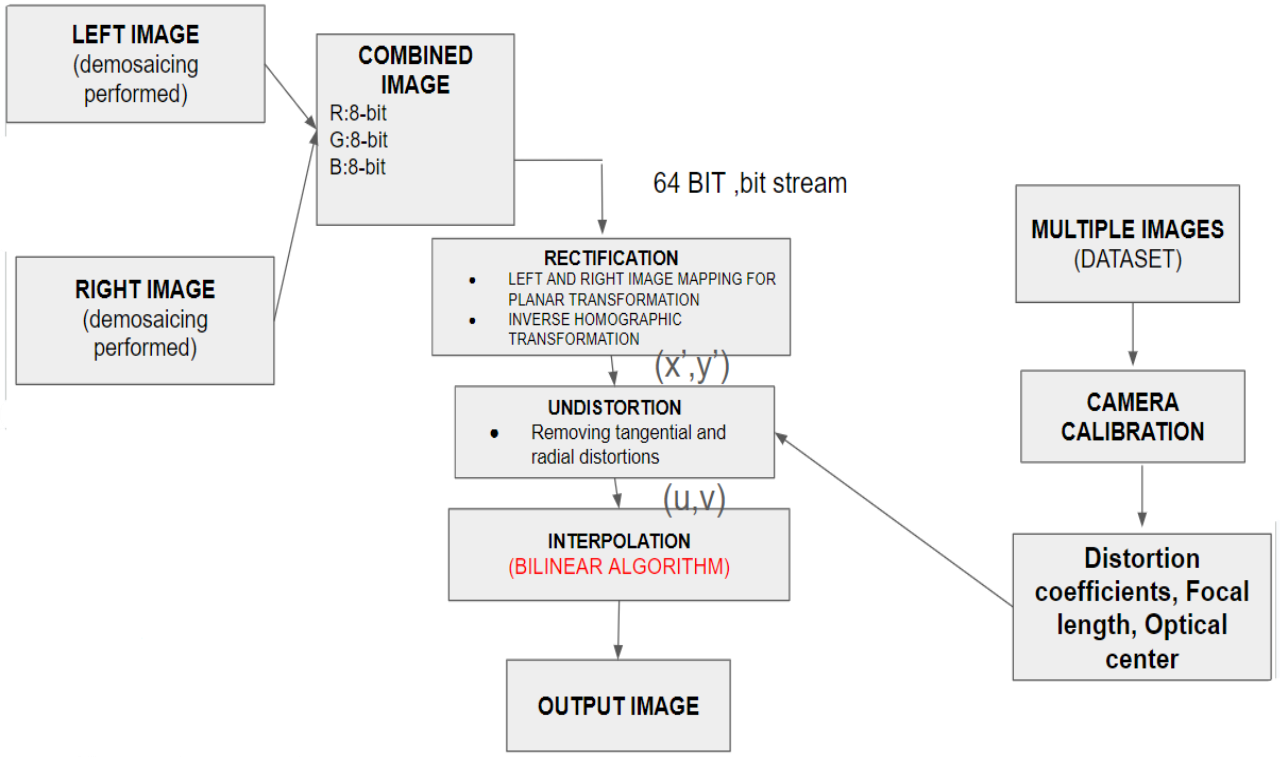


Fig 3.5 Modelling of Existing Image Rectification Algorithm

The suggested system utilizes established optics theories and mathematical principles to implement a computer vision system. Although camera calibration commonly encompasses rectification and undistortion processes, in this context, we specifically use the term to denote the capture of images for calculating calibration parameters, encompassing intrinsic and extrinsic camera properties [2]. Additionally, we employ a linear camera model, simplifying calculations through specific assumptions regarding the translation of points from 3D space to a 2D image plane. Also various physical aspects of the camera system significantly impact the captured scene and necessitate measurement or calculation [1]. These properties encompass focal length, optical center, and distortion (radial and tangential). While a pinhole model may not fully encapsulate all camera system parameters, it is generally considered sufficiently accurate , as the mathematical adjustments are minimal when distortion values are incorporated into existing translation matrices [2]. Upon determining calibration parameters, the proposed system facilitates image rectification and undistortion [1]. This process involves creating new images that compensate for relative offsets between two image sensors and counteract radial and tangential effects [2].

3.3.1 HOMOGRAPHIC TRANSFORMATION

Homographic transformation, also known as a projective transformation or homography, is a mathematical technique employed in computer vision and image processing to map points from one image plane to another. This transformation is particularly useful when dealing with images taken from different perspectives or viewpoints [2]. A homography represents a 3x3 matrix that defines the mapping between two planes in a projective space. It can rectify distortions caused by changes in camera angles, perspectives, or even non-planar surfaces [2]. By applying a homographic transformation, it becomes possible to align and overlay images seamlessly, ensuring accurate spatial correspondence between different views or frames [2]. This technique is fundamental for tasks that involve combining or comparing images captured under different geometric conditions, providing a Powerful tool for enhancing the robustness and accuracy of computer vision applications [2]. Fig 3.6 and Fig 3.7 shows how the homographic transformation is done using inverse homographic matrix operation and thus the rectified coordinates are obtained [3]. Fig 3.7 shows the generalized classification of image transformation where only homographic transformation is alone used in our project.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x_{rec} \\ y_{rec} \\ 1 \end{bmatrix} * H_{3\times3}^{-1} \quad \text{..... Eqn 3.3}$$

Fig 3.6 Homographic Matrix

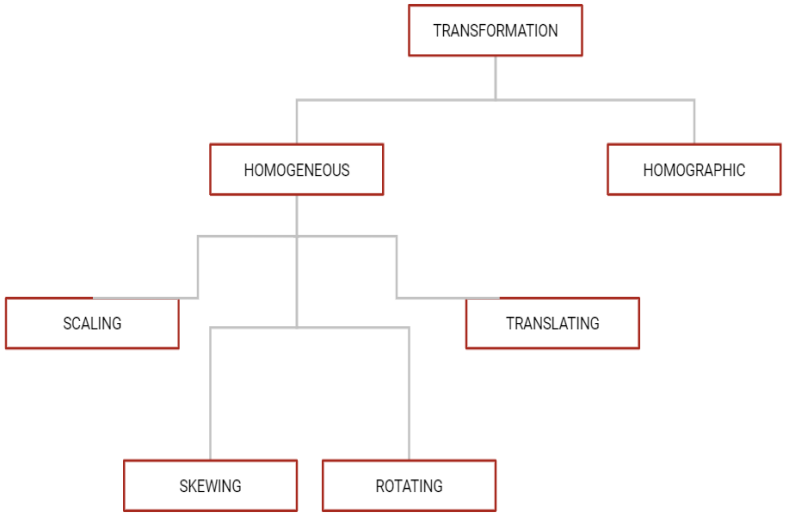


Fig 3.7 Classification of Transformations

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & b_0 \\ A_{10} & A_{11} & b_1 \\ h_0 & h_1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

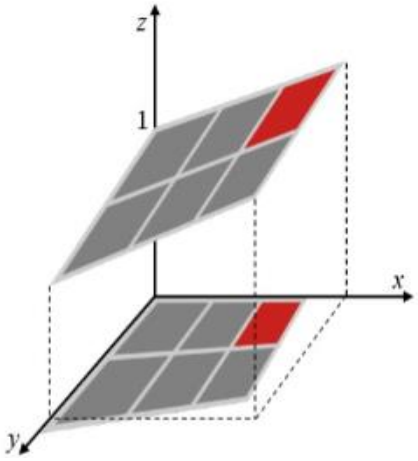


Fig 3.8 Homographic Transformation

3.3.2 UNDISTORTION

Undistortion holds significant importance in computer vision due to its crucial role in rectifying optical distortions inherent in images captured by cameras [2]. The need for undistortion arises from the fact that real-world camera systems introduce various distortions, such as radial and tangential distortions, during the image acquisition process [2]. These distortions can impact the accuracy of subsequent computer vision tasks, including object recognition, feature matching, and depth estimation [1]. Undistortion corrects these distortions, ensuring that the captured images accurately represent the scene geometry and object shapes [2]. In applications like stereo vision and 3D reconstruction, undistortion is pivotal for precise depth estimation and spatial understanding. Additionally, undistorted images facilitate the development and deployment of machine learning models by providing cleaner and more accurate input data. Overall, undistortion plays a fundamental role in improving the reliability and effectiveness of computer vision algorithms, contributing to advancements in various fields, from robotics to augmented reality. Fig 3.9 shows the inverse mapping that takes place in undistortion block.

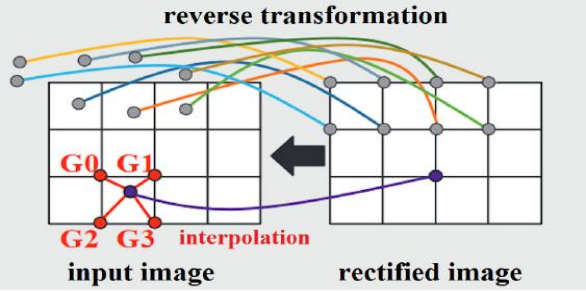


Fig 3.9 Reverse Transformation

Pre-processing for the undistortion process entails normalizing the coordinates (x′, y′) by translating them to the optical center (cx , cy) and dividing them by the focal lengths of the camera (fx , fy)

$$(x_n, y_n) = \left(\frac{x' - c_x}{f_x}, \frac{y' - c_y}{f_y}\right) \quad \text{..... Eqn 3.4}$$

$$v_{radial} = y_n(1 + k1r^2 + k2r^4)$$
$$v_{tangential} = 2p2x_n y_n + p1(r^2 + 2y_n^2) \quad \text{..... Eqn 3.5 (i \& ii)}$$

$$u = u_{radial} + u_{tangential}$$
$$v = v_{radial} + v_{tangential} \quad \text{..... Eqn 3.6 (i \& ii)}$$

Fig 3.10 Mathematical Approach for Undistortion

3.3.3 INTERPOLATION

Interpolation during image preprocessing is a critical step in computer vision that involves estimating pixel values at non-integer coordinates based on the known pixel values in the surrounding neighborhood [3]. This technique is employed to address challenges such as resizing or transforming images, where the target size or orientation may not align precisely with the original pixel grid [3]. Common interpolation methods include bilinear and bicubic interpolation, which consider nearby pixel values to calculate the intensity of pixels at new locations [2]. Interpolation ensures smooth transitions and reduces aliasing artifacts when resizing or transforming images, contributing to the overall quality and accuracy of subsequent computer vision tasks [2]. Fig 3.10 shows how to mathematically derive the desired pixel values by using the surrounding pixel values [2].

$$
\begin{aligned}
compPixel = {} & G_0(1-deltaU)(1-deltaV) \\
& +G_1(1-deltaU)(1-deltaV) \\
& +G_2(1-deltaU)(1-deltaV) \\
& +G_3(1-deltaU)(1-deltaV)
\end{aligned}
\qquad \ldots\ldots\ldots\ldots \text{ Eqn 3.7}
$$

Fig 3.11 Mathematical Approach for Interpolation

3.4 MODELLING OF PROPOSED METHODOLOGY

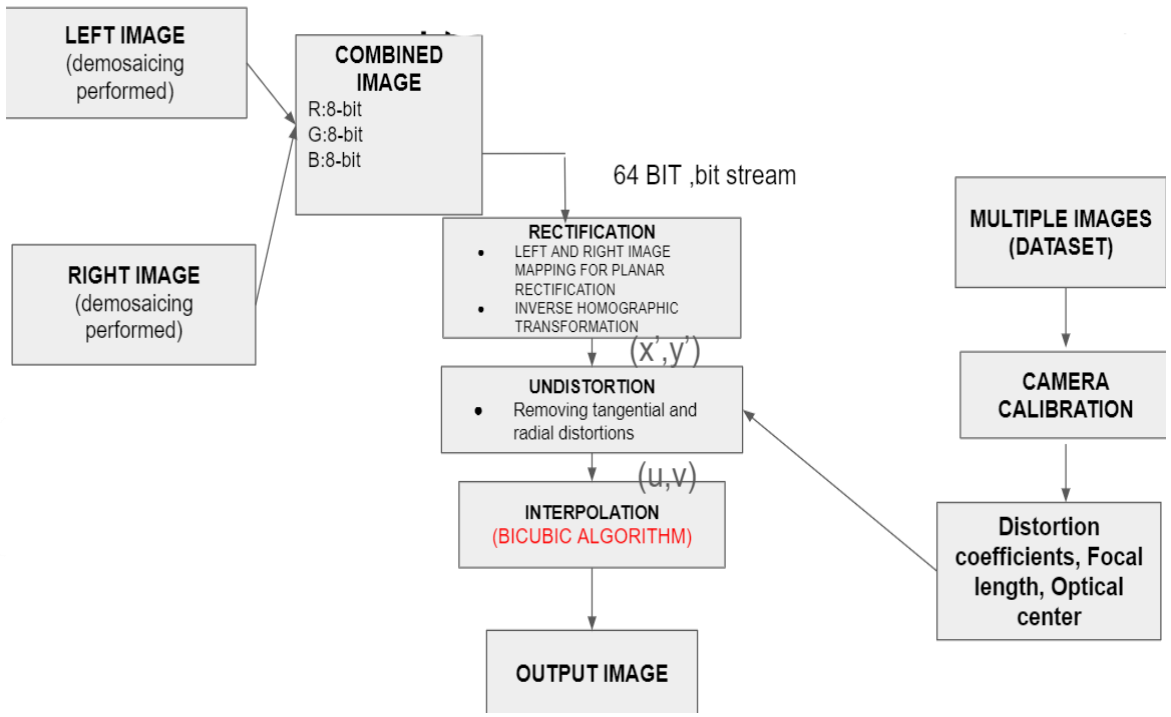


Fig 3.12 Modelling of Proposed Methodology

Fig 3.12 shows the modelling of proposed methodology where the advanced and optimized technique of interpolation called Bicubic interpolation is used rather than the Bilinear interpolation. Bicubic interpolation considers a larger Area of 16 surrounding pixels to calculate the value of the target pixel. This allows for smoother transitions and better preservation of image details, especially when scaling images down [1]. Bicubic interpolation can potentially create sharper images with better edge definition compared to bilinear interpolation, which can sometimes lead to a blurry appearance. Bicubic interpolation is particularly beneficial for images with sharp details like text, lines, or intricate patterns [2]. Bicubic interpolation creates smoother transitions between colors and gradients in the image compared to bilinear interpolation, which can sometimes introduce unwanted blackness [2]. It excels at maintaining these details during resizing. Also the datasets of the same image taken in multiple angles are fed into the matlab toolbox for efficient intrinsic and extrinsic parameters with which the extent of distortions can be easily calculated [3]. These parameters include Distortion coefficients, Focal length, Optical center, etc.

3.5 OPTIMIZATION OF INTERPOLATION TECHNIQUE

A more optimized technique is proposed to be used in the project which is bicubic interpolation instead of bilinear interpolation.

3.5.1 BILINEAR INTERPOLATION

Bilinear interpolation uses a first-degree polynomial, which means it interpolates between four or eight neighboring points using linear functions along each dimension [1]. Bilinear interpolation is computationally less intensive compared to higher-degree methods like cubic interpolation [1]. It essentially creates a simple "tent" function by linearly interpolating between these four points to determine the target pixel's value. This can sometimes lead to a blocky or stepped appearance, especially when scaling images down significantly.

3.5.2 BICUBIC INTERPOLATION

Bicubic interpolation uses a third-degree polynomial for interpolation, resulting in a smoother curve than bilinear interpolation [1]. Mathematical calculations are less compared to bilinear interpolation [1]. It requires solving a system of equations to determine the coefficients of the cubic polynomial. This curve fitting allows for a more gradual transition of color values between pixels in the resized image. The smoother curve also leads to smoother transitions between colors and gradients [2]. This creates a more natural-looking image, especially in Areas with gradual changes in color or intensity.

3.6 ADVANTAGES OF PROPOSED ALGORITHM

Bicubic interpolation is generally considered more accurate than bilinear interpolation, especially for downscaling images [2]. By considering a larger Area of pixels and using a smoother curve fitting technique, bicubic interpolation better approximates the original image's details and color values in the resized version [1]. Typically leads to a lower AME compared to bilinear interpolation [1]. AME measures the average difference between corresponding pixel values in the original and resized images [2].

A lower AME indicates a smaller average difference and therefore higher accuracy. May have a slightly lower PSNR compared to bilinear interpolation, depending on the specific image content. PSNR measures the ratio between the maximum possible signal (peak value) and the background noise in an image. A higher PSNR indicates a better signal (image) to noise ratio. The more accurate estimation of pixel values in bicubic interpolation leads to a smaller average difference between the original and resized image's corresponding pixels [2]. This translates to a lower AME, signifying a higher degree of accuracy in representing the original image's information [1]. While bicubic interpolation might introduce some overshoot artifacts (halos around edges), these might not always register as significant noise in the PSNR calculation [1].

CHAPTER 4

IMPLEMENTATION

System requirements serve as the foundational framework for the project, serving a critical role in its completion [1]. These requirements define the functionality, performance, and constraints that a system must meet to satisfy the objectives of the project. This project includes both Software and Hardware implementations [2]. The major chunks of software implementation revolves around Matlab and Simulink while the hardware implementation is done using Arduino board [3].

4.1 SOFTWARE DESIGN FLOW

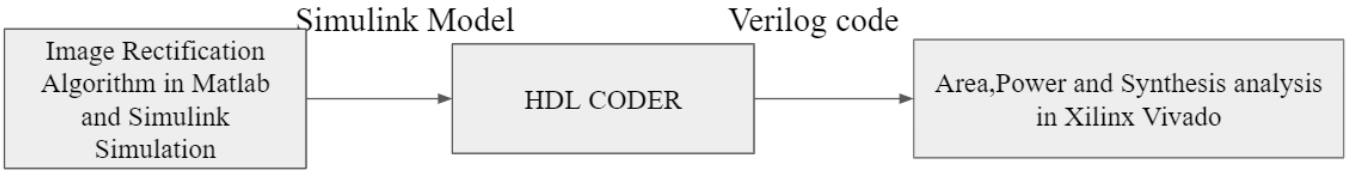


Fig 4.1 Design Flow Diagram

Fig 4.1 shows the software flow of the project where the modelled algorithm for proposed methodology is been implemented in matlab using matlab code and in Simulink as Simulink blocks [2]. The Simulink model thus generated is then converted to Verilog Code using the HDL Coder and the modules that are generated during this process is been transferred to Xilinx Vivado software where various analysis takes place. Once the Verilog modules are generated, they are imported into Xilinx Vivado software, a comprehensive design environment for FPGA (Field-Programmable Gate Array) development. Within Vivado, various analyses are conducted to optimize the design for performance, Power consumption, and resource utilization [1]. These analyses include timing analysis, Power estimation, and resource utilization reports, among others [2]. Through iterative refinement and optimization, the design is honed to meet the project's requirements and constraints, ensuring its viability for deployment on FPGA hardware. This comprehensive software flow, encompassing MATLAB, Simulink, HDL Coder, and Xilinx Vivado, facilitates a streamlined development process for implementing and refining the proposed methodology's algorithm on FPGA platforms [2].

4.2 MATLAB AND SIMULINK FEATURES

Matlab and Simulink play pivotal roles in advancing projects within the domain of image processing, offering a comprehensive and integrated environment for research and development. Matlab's Powerful scripting language facilitates efficient and flexible image manipulation, analysis, and algorithm implementation [1]. Simulink, on the other hand, provides a graphical interface for modeling complex systems, making it an ideal tool for designing and simulating image processing algorithms [2]. The seamless integration of these two platforms allows for the rapid prototyping and testing of diverse image processing

techniques, ensuring a streamlined development process [2]. The extensive range of built-in functions and toolboxes in Matlab provides researchers and developers with a rich set of resources, enabling them to explore innovative solutions for image enhancement, segmentation, and feature extraction [1]. The combination of Matlab and Simulink not only expedites the development cycle but also enhances the overall quality and reliability of image processing projects, making them indispensable tools in the pursuit of cutting-edge advancements in this field [3]. All simulation and HDL code generation was performed with the MathWorks tools namely MATLAB and Simulink. Additional toolboxes were required such as the Fixed-point designer as well as the Vision HDL Toolbox and HDL Coder. Fig 4.2 shows the logo of matlab, Simulink and how the system generator can be configured to make use of system generator toolbox.

Fig 4.2 MATLAB and XILINX System Generator Logo

4.3 SYSTEM GENERATOR

System Generator is a Powerful tool utilized in digital signal processing (DSP) and FPGA (Field-Programmable Gate Array) design [1]. It serves as a high-level design environment for implementing DSP algorithms on FPGA platforms [2]. With System Generator, designers can graphically model and simulate complex systems using Simulink, a MATLAB-based tool, which enables intuitive block diagram-based design [1]. This platform facilitates rapid prototyping and verification of algorithms, significantly reducing development time. By seamlessly integrating with Xilinx FPGA development boards and design flows, System Generator emPowers engineers to efficiently implement and deploy real-time DSP applications, such as image processing, wireless communication, and control systems, onto FPGA hardware, thus unlocking performance gains and flexibility in embedded system design [1].

4.4 XILINX VIVADO

The Xilinx Vivado 2018.2 tool is used for HDL development for the project. The static parts of the design include the camera pre-processing stages such a demosaicing as well as various functional blocks [2]. The Vivado HDL block diagram for this project is designed in such a way that the dynamic portion of the design which auto-generated and one can think of the entire project as containing peripheral wrapper code as well as main algorithmic core code. It creates hardware descriptions using industry-standard Hardware Description Languages (HDLs) like Verilog and VHDL. It translates your HDL code into an optimized netlist suitable for FPGA implementation [1]. It automatically places and routes logic cells and connections within the FPGA fabric to achieve timing and resource utilization goals [2]. It performs

comprehensive simulations to verify the functionality of your design before hardware deployment. It also generates bitstream files containing the configuration data for programming the FPGA. It can debug and test the design on physical hardware platforms using JTAG and other debugging interfaces [2]. It can also integrate your FPGA design with other hardware and software components for complete system creation [1]. This project requires Area, Power and Synthesis analysis to be performed using Xilinx Vivado tool.

4.5 DATASET USED FOR CAMERA CALIBRATION

Collection of 20 (each) left and right images were downloaded from kaggle website(https://www. kaggle. com/datasets). These images are fed to stereo camera calibration matlab toolbox for intrinsic and extrinsic camera parameter calibration [1]. The level of distortion can be calculated in the undistortion block using these parameters [2]. Kaggle website is been navigated to and datasets relating to stereoimaging is searched for, while doing so it has been found to contain some 20 images each for left and right camera captures of chess board images [2]. These are then downloaded as zip file and are then imported to matlab file directory where it is then processed to calculate internal and external camera parameters [2]. Fig 4.3 shows the Kaggle website page from where the chess board images are downloaded [3].

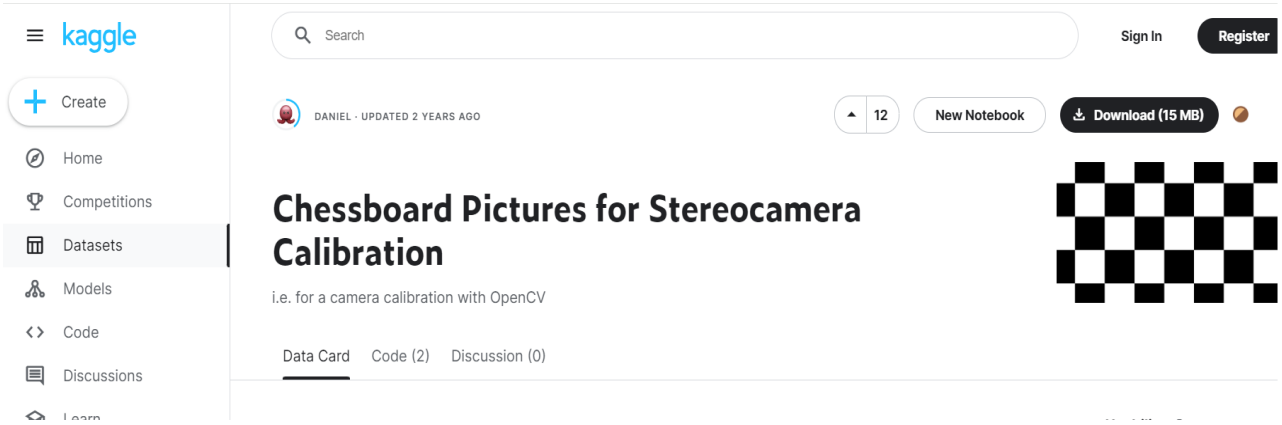


Fig 4.3 Dataset Collection using Kaggle

4.6 DATASET USED FOR ERROR CALIBRATION

Similar to above mentioned dataset collection, some different sets of datasets are also been collected for verification at the output side where these images is tested for the difference between bilinear and bicubic algorithm and the average mean error and performance parameters are estimatd and the efficiency and accuracy has been calculated [1]. The average mean error, often referred to simply as the mean error, is a statistical measure used to quantify the average deviation between observed values and predicted values in a dataset. It is calculated by taking the average of the absolute differences between each observed value and its corresponding predicted value. The Peak Signal-to-Noise Ratio (PSNR) is a metric widely used in image processing and video compression to assess the quality of a reconstructed or compressed image or video compared to the original. It measures the ratio between the maximum possible Power of a signal and the Power of corrupting noise that affects the fidelity of its representation [1].

Table 1 List of Datasets used for Error Analysis

IMAGE NAME	PICTURES
IMAGE 1(MAP)	
IMAGE 2(AUDITORIUM)	
IMAGE 3(BICYCLE)	
IMAGE 4(COMPUTER)	
IMAGE 5(FURNITURE)	
IMAGE 6(DRUMS)	
IMAGE 7(HOOPS)	
IMAGE 8(ROOM)	
IMAGE 9(TABLE LAMP)	
IMAGE 10(VASE)	

IMAGE 11(STAIRS)	
IMAGE 12(MUSICAL INSTRUMENTS)	
IMAGE 13(LIVING ROOM)	
IMAGE 14(CHAIRS)	
IMAGE 15(DUSTBIN)	

4.7 IMPLEMENTATION IN MATLAB

The MATLAB implementation plays a crucial role in simulating, generating, and targeting the stereo rectification and undistortion algorithm, demonstrating the effectiveness of Model Based Design Image Rectification Algorithm. This project is been developed by writing optimized code for each and every functional blocks like Image Rectification, Image Undistortion and Image Interpolation block so that the output obtained is free of any kind of distortions so that it can be utilized for third party applications like object detection or 3d image reconstruction [1]. Under the Image Rectification algorithm ample amount of matrix mathematical calculations are involved for proper homograpohic transformation so that it is in a way that is suitable to be fed into the undistortion block. Under undistortion block Matlab toolbox imports the necessary extrinsic and intrinsic patrameters that are required for the extent of distortion calculation [1]. After image undistortion intensity of image pixels are been improved by interpolation technique where the advanced bicubic technique of image interpolation technique is used [3]. ,Matlab is also used for calculation of performance Metrix such as Average Mean Error.

4.8 IMPLEMENTATION IN SIMULINK

In the Simulink implementation of the project, the focus shifts towards translating the MATLAB-based algorithmic logic into a visual and modular representation [1]. Each component of the stereo rectification and undistortion algorithm is encapsulated within Simulink blocks, facilitating a more intuitive and accessible design environment. The Image Rectification, Image Undistortion, and Image Interpolation functionalities are encapsulated as interconnected blocks, reflecting the flow of data and computations [2]. The Simulink model enables seamless integration with MATLAB, allowing for iterative refinement and testing of the algorithm in a real-time simulation environment. Additionally, Simulink provides tools for visualization and analysis, enabling engineers to gain insights into system behavior and performance. By leveraging Simulink's capabilities, the project ensures not only the accuracy and efficiency of the algorithm but also its scalability and ease of deployment in real-world applications [2].

4.8.1 IMAGE RECTIFICATION IN SIMULINK

To implement a stereo image rectification HDL block in Simulink, first, grasp the chosen rectification algorithm [1]. Next, translate this algorithm into MATLAB or Simulink code, focusing on aligning corresponding points in stereo image pairs to simplify subsequent processing. Then, encapsulate this code into a custom Simulink block, ensuring inputs are stereo image pairs and outputs are rectified stereo image pairs [2]. If HDL compatibility is required, optimize the block for HDL code generation by using fixed-point data types, avoiding unsupported MATLAB functions, and ensuring hardware suitability. Validate the block's functionality by testing with sample stereo image pairs and verifying rectification accuracy. If needed, optimize performance through techniques like pipelining or parallelization [1]. Document the block comprehensively, detailing its functionality, inputs, outputs, and any constraints [2]. Consulting Simulink documentation and relevant resources will provide further guidance on creating custom blocks and working with HDL code generation [1]. Fig 4.4 shows the arrangement and flow of images that was designed in Matlab Simulink based on the modelled algorithm for the proposed methodology.

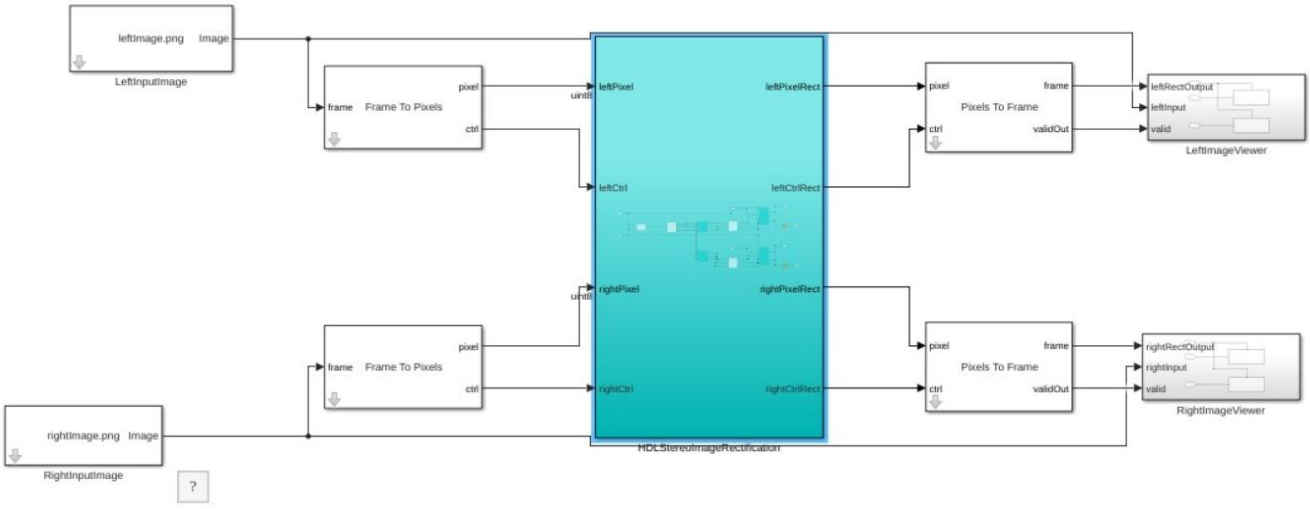


Fig 4.4 Designed Image Rectification Simulink Blocks

4.8.2 UNDISTORTION IN SIMULINK

To implement an Image Undistort HDL block in Simulink, it is needed to understand the undistortion algorithm, typically involving correcting lens distortions like radial and tangential distortions [2]. Once the algorithm is clear, it can be proceeded to implement the algorithm within Simulink. You can utilize MATLAB or Simulink blocks, like MATLAB Function blocks, to encapsulate the undistortion algorithm. Define the block's inputs as distorted images and outputs as undistorted images [2]. Ensure the implementation is compatible with HDL code generation, adhering to HDL-compatible practices such as using fixed-point data types and avoiding unsupported functions [2]. Thoroughly test the block with sample distorted images to validate its functionality and accuracy in correcting distortions [2]. If necessary, optimize the block's performance through techniques like pipelining or parallelization [1]. Document the block's functionality, inputs, outputs, and any constraints for future reference. Fig 4.5 depicts the designed Simulink blocks that contains the undistortion algorithm where input image is taken as input, processed by checking and correction of distortions and distortion free output is obtained [3]. Fig 4.6 shows the inbuilt values for intrinsic and extrinsic parameters that are present in Simulink for distortion removal.

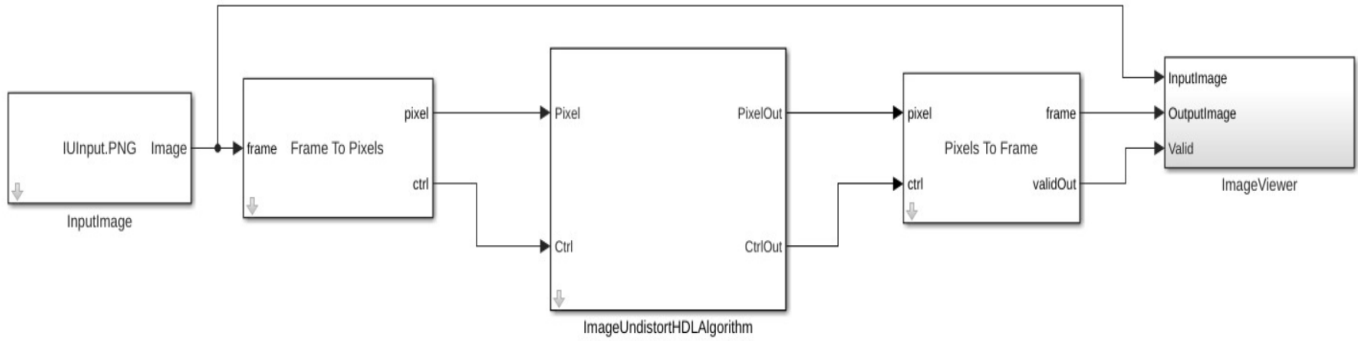


Fig 4.5 Image Undistortion Simulink Blocks

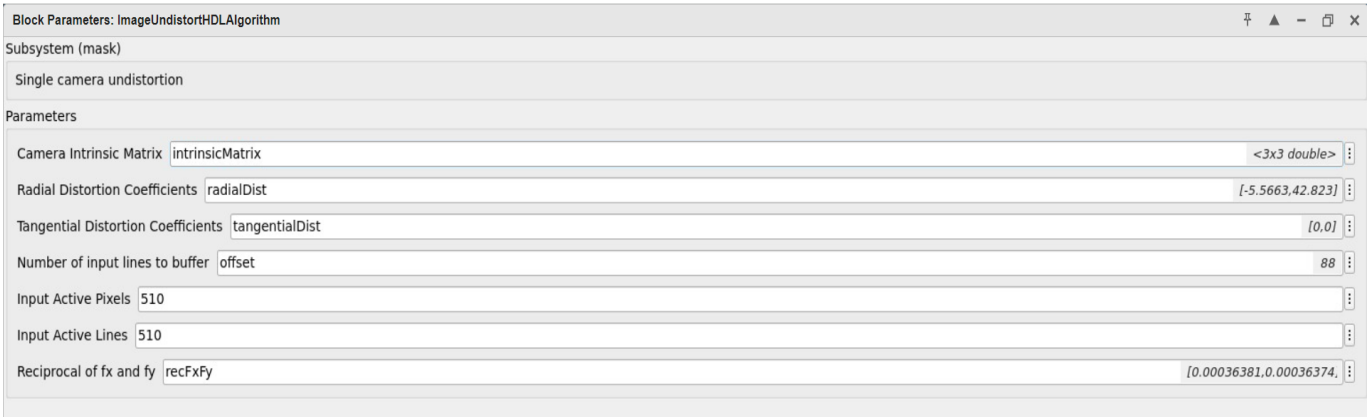


Fig 4.6 Camera Calibration Parameters for Undistortion

4.9 CONFIGURING SYSTEM GENERATOR

Firstly, the Xilinx Vivado setup has been downloaded and opened, in the wizard window , the system generator toolbox is been checked and the corresponding toolbox is been downloaded . After downloading the System Generator, it appears as an application on the desktop. The System generator is then opened and the compatible version of matlab is been configured with it. After the configuration, the configured matlab is been opened and Simulink of that particular matlab version is launced and the Xilinx Toolkit is found in the Library Browser. These blocks are then used to simulate the blocks of the required algorithm in the Simulink workspace. Fig 4.7 shows the different steps involved in configuring System generator to Matlab software.

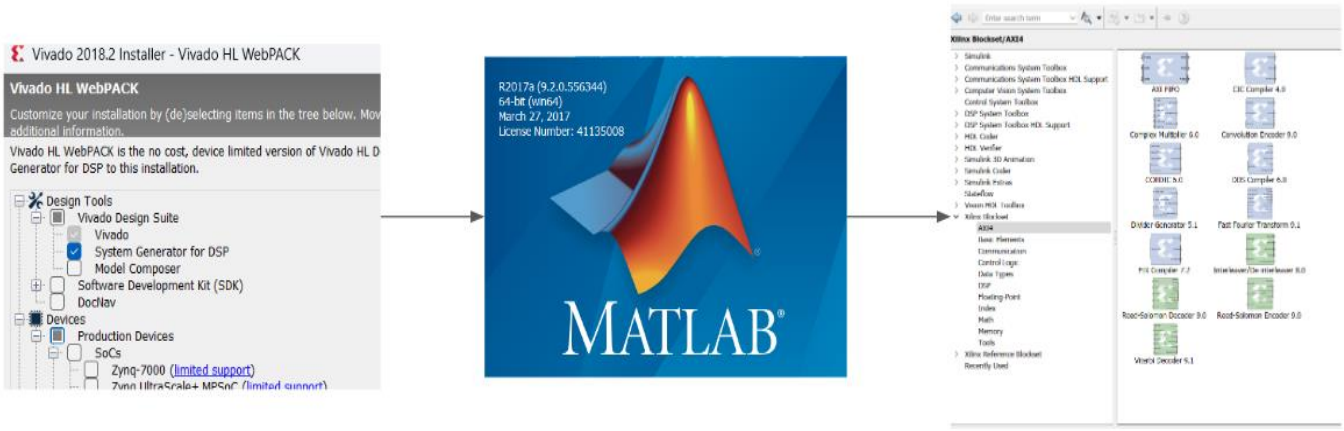
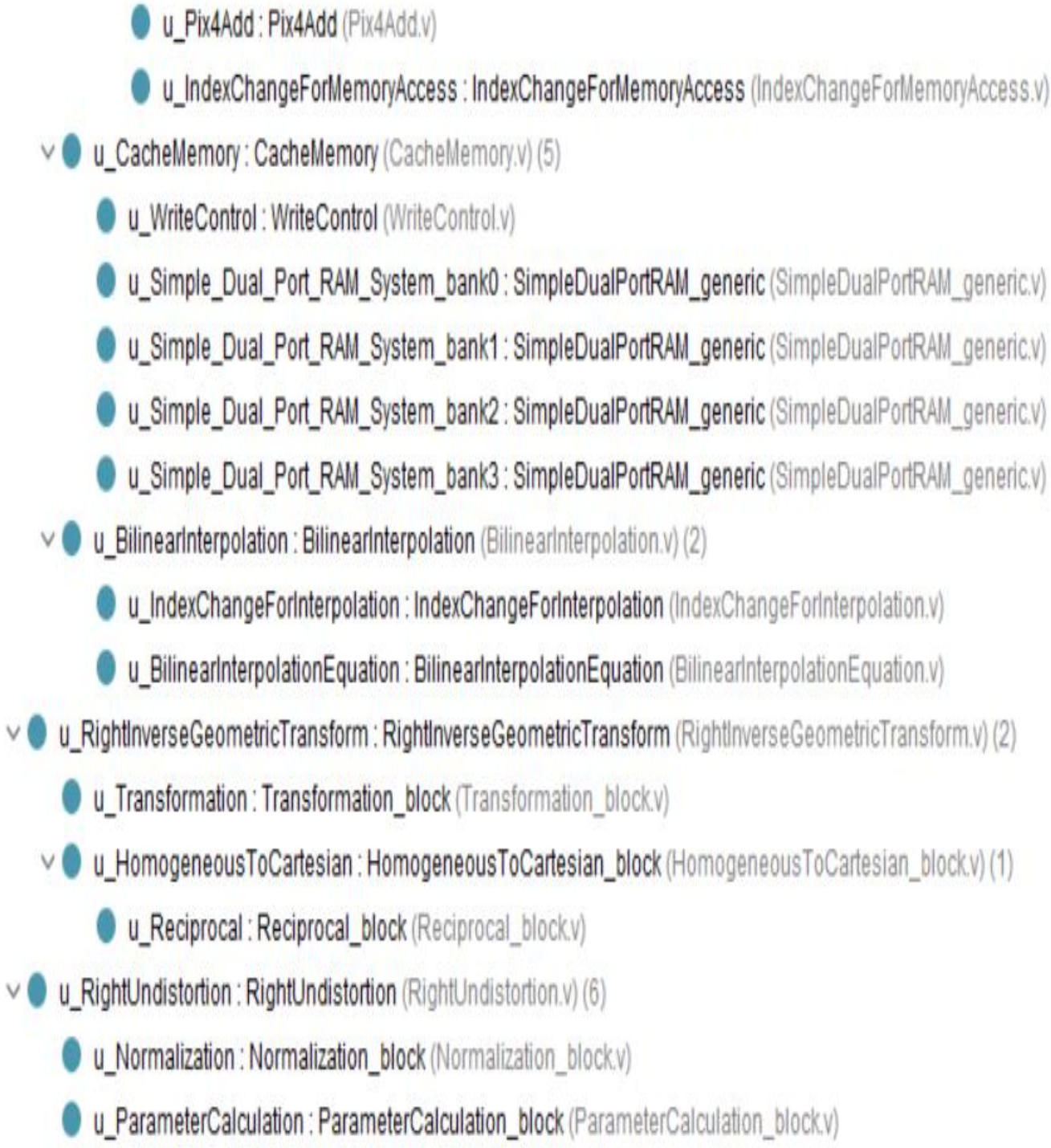
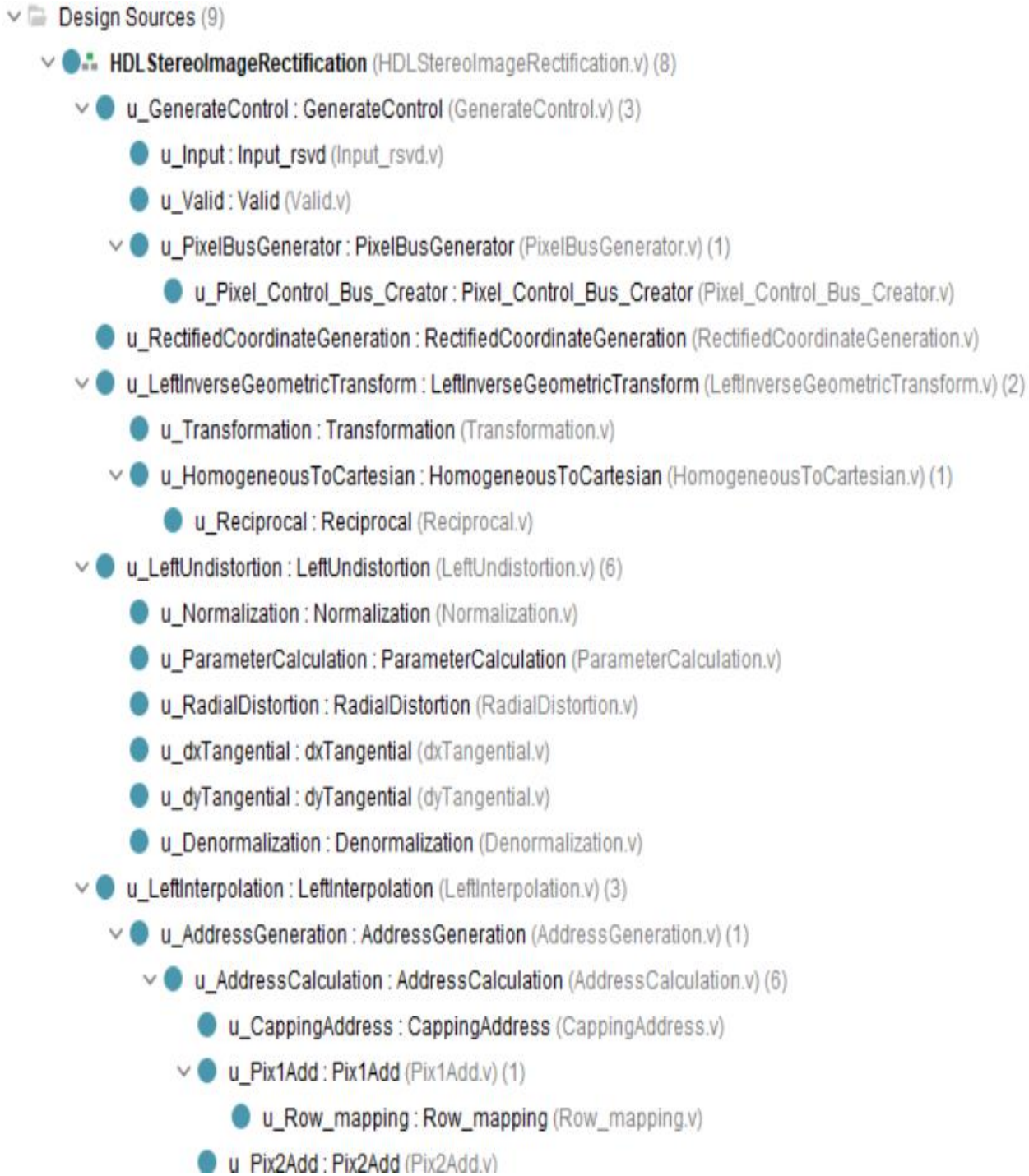


Fig 4.7 Steps involved in Configuring System Generator

4.10 IMPLEMENTATION IN XILINX VIVADO

The below modules were obtained using HDL Coder which converts matlab code directly into Verilog code which can be implemented in vivado. Fig 4.8 show the number of modules that are generated by HDL coder corresponding to the functionality of each blocks in Simulink model. These design source modules are organized into three level of hierarcy where modules are invoked where ever they are required [3]. The code is generated in such a way that onboard BRAM can be used for storing the streams of pixel values [2]. These streams are then processed in stacks and then the signals atre send to the display unit wherfe the rectified and undistorted oimgae is displayed [3]. Thus ample amount of latency has also been reduced [3].


u_Pix4Add : Pix4Add (Pix4Add.v)
u_IndexChangeForMemoryAccess : IndexChangeForMemoryAccess (IndexChangeForMemoryAccess.v)
u_CacheMemory : CacheMemory (CacheMemory.v) (5)
u_WriteControl : WriteControl (WriteControl.v)
u_Simple_Dual_Port_RAM_System_bank0 : SimpleDualPortRAM_generic (SimpleDualPortRAM_generic.v)
u_Simple_Dual_Port_RAM_System_bank1 : SimpleDualPortRAM_generic (SimpleDualPortRAM_generic.v)
u_Simple_Dual_Port_RAM_System_bank2 : SimpleDualPortRAM_generic (SimpleDualPortRAM_generic.v)
u_Simple_Dual_Port_RAM_System_bank3 : SimpleDualPortRAM_generic (SimpleDualPortRAM_generic.v)
u_BilinearInterpolation : BilinearInterpolation (BilinearInterpolation.v) (2)
u_IndexChangeForInterpolation : IndexChangeForInterpolation (IndexChangeForInterpolation.v)
u_BilinearInterpolationEquation : BilinearInterpolationEquation (BilinearInterpolationEquation.v)
u_RightInverseGeometricTransform : RightInverseGeometricTransform (RightInverseGeometricTransform.v) (2)
u_Transformation : Transformation_block (Transformation_block.v)
u_HomogeneousToCartesian : HomogeneousToCartesian_block (HomogeneousToCartesian_block.v) (1)
u_Reciprocal : Reciprocal_block (Reciprocal_block.v)
u_RightUndistortion : RightUndistortion (RightUndistortion.v) (6)
u_Normalization : Normalization_block (Normalization_block.v)
u_ParameterCalculation : ParameterCalculation_block (ParameterCalculation_block.v)


Design Sources (9)
HDLStereoImageRectification (HDLStereoImageRectification.v) (8)
u_GenerateControl : GenerateControl (GenerateControl.v) (3)
u_Input : Input_rsvd (Input_rsvd.v)
u_Valid : Valid (Valid.v)
u_PixelBusGenerator : PixelBusGenerator (PixelBusGenerator.v) (1)
u_Pixel_Control_Bus_Creator : Pixel_Control_Bus_Creator (Pixel_Control_Bus_Creator.v)
u_RectifiedCoordinateGeneration : RectifiedCoordinateGeneration (RectifiedCoordinateGeneration.v)
u_LeftInverseGeometricTransform : LeftInverseGeometricTransform (LeftInverseGeometricTransform.v) (2)
u_Transformation : Transformation (Transformation.v)
u_HomogeneousToCartesian : HomogeneousToCartesian (HomogeneousToCartesian.v) (1)
u_Reciprocal : Reciprocal (Reciprocal.v)
u_LeftUndistortion : LeftUndistortion (LeftUndistortion.v) (6)
u_Normalization : Normalization (Normalization.v)
u_ParameterCalculation : ParameterCalculation (ParameterCalculation.v)
u_RadialDistortion : RadialDistortion (RadialDistortion.v)
u_dxTangential : dxTangential (dxTangential.v)
u_dyTangential : dyTangential (dyTangential.v)
u_Denormalization : Denormalization (Denormalization.v)
u_LeftInterpolation : LeftInterpolation (LeftInterpolation.v) (3)
u_AddressGeneration : AddressGeneration (AddressGeneration.v) (1)
u_AddressCalculation : AddressCalculation (AddressCalculation.v) (6)
u_CappingAddress : CappingAddress (CappingAddress.v)
u_Pix1Add : Pix1Add (Pix1Add.v) (1)
u_Row_mapping : Row_mapping (Row_mapping.v)
u_Pix2Add : Pix2Add (Pix2Add.v)

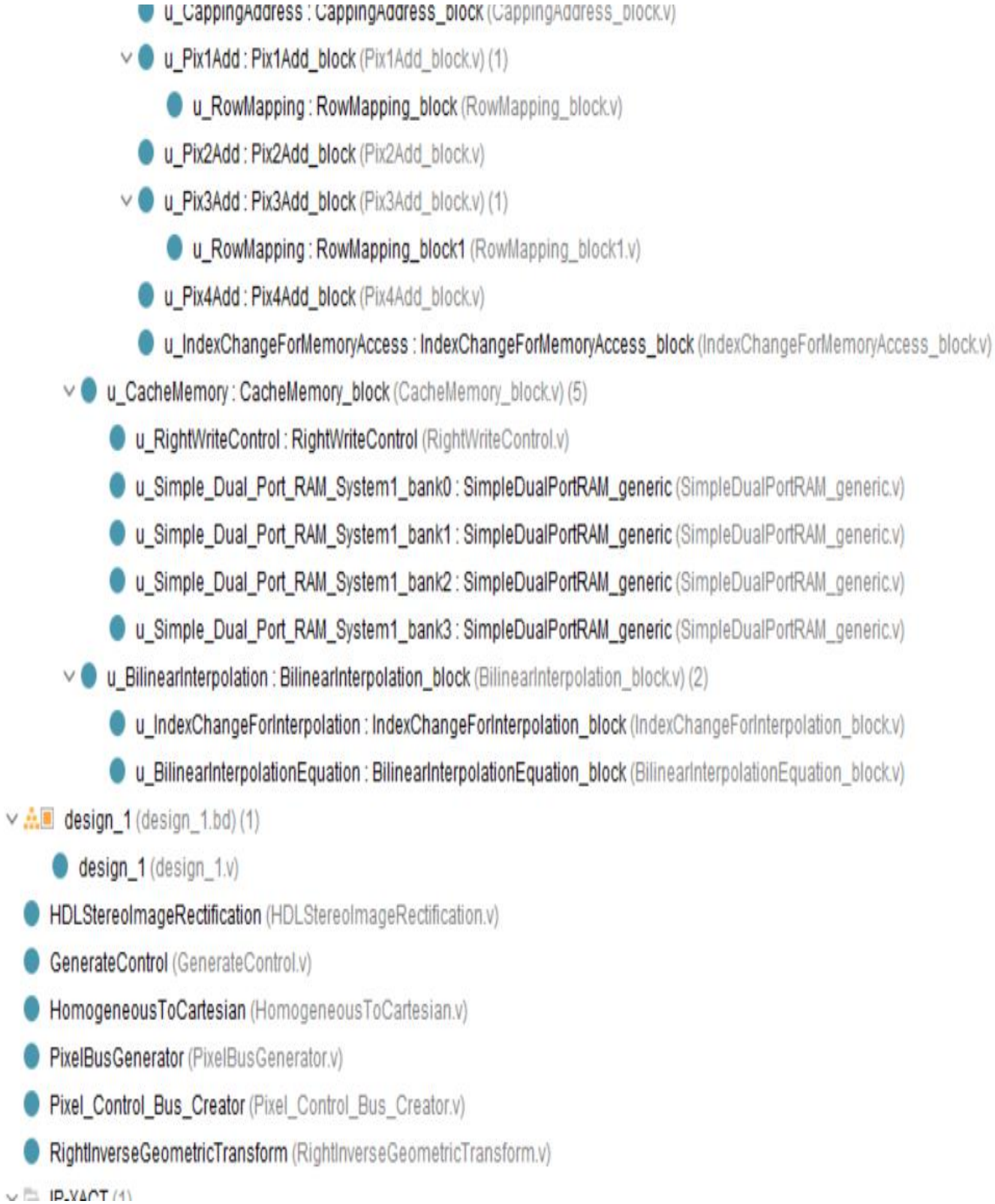


Fig 4.8 Modules included in Xilinx Vivado

4.11 HARDWARE IMPLEMENTATION

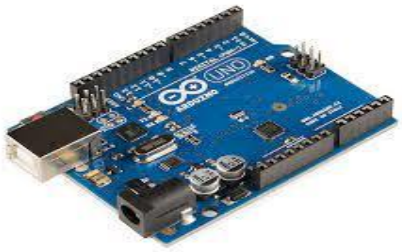


Fig 4.9 Arduino UNO Board

Fig 4.9 shows the typical image of an Arduino uno board [3]. Incorporating Arduino Uno into the project adds a tangible hardware component to complement the software simulations [2]. Arduino Uno serves as a versatile microcontroller platform, capable of interfacing with various sensors and actuators, thus expanding the project's capabilities beyond mere simulation [1]. By integrating Arduino Uno, we can interface with real-world devices such as cameras, motors, or sensors, enhancing the project's practical applicability. For instance, Arduino Uno can facilitate real-time acquisition of image data from cameras or enable physical manipulation of objects based on the processed image data. Additionally, Arduino Uno's ease of use and extensive community support streamline the development process, allowing for rapid prototyping and iteration [1]. Overall, the integration of Arduino Uno augments our project with tangible hardware interaction, bridging the gap between simulation and real-world

4.11.1 ARDUINO UNO SPECIFICATION

The Arduino Uno is a popular microcontroller board renowned for its simplicity, versatility, and wide range of applications. Here are some of its key specifications:

1. **Microcontroller**: The Arduino Uno is Powered by the ATmega328P microcontroller, clocked at 16 MHz.
2. **Digital I/O Pins**: It features 14 digital input/output pins (of which 6 can be used as PWM outputs) for interfacing with various sensors, actuators, and other devices.
3. **Analog Inputs**: The Uno includes 6 analog inputs, allowing for analog sensor interfacing.
4. **Operating Voltage**: It operates at 5 volts, making it compatible with a wide range of sensors and components.
5. **Input Voltage:** The recommended input voltage for the Uno is 7-12 volts. However, it can accept voltages as low as 6 volts and as high as 20 volts.
6. **Flash Memory:** The ATmega328P microcontroller on the Uno has 32 KB of flash memory, of which 0. 5 KB is used by the bootloader.
7. **SRAM:** It has 2 KB of SRAM, which is used for storing variables and other runtime data.

8. **EEPROM:** The Uno has 1 KB of EEPROM for non-volatile storage, allowing data to be retained even when Power is removed.
9. **Communication:** It supports serial communication via USB and UART, making it easy to connect to computers and other devices.
10. **Programming:** The Uno can be programmed using the Arduino Software (IDE), which provides a simple and intuitive development environment based on the C and C++ programming languages.

4.11.2 HARDWARE CONFIGURATION

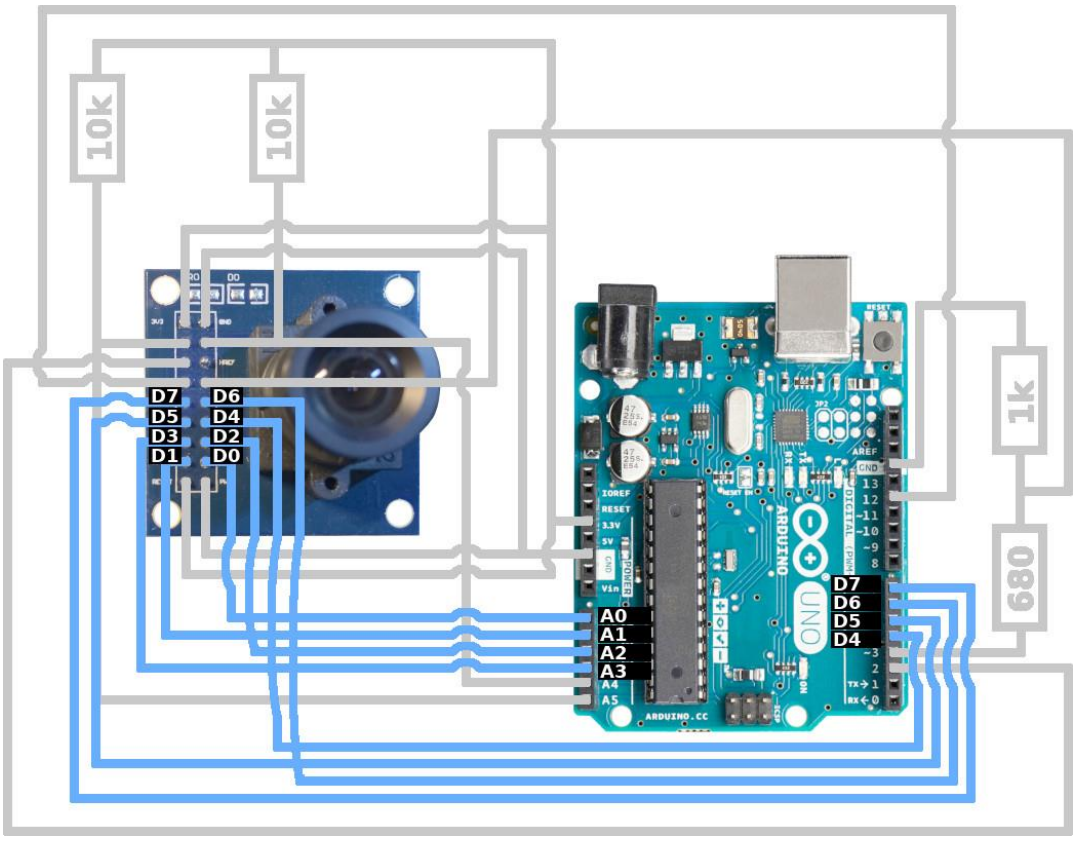


Fig 4.10 Hardware Configuration

Fig 4.10 shows the circuit configuration between Arduino UNO and OV7670. The connection steps are, make a voltage divider from the Arduino pin 3 to the XCLK pin of the camera. Make the I2C connections. Arduino pin A5 to SIOC and Arduino pin A4 to SIOD [3]. Then add a 10k pull-up resistors to 3.3V to both of the wires (A5 to 10k to 3. 3V, A4 to 10k to 3. 3V). VSYNC to Arduino pin 2. PCLK to Arduino pin 12. Connect Power to the camera. From Arduino 3. 3V pin to the camera's 3. 3V input and from Arduino GND pin to the camera's GND. Connect the camera's RESET pin to 3. 3V and PWDN to GND. Connect Camera's D0 to D3 to the Arduino pins A0 to A3. Connect Camera's D4 to D7 to the Arduino pins 4 to 7.

4.11.3 CAMERA MODULE OV7670

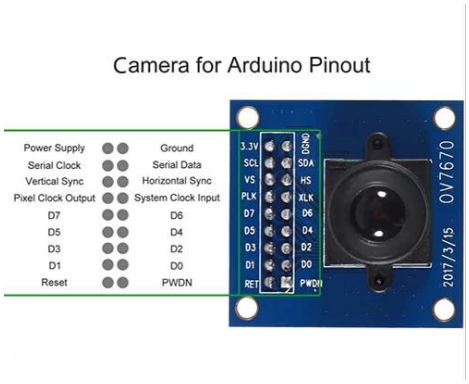


Fig 4.11 Camera Module OV7670

- High sensitivity for low-light operation
- Low operating voltage for embedded portable applications
- Standard SCCB interface compatible with I2C interface
- Output support for Raw RGB, RGB (GRB 4:2:2,
- Automatic image control functions including Automatic Exposure Control (AEC), Automatic Gain Control (AGC), Automatic White Balance (AWB), Automatic Band Filter (ABF), and Automatic Black-Level Calibration (ABLC)
- Image quality controls include color saturation, hue, gamma, sharpness (edge enhancement), and anti-blooming
- ISP includes noise reduction and defect correction
- Supports LED and flash strobe mode
- Saturation level auto adjust (UV adjust)
- Edge enhancement level auto adjust

4.11.4 IMAGE CAPTURING USING OV7670

Capturing images with the OV7670 camera module involves connecting it (Power, ground, SCCB) to a microcontroller board like Arduino. Libraries are available to initialize communication and configure settings like resolution and format (YUV or RGB565). The library offers functions to capture single frames or stream images [2]. Remember to consult the datasheet for pin details and choose a compatible library for your microcontroller. Experiment with settings to optimize image quality for your project. Fig 4.12 shows two different images that are captured at a particular frame. These frames are then stored in a particular matlab directly from where it is directly given to matlab for the algorithm implementation

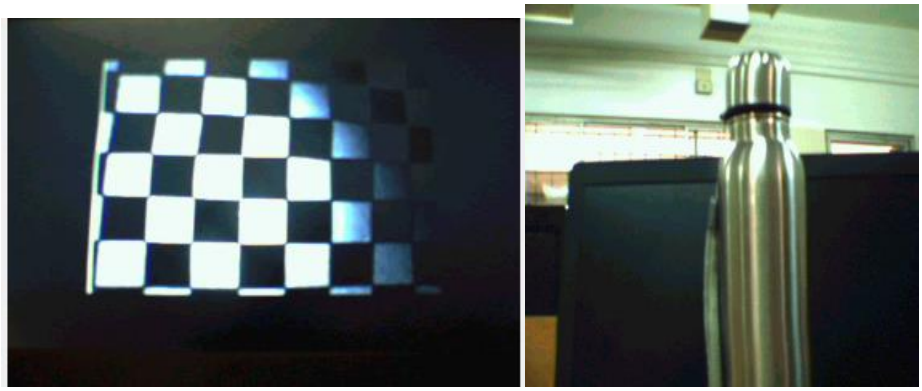


Fig 4.12 Images Captured using OV7670

4.11.5 FRAME BASED REAL TIME DATA COLLECTION

In frame-based real-time image processing, individual video frames are captured sequentially by a camera or other source. Each frame undergoes image processing algorithms on the fly, meaning results are generated and potentially displayed within a timeframe that allows for near-instantaneous interaction or analysis [2]. This approach is crucial for applications like object recognition in robotics, where immediate decisions need to be made based on visual data. Fig 4.13 shows the frame based capturing of the image that is captured in the Arduino IDE serial window.

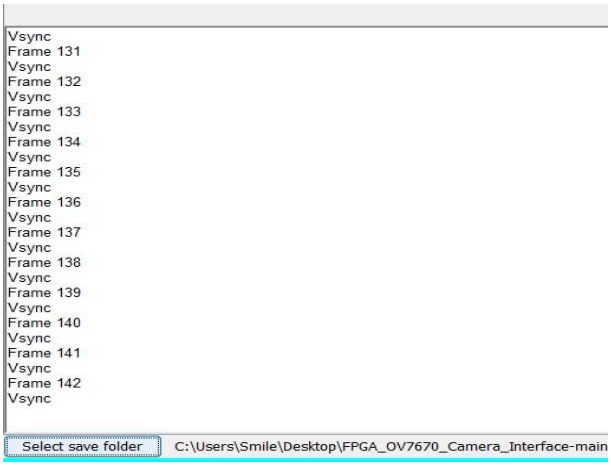


Fig 4.13 Frame based Data

4.12 ARDUINO IMPLEMENTATION

Fig 4.14 shows the hardware implementation that is done in Arduino Board [3]. There are several c++ modules that are written in the Arduino IDE and is dumped into the board [3]. After the code is dumped, image is captured by frame wise from ov7670 module that is connected tio the board. As a sign of indication the transmitter LED glows [2]. Each time when the LED glows image is transmitted from the board to the Arduino IDE and the image gets displayed in ArduImage capture window. ArduImage Capture must be downloaded before to implement.

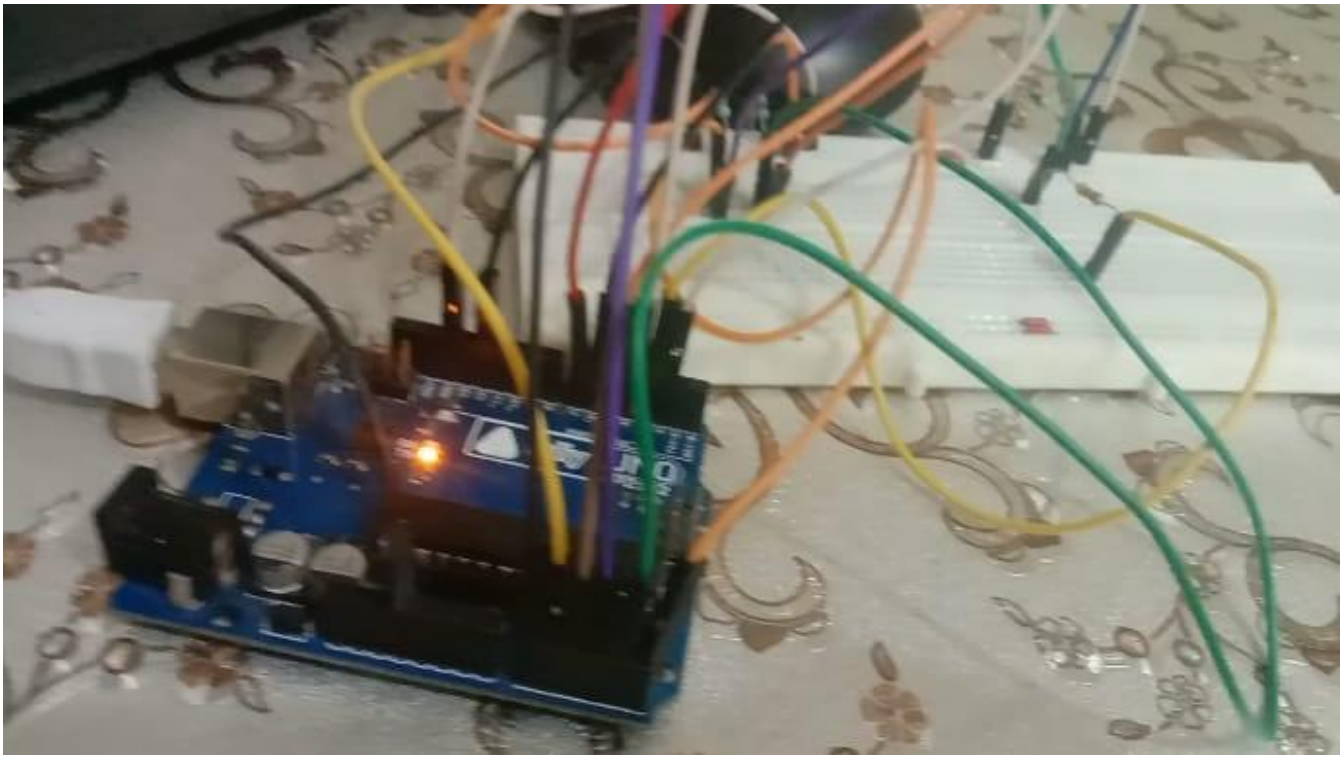


Fig 4.14 Implementation in Arduino Board

CHAPTER 5

RESULTS AND ANALYSIS

Various results and result analysis are performed for this project. Using a Xilinx Vivado tool Area, Power and Synthesis analysis is been performed [3]. On performing Area and Power Analysis both seems to reduce proving that the algorithm is verry effective.

5.1 MATLAB RESULTS

The project utilized MATLAB for various purposes, including simulating and converting a model of a stereo rectification and undistortion system to HDL using the HDL Coder tool. The HDL was then compiled to a bit file and targeted to the FPGA-based camera system. Taken from dataset

5.1.1 IMAGE DEMOSAICING

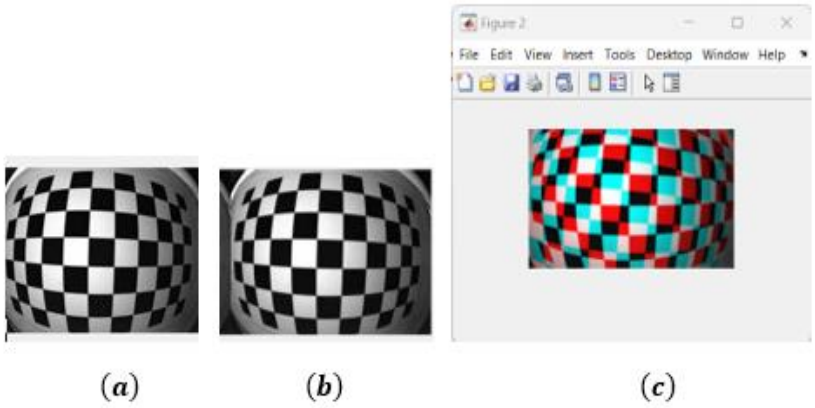


Fig 5.1 Image Demosaicing Results

The purpose of image demosaicing is to reconstruct a full-color image from the incomplete color information captured by a single sensor with a color filter array (CFA). The process involves interpolating the missing color values for each pixel by using the available color information from neighboring pixels [2]. This is necessary because in a CFA, each pixel only captures one color channel (red, green, or blue), and demosaicing is used to estimate the missing color channels for each pixel in order to create a complete RGB image. Here, the above figure 5.1 (a)represents the left image figure 5.1(b)represents the left image and 5. 1(c)represents the demosaiced image. By performing demosasicing as the initial step of the complete algorithm , the 3d perception and depth information is been obtained [3].

5.1.2 STAGEWISE OUTPUTS OBTAINED

Stage wise output is obtained by separately performing each block's functionality using optimized matlab code. Figures below show the results of image transformations that had occurred at each stage [2]. Fig 5.4 is the image that was a real time capture from Arduino board, the changes that takes place at each stage is quite obvious and relevant.

Fig 5.2 Stage-wise Output for Tower Image

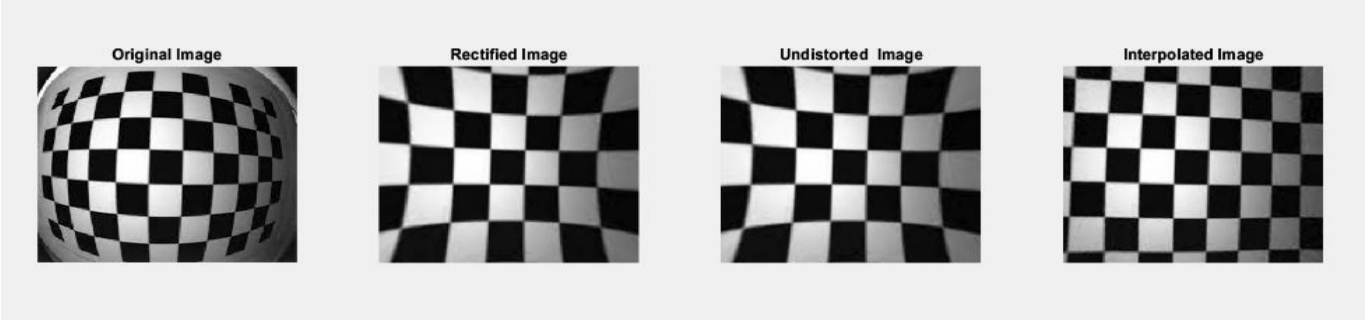


Fig 5.3 Stage-wise Output for Chessboard Image

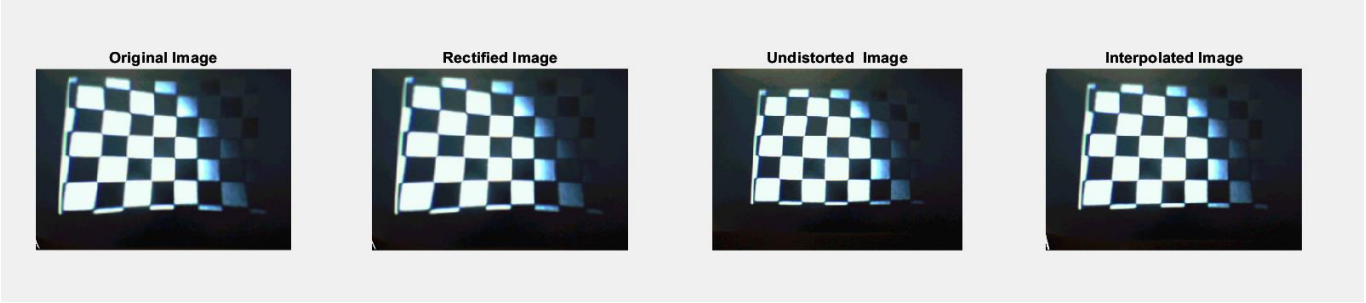


Fig 5.4 Stage-wise Output for Real Time Image

5.2 SIMULINK RESULTS

When compared to results obtained from matlab, the Simulink results seems to be more accurate and clear because the functional blocks are very much optimized with both built-in functions and built in accurate values which paves the way for accurate transformatiuon [1].

5.2.1 IMAGE RECTIFICATION RESULTS

The below figure Fig 5.5 shows the original image and the final rectified image. The transformation is quite evident that the spherical aberration that was caused due to tangential and radial distortion components is been reduced to larger extent**.**

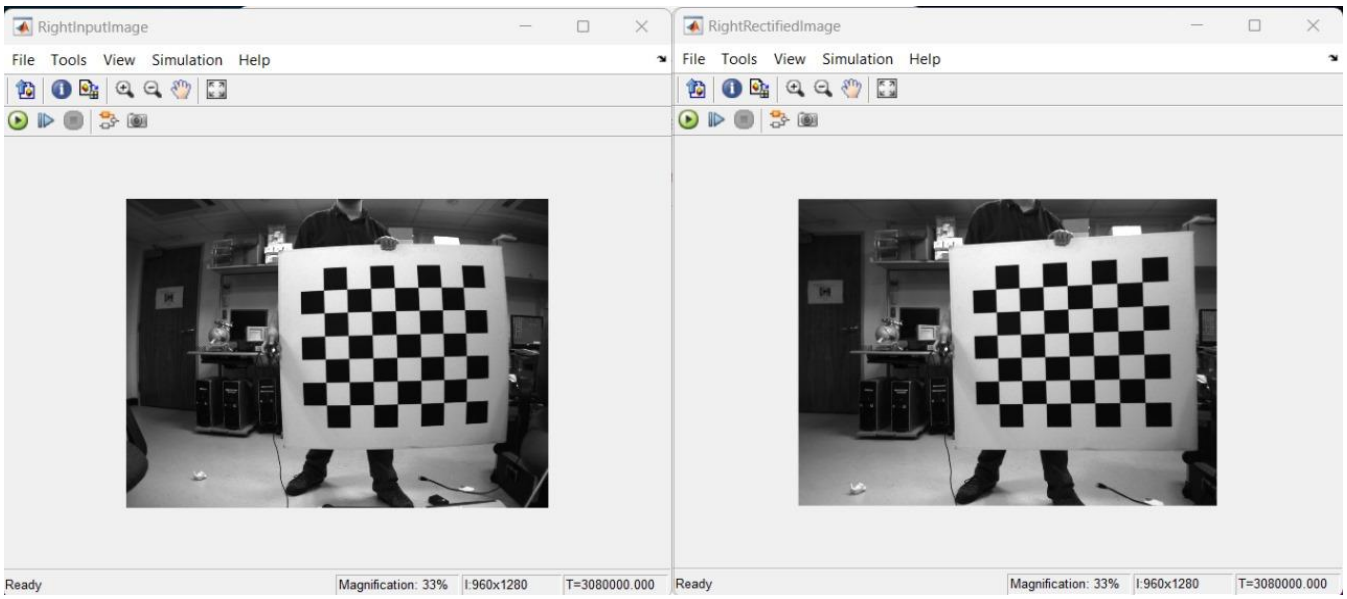


Fig 5.5 Simulink Results of Image Rectification

5.2.2 IMAGE UNDISTORTION RESULTS

Spherical Distortions that arise due to tangential and radial components are been removed under the image undistortion block. In the below figure, Fig 5.7, the original images with spherical edges has been processed and all the distortions are removed [3]. During processing an ample collection of dataset is provided to the camera calibration toolbox, where the extent of distortion has been calculated, which can eventually be used to remove distortions from original image using feedback mechanism**.**

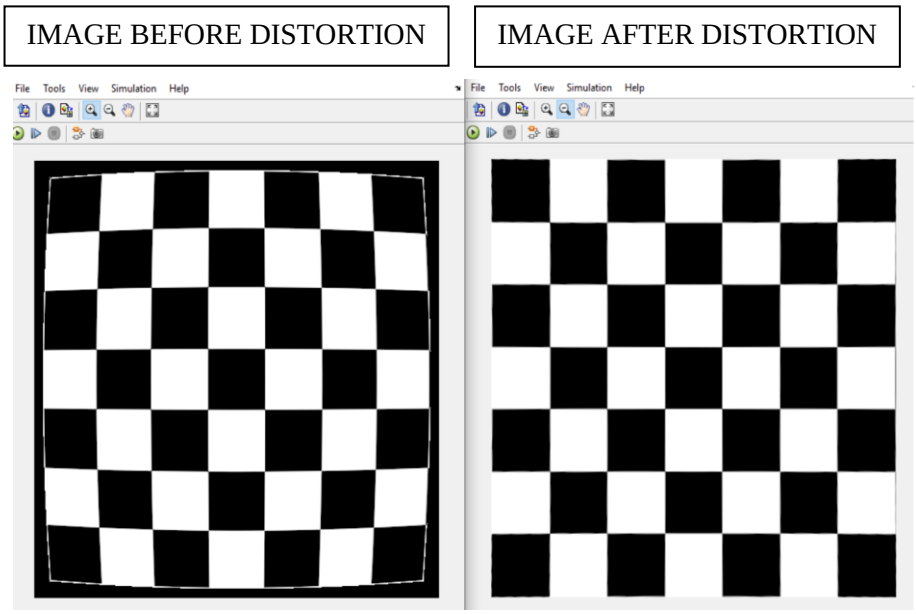
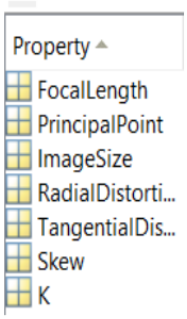


Fig 5.6 Simulink Results for Undistortion

5.3 CAMERA CALIBRATION RESULTS

Multiple images are uploaded into camera calibration toolbox present inside matlab toolbox and the intrinsic and extrinsic parameters are calibrated and are stored as values inside the matrix variable, 'CameraParams'. The intrinsic parameters include optical center, focal length. The extrinsic camera parameters include camera position and its 3d orientation [1].



S.No	FROM DATASET
1.Image 1	cam0=[589.249 0 313.055; 0 589.249 233.89; 0 0 1] cam1=[589.249 0 331.435; 0 589.249 233.89; 0 0 1]
2.Image 2	cam0=[1625.408 0 199.345; 0 1625.408 240.165; 0 0 1] cam1=[1625.408 0 284.364; 0 1625.408 240.165; 0 0 1]
3.Image 3	cam0=[989.087 0 224.787; 0 989.087 237.567; 0 0 1] cam1=[989.087 0 253.722; 0 989.087 237.567; 0 0 1]
4.Image 4	cam0=[1610.51 0 -57.92; 0 1610.51 228.42; 0 0 1] cam1=[1610.51 0 -5.707; 0 1610.51 228.42; 0 0 1]

Fig 5.7 Camera Calibration Parameters for Different Images

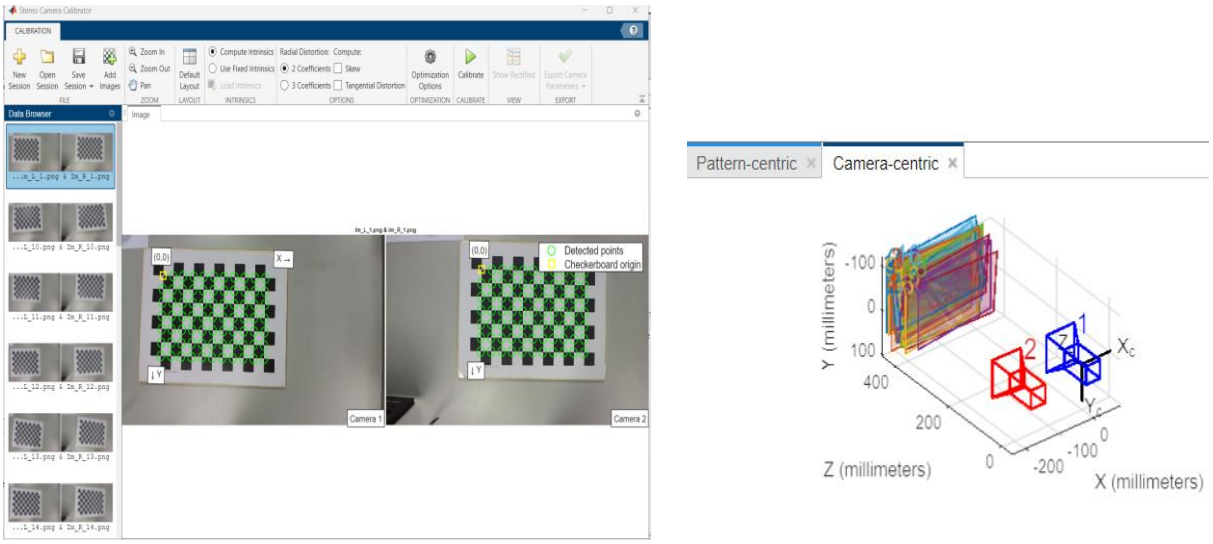


Fig 5.8 Camera Calibration Process

Fig 5.8 shows the camera calibration parameters and its values below are the explanation for each parameter [2].

- **FOCAL LENGTH:**
  The focal length of a camera lens is essentially the distance between the lens and the image sensor when the lens is focused on a subject
- **PRINCIPAL POINT:**
  The principal point in a camera refers to the point on the image sensor or film where the light rays converge after passing through the camera lens
- **RADIAL DISTORTION:**
  Radial distortion is a type of optical distortion that occurs in camera lenses, particularly in wide-angle and fisheye lenses [2].
  Types: Barrel Distortion, Pincushion Distortion [1].
- **TANGENTIAL DISTORTION:**
  Tangential distortion is another type of optical distortion that can occur in camera lenses [2]. Unlike radial distortion, which affects the straightness of lines, tangential distortion involves the displacement of points away from the center of the image. This type of distortion causes points away from the image center to be shifted either inward or outward [3].
- **SKEW:**
  Skew in images refers to a form of distortion where the image appears tilted or slanted,
- **K:** Scaling constant

Property	Value
FocalLength	[721.0813,732.03...
PrincipalPoint	[525.2292,284.65...
ImageSize	[576,1024]
RadialDistorti...	[0.0113,-0.0694]
TangentialDis...	[0,0]
Skew	0
K	[721.0813,0,525.2...

Property	Value
FocalLength	[717.9143,728.56...
PrincipalPoint	[512.4248,296.02...
ImageSize	[576,1024]
RadialDistorti...	[0.0058,-0.0513]
TangentialDis...	[0,0]
Skew	0
K	[717.9143,0,512.4...

Fig 5.9 Camera Calibration Results

5.4 XILINX VIVADO RESULTS

Xilinx Vivado serves as an excellent tool for RTL, Area, Power and Timing analysis, the following are the results obtained from Xilinx Vivado.

5.4.1 RTL SYNTHESIS RESULTS

RTL (Register Transfer Level) analysis in Xilinx Vivado plays a critical role in the design and verification of digital circuits implemented on FPGAs (Field-Programmable Gate Arrays). RTL analysis involves examining the behavior and structure of the digital circuit at the register transfer level, focusing on how data moves between registers and how operations are performed [3]. In Vivado, RTL analysis tools provide comprehensive insights into the design's functionality, performance, and timing characteristics [2]. One key aspect of RTL analysis in Vivado is timing analysis, which evaluates the timing constraints and ensures that the design meets timing requirements [2]. Timing analysis examines the propagation delays through various paths in the design, identifying potential timing violations such as setup and hold time violations [2]. Fig 5.10 shows the RTL Analysis that is obtained from Xilinx Vivado.

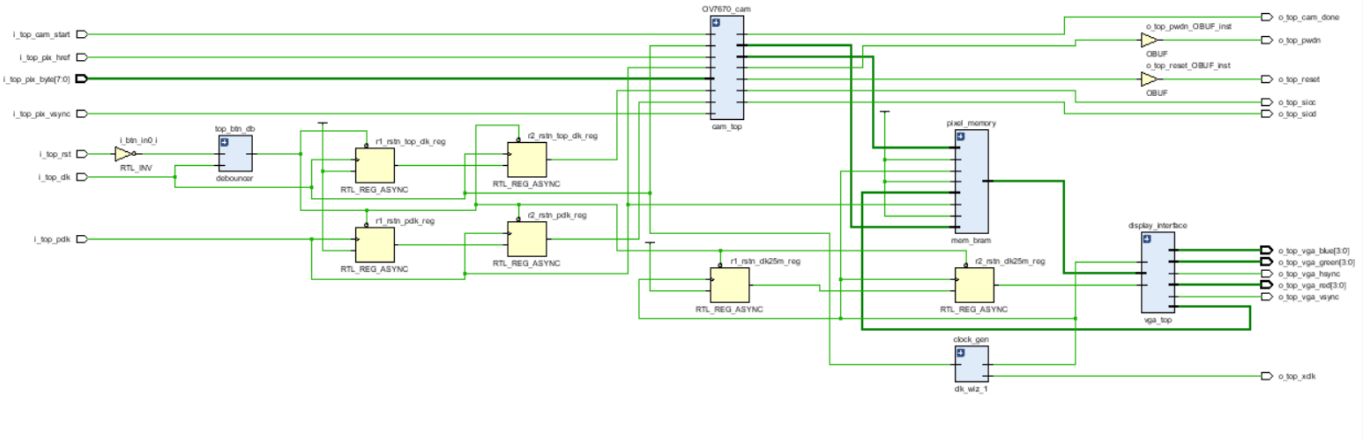


Fig 5.10 RTL Schematic

5.5 PERFORMANCE METRICES ANALYSIS

Performance metrics such as Average Mean Error (AME) and Peak Signal-to-Noise Ratio (PSNR) are fundamental in signal processing, providing quantitative measures to evaluate the quality and accuracy of processed signals [2]. These metrics are calculated using Matlab Software environment where necessary formulas are implemented in form of matlab code [2]. The Average Mean Error (AME) is a statistical measure that quantifies the average discrepancy between observed values and predicted values in a dataset. It is calculated by taking the average of the absolute differences between each observed value and its corresponding predicted value. AME provides insights into the overall accuracy and precision of a predictive model or signal processing algorithm. On the other hand, the Peak Signal-to-Noise Ratio (PSNR) is a widely used metric to assess the fidelity of a processed signal, particularly in image and video processing. PSNR measures the ratio between the maximum possible Power of a signal and the Power of corrupting noise that affects its fidelity, expressed in decibels (dB).

5.5.1 AVERAGE MEAN ERROR

Average Mean Error (AME) or Mean Absolute Error (MAE) is a metric that calculates the average absolute difference between corresponding pixels in the original and reconstructed images [2]. It provides a measure of the average magnitude of the errors between the images [2]. The formula is:

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} \qquad \text{..... Eqn 5.1}$$

Table 2 Average Mean Error for Different Images

INPUT IMAGE	BILINEAR INTERPOLATION (EXISTING METHODOLOGY)	BICUBIC INTERPOLATION (PROPOSED METHODOLOGY)
IMAGE 1:	30.7524	29.9998
IMAGE 2:	19.5955	5.8631
IMAGE 3:	12.4806	3.3305
IMAGE 4:	16.9365	9.9014
IMAGE 5:	15.6556	8.1108
IMAGE 6:	15.4446	12.5608
IMAGE 7:	22.9396	10.4175
IMAGE 8:	19.8762	14.7539
IMAGE 9:	23.4587	19.7276
IMAGE 10:	15.3987	15.0981
IMAGE 11:	23.7128	13.0889
IMAGE 12:	13.8486	13.6931
IMAGE 13:	37.372	7.8876
IMAGE 14:	58.0333	11.3284
IMAGE 15:	44.0547	10.8327
AVG	**24. 62732**	**12. 4396**

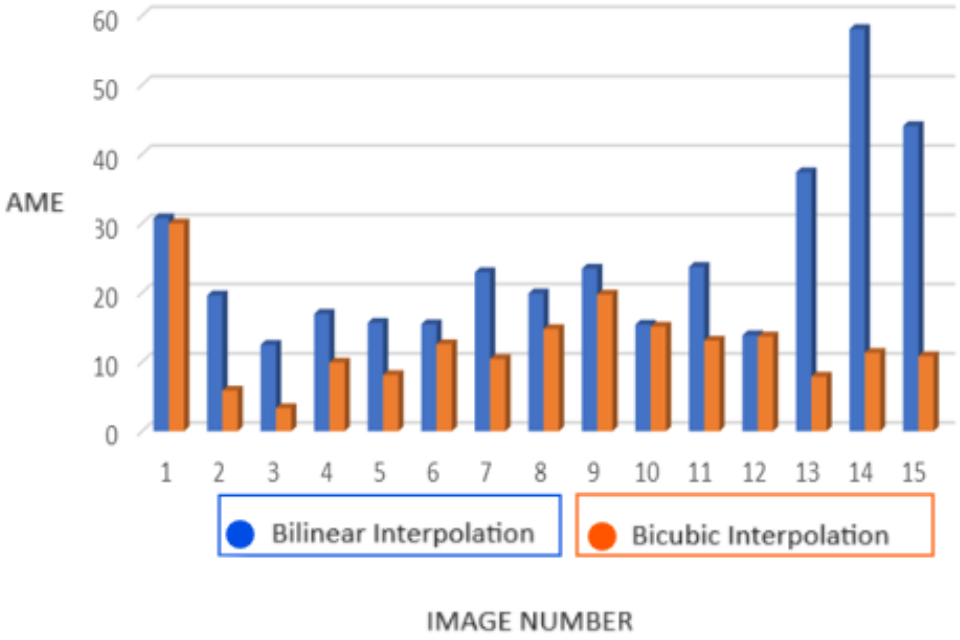


Fig 5.11 Bar Graph Representation of AME

5.5.2 PEAK SIGNAL TO NOISE RATIO

PSNR is a metric used to measure the quality of an image or signal in the presence of noise. It's the ratio between the maximum value of a signal and the Power of noise that affects its quality.

Table 3 PSNR Results

EXISTING METHODOLOGY	PROPOSED METHODOLOGY
IMAGE 1: PSNR VALUE:14. 7474	IMAGE 1: PSNR VALUE:17. 3593
IMAGE 2: PSNR VALUE:17. 0125	IMAGE 2: PSNR VALUE:17. 0707
Avg: 15.87995	**Avg: 17.215**

5.6 RESULT ANALYSIS

Result analysis plays a crucial role in this project, result analysis includes Area, Power and Timing Analysis which is important for any design.

5.6.1 AREA ANALYSIS

Area analysis involves assessing the utilization of FPGA resources, including the number of logic cells, DSP slices, block RAMs and other resources used by the design [1]. Vivado provides detailed reports on the resource utilization, helping designers to optimize the design to efficiently utilize the available resources. Effective Area utilization is very crucial for achieving a compact and cost-effective implementation [1]. Fig 5.12 shows that the Area and utilization of different resources that are present has been drastically reduced which makes the Algorithm more flexible and efficient. The number of LUTs are reduced by 61.764%.

EXISTING METHODOLOGY

UTILIZATION REPORT:

Resource	Estimation	Available	Utilization ...
LUT	18346	53200	34.48
LUTRAM	1127	17400	6.48
FF	14873	106400	13.98
BRAM	128	140	91.43
DSP	168	220	76.36
IO	52	200	26.00
BUFG	1	32	3.13

PROPOSED METHODOLOGY

UTILIZATION REPORT:

Resource	Estimation	Available	Utilization ...
LUT	18137	134600	13.47
LUTRAM	1096	46200	2.37
FF	14630	269200	5.43
BRAM	128	365	35.07
DSP	158	740	21.35
IO	52	285	18.25
BUFG	1	32	3.13

Fig 5.12 Area Analysis

5.6.2 POWER ANALYSIS

Power analysis involves estimating the Power consumption of the design [1]. Vivado provides tools to evaluate Power at both the logical and physical levels [2]. Power estimation takes into account various factors such as activity factors, capacitance, switching frequency, and voltage levels [2]. Analyzing Power helps in optimizing the design to reduce Power consumption, which is crucial for energy efficiency and ensuring the design operates within specified Power budgets [2]. Fig 5.14 shows there is a drastic lowering of Power in proposed methodology when compared to existing methodology.

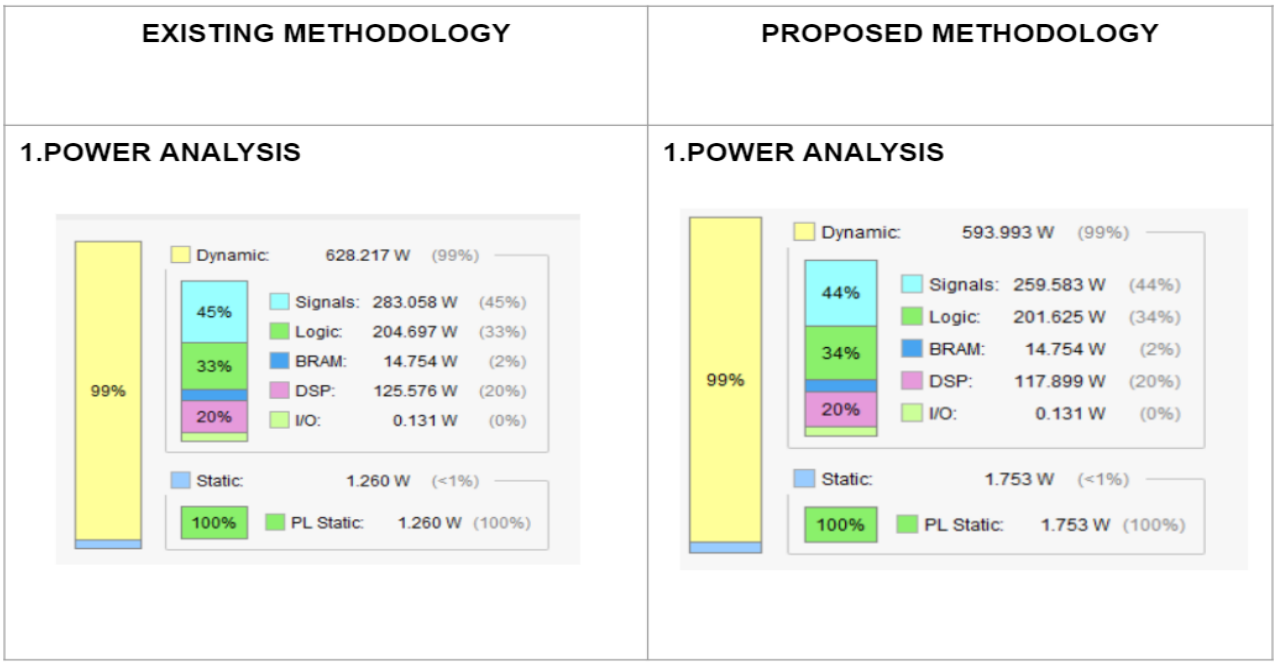


Fig 5.13 Power Analysis

5.6.3 TIMING ANALYSIS

Timing analysis evaluates the critical paths within the design to ensure that the design meets the specified timing requirements [2]. Vivado allows designers to analyze setup and hold times, clock-to-Q delays, and other timing constraints [2]. Timing analysis ensures that the design operates within the desired clock frequency and meets the required performance targets [2]. Fine-tuning the design based on timing analysis is essential for achieving the desired functionality while meeting performance goals [2].

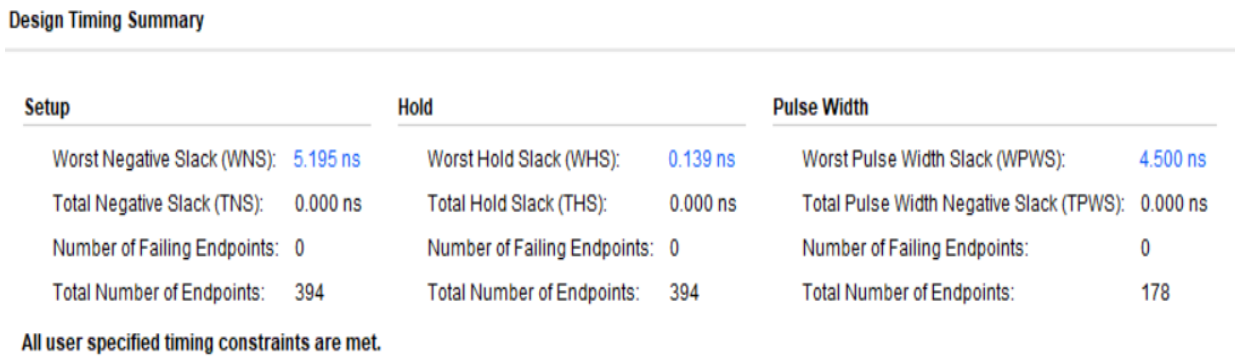


Fig 5.14 Timing Analysis

5.7 SUMMARY OF RESULTS

Based on the table provided below, it appears that the proposed methodology offers potential improvements in terms of Power consumption and efficiency. It reduces total Power consumption by 5.35% compared to the existing methodology. There are also significant reductions in the number of Look-Up Tables (LUTs) and Block RAMs (BRAMs) required, suggesting a more streamlined design with potentially less hardware complexity. These reductions come in at 61.764% and 61.536% respectively. However, the proposed methodology might come at a cost in terms of image quality. The PSNR (Peak Signal-to-Noise Ratio), a metric for image fidelity, shows an 8.407% increase, which could indicate a slight decrease in image quality. The AME (Absolute Mean Error), another metric for image quality, shows a significant improvement of 49.4%, suggesting the proposed methodology might be better at handling certain types of errors [2]. Overall, the proposed methodology seems to offer trade-offs between efficiency, hardware complexity, and image quality. It depends on the specific application requirements to determine if the efficiency gains outweigh the potential decrease in image quality.

Table 4 Summary of Results

	EXISTING METHODOLOGY	PROPOSED METHODOLOGY	% IMPROVEMENT
TOTAL POWER	629.471 W	595.743 W	**5.35% decrease**
NO OF LUT(in percentage)	34%	13%	**61.764% decrease**
NO OF BRAM(in percentage)	91%	35%	**61.536% decrease**
PSNR	15.87995	17.215	**8.407% increase**
AME	24.62732	12.4396	**49.4% decrease**

CHAPTER 6

CONCLUSION

In conclusion, the proposed algorithm is implemented using bicubic interpolation instead of bilinear interpolation. The project shows that the existing methodology is based on bilinear interpolation and proposed methodology is based on bicubic interpolation. Matlab simulation of performance metrices shows that the output image is been free of much error in proposed methodology rather than existing methodology. There are also significant reductions in the number of Look-Up Tables (LUTs) and Block RAMs (BRAMs) required, suggesting a more streamlined design with potentially less hardware complexity. These reductions come in at 61. 764% and 61. 536% respectively. However, the proposed methodology might come at a cost in terms of image quality. The PSNR (Peak Signal-to-Noise Ratio), a metric for image fidelity, shows an 8. 407% increase, which could indicate a slight decrease in image quality. The AME (Absolute Mean Error), another metric for image quality, shows a significant improvement of 49. 4%, suggesting the proposed methodology might be better at handling certain types of errors. Overall, the proposed methodology seems to offer trade-offs between efficiency, hardware complexity, and image quality.

Future scope of the project includes implementing the algorithm in any kind of FPGA platform, so that the algorithm can work with real time image processing, where live footage from real world can be captured and can be displayed on to screen after processing the real time data for rectification and undistortion. Future work entails potentially combining the demosaicing and the rectification operations into a single process is demonstrated and using HDL Coder to auto-generate the associated IP core. With larger FPGA fabrics it may also be beneficial to convert the entire algorithm into floating point logic such that the fixed point datapath does not need to be tweaked for various image sensors. Given more BRAM resources it would also be nice to create a full RGB rectification and undistortion algorithm.

CHAPTER 7

REFERENCES

[1] C. Junger, A. Hess, M. Rosenberger, and G. Notni, ‘‘FPGA-based lens undistortion and image rectification for stereo vision applications, ’’Proc. SPIE, vol. 11144, pp. 284–291, Sep. 2019, **DOI**: 10.1117/12.2530692.

[2] S. Michalik, S. Michalik, J. Naghmouchi, and M. Berekovic, ‘‘Real-time smart stereo camera based on FPGA-SoC,’’ in Proc. IEEE-RAS 17th Int. Conf. Humanoid Robot., pp. 311–317, Nov. 2017,
DOI: 10.1109/HUMANOIDS.2017.8246891.

[3] Y. Xu, Q. Zhou, L. Gong, M. Zhu, X. Ding, and R. K. F. Teng, ‘‘Highspeed simultaneous image distortion correction transformations for a multicamera cylindrical panorama real-time video system using FPGA,’’IEEE Trans. Circuits Syst. Video Technol., vol. 24, no. 6, pp. 1061–1069, Jun. 2014,
DOI: 10.1109/TCSVT.2013.2290576.

[4] J. Mun and J. Kim, ‘‘Real-time FPGA rectification implementation combined with stereo camera,’’ in Proc. Int. Symp. Consum. Electron. (ISCE), pp. 1–2, Jun. 2015, **DOI:** 10.1109/ISCE.2015.7177765

[5] K. S. Changan and P. G. Chilveri, ‘‘Stereo image feature matching using Harris corner detection algorithm**,**’’ in Proc. Int. Conf. Autom. Control Dyn. Optim. Techn., pp. 691–694, Sep. 2016, **DOI:** 10.1109/ICACDOT.2016.7877675.

[6] T. Kenter, H. Schmitz, and C. Plessl, ‘‘Kernel-centric acceleration of high accuracy stereo-matching**,**’’ in Proc. Int. Conf. Reconfigurable Comput. FPGAs, pp. 1–8, Dec. 2014,
DOI: 10.1109/ReConFig.2014.7032535.

[7] J. Zhu, Y. Li, Z. Ma, Y. Jiao, and B. Zhang, ‘‘Study on multiple image processing hardware system based on DSP**,**’’ in Proc. IEEE Int. Conf. Mechatronics Autom., pp. 1844–1849, Aug. 2014,
DOI: 10.1109/ICMA.2014.6885982.

[8] S. Perri, F. Frustaci, F. Spagnolo, and P. Corsonello, ‘‘Design of realtime FPGA-based embedded system for stereo vision,’’ in Proc. IEEE Int. Symp. Circuits Syst. (ISCAS), pp. 1–5, May 2018,
DOI: 10.1109/ISCAS.2018.8351886.

[9] M. Loni, A. Majd, A. Loni, M. Daneshtalab, M. Sjodin, and E. Troubitsyna, ‘‘Designing compact convolutional neural network for embedded stereo vision systems,’’ in Proc. IEEE 12th Int. Symp. Embedded Multicore/Many-Core Syst.-Chip (MCSoC), pp. 244–251, Sep. 2018..
DOI: 10.1109/MCSoC2018.2018.00049.

[10] J.-H. Kim, J.-K. Oh, S.-M. Kang, and J.-D. Cho, ‘‘A real-time rectification using an adaptive differential encoding for high-resolution video,’’ in Proc. 16th Int. Conf. Adv. Commun. Technol., pp. 666–670, Feb. 2014, **DOI:** 10.1109/ICACT.2014.6779046.

[11] S. N. Hung, J. Lee, and B.-J. You, ‘‘Real-time stereo rectification using compressed look-up table with variable breakpoint indexing,’’ in Proc.42nd Annu. Conf. IEEE Ind. Electron. Soc., pp.4814–4819,
oct 2016, **DOI:** 10.1109/IECON.2016.7793679.

[12] E. Staudinger, M. Humenberger, and W. Kubinger, ‘‘FPGA-based rectification and lens undistortion for a real-time embedded stereo vision sensor**,**’’ 2008.

[13] P. Di Febbo, S. Mattoccia, and C. D. Mutto, ‘‘Real-time image distortion correction: Analysis and evaluation of FPGA-compatible algorithms**,**’’ in Proc. Int. Conf. Reconfigurable Comput. FPGAs, Nov. 2016, pp. 1-6. **DOI:** 10.1109/ReConFig.2016.7857182

[14] D. W. Yang, L. C. Chu, C. W. Chen, J. Wang, and M. D. Shieh, ‘‘Depthreliability-based stereo-matching algorithm and its VLSI architecture design**,**’’ IEEE Trans. Circuits Syst. Video Technol., vol. 25, no. 6, pp. 1038–1050, Jun. 2015, **DOI:** 10.1109/TCSVT.2014.2361419

[15] M. Bilal, ‘‘Resource-efficient FPGA implementation of perspective transformation for bird’s eye view generation using high-level synthesis framework, ’’IET Circuits, Devices Syst., vol. 13, no. 6, pp. 756–762, Sep. 2019, DOI: 10.1049/IET-CDS.2018.5263.

[16] A. Hernandez, A. Gardel, L. Perez, I. Bravo, R. Mateos, and E. Sanchez, ‘‘Real-time image distortion correction using FPGA-based system,’’ in Proc. 32nd Annu. Conf. IEEE Ind. Electron, pp. 7–11, Nov 2006, **DOI:** 10.1109/IECON.2006.347233

[17] B. Maldeniya, D. Nawarathna, K. Wijayasekara, T. Wijegoonasekara,and R. Rodrigo, ‘‘Computationally efficient implementation of video rectification in an FPGA for stereo vision applications,’’ in Proc. 5th Int.Conf. Inf. Autom. Sustainability, pp. 219–224, Dec. 2010, **DOI:** 10.1109/ICIAFS.2010.5715663

[18] Digital Image Processing, 3rd edition Book by Rafael C. Gonzalez and Richard E. Woods.

[19] Collection of 20 (each) left and right images were downloaded from kaggle website. (https://www. kaggle.com/datasets).

[20] For comparing difference between bilinear and bicubic algorithm and the average mean error and performance parameters the datasets (images) are taken from Middlebury website. (https:// vision [1]. middlebury.edu/ stereo/submit3/zip/ MiddE).

.

APPENDIX

ANNEXURE 1 – MATLAB CODE

TOOL USED: MATLAB R2017a

MATLAB CODE - DEMOSAICING

```
I1 = imread("leftImage.jpeg");
I2 = imread("rightImage.jpeg");
load yellowstone_inlier_points;
showMatchedFeatures(I1,I2,inlier_points1,inlier_points2,"montage");
title("Original Images and Matching Feature Points");
f = estimateFundamentalMatrix(inlier_points1,inlier_points2, ...
   "Method","Norm8Point");
[tform1,tform2] = estimateStereoRectification(f,inlier_points1,...
   inlier_points2,size(I2));
[I1Rect,I2Rect] = rectifyStereoImages(I1,I2,tform1,tform2);
figure
imshow(stereoAnaglyph(I1Rect,I2Rect))
```

MATLAB CODE - HOMOGRAPHIC TRANSFORMATION

```
% Read the input image
inputImage = imread('LeftImage.jpeg');  % Replace 'your_image.jpg'
with the actual image file name
% Display the original image
figure;
imshow(inputImage);
title('Original Image');
% Define the source and destination points for the homography
sourcePoints = [0, 0; size(inputImage, 2), 0; size(inputImage, 2),
size(inputImage, 1); 0, size(inputImage, 1)];
destinationPoints = [0, 0; 300, 0; 300, 400; 0, 400];  % Adjust these
points as needed
% Create a homography matrix
H = fitgeotrans(sourcePoints, destinationPoints, 'projective');
% Perform the homographic transform
outputImage = imwarp(inputImage, H);
```

```
% Display the transformed image
figure;
imshow(outputImage);
title('Transformed Image');
% Optionally, save the transformed image
imwrite(outputImage, 'transformed_image.jpg');  % Replace
'transformed_image.jpg' with the desired output file name
```

MATLAB CODE - BLOCKWISE IMPLEMENTATION

```
img = imread('leftImage.jpeg');

% Perform image rectification
[imgRect, newOrigin] = undistortImage(img, cameraParams);

% Perform image undistortion
[imgUndist, newOrigin] = undistortImage(imgRect, cameraParams);

% Perform image interpolation
imgInterp = imresize(imgUndist, 1);

% Display the results
figure;
%subfigure;
subplot(1,4,1);
imshow(img);
title('Original Image');


subplot(1,4,2);
imshow(imgInterp);
title('Rectified Image');

subplot(1,4,3);
imshow(imgUndist);
title('Undistorted  Image');

subplot(1,4,4);
imshow(imgRect);
title('Interpolated Image');
```

MATLAB CODE - INTERPOLATION (BILINEAR AND BICUBIC)

```
% Read the input image
original_image = imread('j.png');

% Define the scale factor
scale_factor = 2;

% Perform bilinear interpolation
bilinear_image = imresize(original_image, scale_factor, 'bilinear');

% Perform bicubic interpolation
bicubic_image = imresize(original_image, scale_factor, 'bicubic');

% Display the original, bilinear, and bicubic images
subplot(1, 3, 1);
imshow(original_image);
title('Original Image');

subplot(1, 3, 2);
imshow(bilinear_image);
title('Bilinear Interpolation');

subplot(1, 3, 3);
imshow(bicubic_image);
title('Bicubic Interpolation');
```

MATLAB CODE- AVERAGE ABSOLUTE ERROR

```
% Read the images
image1 = imread('j.png');
image2 = imread('j2.png');
newWidth = 200; % New width in pixels
newHeight = 150; % New height in pixels
image1 = imresize(image1, [newHeight, newWidth]);
image2 = imresize(image2, [newHeight, newWidth]);n=1;

% Convert images to grayscale if they are not already
if size(image1, 3) == 3
    image1 = rgb2gray(image1);
end

if size(image2, 3) == 3
    image2 = rgb2gray(image2);
end
```

```
% Ensure images have the same size
assert(all(size(image1) == size(image2)), 'Images must have the same
size.');

% Compute the absolute difference between the images
abs_diff = abs(double(image1) - double(image2));

% Calculate the average absolute error
avg_abs_error = mean(abs_diff(:))/n;

disp(['Average Absolute Error: ', num2str(avg_abs_error)]);
```

MATLAB CODE- PSNR CALCULATION

```
% Read the two images
image1 = imread('bi2.png'); % Replace 'image1.jpg' with the filename of
your first image
image2 = imread('ori1.png'); % Replace 'image2.jpg' with the filename of
your second image
newWidth = 200; % New width in pixels
newHeight = 150; % New height in pixels
image1 = imresize(image1, [newHeight, newWidth]);
image2 = imresize(image2, [newHeight, newWidth]);

% Convert images to double precision
image1 = im2double(image1);
image2 = im2double(image2);

% Calculate PSNR
psnrValue = psnr(image1, image2);

% Display PSNR value
disp(['PSNR value between the two images is: ', num2str(psnrValue)]);
```

APPENDIX

ANNEXURE 2 – VERILOG MODULES

TOOL USED: XILINX VIVADO 2018.2

TOP MODULE:

```
`timescale 1ns / 1ps

`default_nettype none

module top
   (   input wire i_top_clk,
       input wire i_top_rst,

       input wire  i_top_cam_start,
       output wire o_top_cam_done,

       // I/O to camera
       input wire       i_top_pclk,
       input wire [7:0] i_top_pix_byte,
       input wire       i_top_pix_vsync,
       input wire       i_top_pix_href,
       output wire      o_top_reset,
       output wire      o_top_pwdn,
       output wire      o_top_xclk,
       output wire      o_top_siod,
       output wire      o_top_sioc,

       // I/O to VGA
       output wire [3:0] o_top_vga_red,
       output wire [3:0] o_top_vga_green,
       output wire [3:0] o_top_vga_blue,
       output wire       o_top_vga_vsync,
       output wire       o_top_vga_hsync
   );

   // Connect cam_top/vga_top modules to BRAM
   wire [11:0] i_bram_pix_data,    o_bram_pix_data;
   wire [18:0] i_bram_pix_addr,    o_bram_pix_addr;
   wire        i_bram_pix_wr;

   // Reset synchronizers for all clock domains
   reg r1_rstn_top_clk,    r2_rstn_top_clk;
```

```
    reg r1_rstn_pclk,       r2_rstn_pclk;
    reg r1_rstn_clk25m,     r2_rstn_clk25m;

    wire w_clk25m;

    // Generate clocks for camera and VGA
    clk_wiz_1
    clock_gen
    (
       .clk_in1(i_top_clk         ),
       .clk_out1(w_clk25m         ),
       .clk_out2(o_top_xclk       )
    );

    wire w_rst_btn_db;

    // Debounce top level button - invert reset to have debounced negedge reset
    localparam DELAY_TOP_TB = 240_000; //240_000 when uploading to hardware, 10 when
simulating in testbench
    debouncer
    #(  .DELAY(DELAY_TOP_TB)    )
    top_btn_db
    (
       .i_clk(i_top_clk       ),
       .i_btn_in(~i_top_rst    ),
       .o_btn_db(w_rst_btn_db  )
    );

    // Double FF for negedge reset synchronization
    always @(posedge i_top_clk or negedge w_rst_btn_db)
       begin
          if(!w_rst_btn_db) {r2_rstn_top_clk, r1_rstn_top_clk} <= 0;
          else              {r2_rstn_top_clk, r1_rstn_top_clk} <= {r1_rstn_top_clk, 1'b1};
       end
    always @(posedge w_clk25m or negedge w_rst_btn_db)
       begin
          if(!w_rst_btn_db) {r2_rstn_clk25m, r1_rstn_clk25m} <= 0;
          else              {r2_rstn_clk25m, r1_rstn_clk25m} <= {r1_rstn_clk25m, 1'b1};
       end
    always @(posedge i_top_pclk or negedge w_rst_btn_db)
       begin
          if(!w_rst_btn_db) {r2_rstn_pclk, r1_rstn_pclk} <= 0;
          else              {r2_rstn_pclk, r1_rstn_pclk} <= {r1_rstn_pclk, 1'b1};
       end

    // FPGA-camera interface
```

```
cam_top
#(  .CAM_CONFIG_CLK(100_000_000)    )
OV7670_cam
(
    .i_clk(i_top_clk              ),
    .i_rstn_clk(r2_rstn_top_clk     ),
    .i_rstn_pclk(r2_rstn_pclk       ),

    // I/O for camera init
    .i_cam_start(i_top_cam_start    ),
    .o_cam_done(o_top_cam_done      ),

    // I/O camera
    .i_pclk(i_top_pclk            ),
    .i_pix_byte(i_top_pix_byte      ),
    .i_vsync(i_top_pix_vsync        ),
    .i_href(i_top_pix_href          ),
    .o_reset(o_top_reset            ),
    .o_pwdn(o_top_pwdn              ),
    .o_siod(o_top_siod              ),
    .o_sioc(o_top_sioc              ),

    // Outputs from camera to BRAM
    .o_pix_wr(                    ),
    .o_pix_data(i_bram_pix_data     ),
    .o_pix_addr(i_bram_pix_addr     )
);

mem_bram
#(  .WIDTH(12                     ),
    .DEPTH(640*480)                 )
 pixel_memory
 (
    // BRAM Write signals (cam_top)
    .i_wclk(i_top_pclk              ),
    .i_wr(1'b1                    ),
    .i_wr_addr(i_bram_pix_addr      ),
    .i_bram_data(i_bram_pix_data    ),
    .i_bram_en(1'b1                 ),

     // BRAM Read signals (vga_top)
    .i_rclk(w_clk25m                ),
    .i_rd(1'b1                    ),
    .i_rd_addr(o_bram_pix_addr      ),

   .o_bram_data(o_bram_pix_data    )
```

```
    );

    wire X;
    wire Y;

    vga_top
    display_interface
    (
        .i_clk25m(w_clk25m            ),
        .i_rstn_clk25m(r2_rstn_clk25m  ),

        // VGA timing signals
        .o_VGA_x(X                    ),
        .o_VGA_y(Y                    ),
        .o_VGA_vsync(o_top_vga_vsync   ),
        .o_VGA_hsync(o_top_vga_hsync   ),
        .o_VGA_video(                 ),

        // VGA RGB Pixel Data
        .o_VGA_red(o_top_vga_red      ),
        .o_VGA_green(o_top_vga_green   ),
        .o_VGA_blue(o_top_vga_blue     ),

        // VGA read/write from/to BRAM
        .i_pix_data(o_bram_pix_data    ),
        .o_pix_addr(o_bram_pix_addr    )
    );
    endmodule
```

module clk_wiz_1:

```
module clk_wiz_1(

    input wire clk_in1,
    output reg clk_out1,
    output reg clk_out2
);
reg [7:0] counter;

always @(posedge clk_in1) begin
    counter <= counter + 1;
    if (counter == 3'd0) begin
        clk_out1 <= ~clk_out1;
    end

    if (counter == 7'd0) begin
```

```
        clk_out2 <= ~clk_out2;
    end
end

endmodule
```

module debouncer:

```
module debouncer
#(parameter DELAY = 1_000_000)
(
    input wire  i_clk,
    input wire  i_btn_in,
    output wire o_btn_db
);

    localparam MAX_COUNT = $clog2(DELAY);
    reg [MAX_COUNT-1:0] counter;
    reg                 r_sample;

    initial { counter, r_sample } = 0;

    always @(posedge i_clk)
        begin
            if(i_btn_in !== r_sample && counter < DELAY)
                counter <= counter + 1'b1;
            else if(counter == DELAY)
                begin
                    counter <= 0;
                    r_sample <= i_btn_in;
                end
            else
                counter <= 0;
        end

assign o_btn_db = r_sample;

endmodule
```

module cam top:

```
module cam_top
#(parameter CAM_CONFIG_CLK = 100_000_000)
    (  input wire         i_clk,
       input wire         i_rstn_clk,
       input wire         i_rstn_pclk,

       // Start/Done signals for cam init
       input wire         i_cam_start,
       output wire         o_cam_done,

       // I/O camera
       input wire         i_pclk,
       input wire [7:0]    i_pix_byte,
       input wire         i_vsync,
       input wire         i_href,
       output wire         o_reset,
       output wire         o_pwdn,
       output wire         o_siod,
       output wire         o_sioc,

       // Outputs to BRAM
       output wire         o_pix_wr,
       output wire [11:0]  o_pix_data,
       output wire [18:0]  o_pix_addr
    );

    assign o_reset = 1;       // 0: reset registers   1: normal mode
    assign o_pwdn  = 0;       // 0: normal mode       1: Power down mode

    wire       w_start_db;

    debouncer
    #(  .DELAY(240_000)         )
    cam_btn_start_db
    (   .i_clk(i_clk           ),
       .i_btn_in(i_cam_start   ),

       // Debounced button to start cam init
       .o_btn_db(w_start_db    )
    );




    cam_init
    #(  .CLK_F(CAM_CONFIG_CLK       ),
```

```
        .SCCB_F(400_000)          )
    configure_cam
    (   .i_clk(i_clk              ),
        .i_rstn(i_rstn_clk        ),

        // Start/Done signals for cam init
        .i_cam_init_start(w_start_db),
        .o_cam_init_done(o_cam_done ),

        // SCCB lines
        .o_siod(o_siod            ),
        .o_sioc(o_sioc            ),

        // Signals used for testbench
        .o_data_sent_done(        ),
        .o_SCCB_dout(             )
    );

    cam_capture
    cam_pixels
    (   // Cam VGA frame timing signals
        .i_pclk(i_pclk        ),
        .i_vsync(i_vsync      ),
        .i_href(i_href        ),

        // Poll for when the cam is done init
        .i_cam_done(o_cam_done ),

        .i_D(i_pix_byte       ),
        .o_pix_addr(o_pix_addr ),
        .o_wr(o_pix_wr        ),
        .o_pix_data(o_pix_data )
    );

  endmodule
```

module debouncer:

```
module debouncer
#(parameter DELAY = 1_000_000)
(
   input wire  i_clk,
   input wire  i_btn_in,
   output wire o_btn_db
);
```

```
    localparam MAX_COUNT = $clog2(DELAY);
    reg [MAX_COUNT-1:0] counter;
    reg                 r_sample;

    initial { counter, r_sample } = 0;

    always @(posedge i_clk)
        begin
            if(i_btn_in !== r_sample && counter < DELAY)
                counter <= counter + 1'b1;
            else if(counter == DELAY)
                begin
                    counter <= 0;
                    r_sample <= i_btn_in;
                end
            else
                counter <= 0;
        end

assign o_btn_db = r_sample;

endmodule
```

module cam_init:

```
    module cam_init
    #(parameter CLK_F = 100_000_000,
        parameter SCCB_F = 400_000)
        (   input wire      i_clk,
            input wire      i_rstn,
            input wire      i_cam_init_start,
            output wire     o_siod,
            output wire     o_sioc,
            output wire     o_cam_init_done,

            // Signal used only for testbench
            output wire         o_data_sent_done,
            output wire [7:0]   o_SCCB_dout
        );

        wire [7:0]  w_cam_rom_addr;
        wire [15:0] w_cam_rom_data;
        wire [7:0]  w_send_addr,    w_send_data;
        wire        w_start_sccb,   w_ready_sccb;
```

```
cam_rom
   OV7670_Registers
   (   .i_clk(i_clk          ),
       .i_rstn(i_rstn        ),

       .i_addr(w_cam_rom_addr  ),
       .o_dout(w_cam_rom_data  )
   );

   cam_config
   #(  .CLK_F(CLK_F)               )
   OV7670_config
   (   .i_clk(i_clk              ),
       .i_rstn(i_rstn            ),

        // Ready/Start signals for SCCB: Poll for ready signal to start sending cam ROM data
       .i_i2c_ready(w_ready_sccb      ),
       .o_i2c_start(w_start_sccb      ),

       // Start/Done signals for cam init
       .i_config_start(i_cam_init_start),
       .o_config_done(o_cam_init_done  ),

       // Read through cam ROM
       .i_rom_data(w_cam_rom_data     ),
       .o_rom_addr(w_cam_rom_addr     ),
       .o_i2c_addr(w_send_addr        ),
       .o_i2c_data(w_send_data        )
   );

   sccb_master
   #(  .CLK_F(CLK_F),
       .SCCB_F(SCCB_F)       )
   SCCB_HERE
   (   .i_clk(i_clk          ),
       .i_rstn(i_rstn        ),

       // SCCB control signals
       .i_read(1'b0          ),
       .i_write(1'b1         ),
       .i_start(w_start_sccb   ),
       .i_restart(1'b0       ),
       .i_stop(1'b0          ),
       .o_ready(w_ready_sccb   ),

       // SCCB addr/data signals
```

```
        .i_din(w_send_data      ),
        .i_addr(w_send_addr     ),

        // Slave->Master com signals
        .o_dout(o_SCCB_dout     ),
        .o_done(o_data_sent_done),
        .o_ack(                 ),

        // SCCB Lines
        .io_sda(o_siod          ),
        .o_scl(o_sioc           )
    );

endmodule
```

module cam_rom:

```
module cam_rom
    (   input wire       i_clk,
        input wire       i_rstn,
        input wire  [7:0] i_addr,
        output reg [15:0] o_dout
    );

    // Registers for OV7670 for configuration of RGB 444
    always @(posedge i_clk or negedge i_rstn) begin
        if(!i_rstn) o_dout <= 0;
        else begin
            case(i_addr)
            0:  o_dout <= 16'h12_80;  // COM7:       Reset SCCB registers
            1:  o_dout <= 16'hFF_F0;  // Delay
            2:  o_dout <= 16'h12_04;  // COM7,       Set RGB color output
            3:  o_dout <= 16'h11_00;  // CLKRC       Internal PLL matches input clock (24 MHz).
            4:  o_dout <= 16'h0C_00;  // COM3,       *Leave as default.
            5:  o_dout <= 16'h3E_00;  // COM14,      *Leave as default. No scaling, normal pclock
            6:  o_dout <= 16'h04_00;  // COM1,       *Leave as default. Disable CCIR656
            7:  o_dout <= 16'h8C_02;  // RGB444      Enable RGB444 mode with xR GB.
            8:  o_dout <= 16'h40_D0;  // COM15,      Output full range for RGB 444.
            9:  o_dout <= 16'h3a_04;  // TSLB        set correct output data sequence (magic)
            10: o_dout <= 16'h14_18;  // COM9        MAX AGC value x4
            11: o_dout <= 16'h4F_B3;  // MTX1        all of these are magical matrix coefficients
            12: o_dout <= 16'h50_B3;  // MTX2
            13: o_dout <= 16'h51_00;  // MTX3
            14: o_dout <= 16'h52_3d;  // MTX4
            15: o_dout <= 16'h53_A7;  // MTX5
            16: o_dout <= 16'h54_E4;  // MTX6
```

```
        17: o_dout <= 16'h58_9E;  // MTXS
        18: o_dout <= 16'h3D_C0;  // COM13       sets gamma enable, does not preserve reserved bits,
may be wrong?
        19: o_dout <= 16'h17_14;  // HSTART      start high 8 bits
        20: o_dout <= 16'h18_02;  // HSTOP       stop high 8 bits //these kill the odd colored line
        21: o_dout <= 16'h32_80;  // HREF        edge offset
        22: o_dout <= 16'h19_03;  // VSTART      start high 8 bits
        23: o_dout <= 16'h1A_7B;  // VSTOP       stop high 8 bits
        24: o_dout <= 16'h03_0A;  // VREF        vsync edge offset
        25: o_dout <= 16'h0F_41;  // COM6        reset timings
        26: o_dout <= 16'h1E_00;  // MVFP        disable mirror / flip //might have magic value of 03
        27: o_dout <= 16'h33_0B;  // CHLF        magic value from the internet
        28: o_dout <= 16'h3C_78;  // COM12       no HREF when VSYNC low
        29: o_dout <= 16'h69_00;  // GFIX        fix gain control
        30: o_dout <= 16'h74_00;  // REG74       Digital gain control
        31: o_dout <= 16'hB0_84;  // RSVD        magic value from the internet *required* for good
color
        32: o_dout <= 16'hB1_0c;  // ABLC1
        33: o_dout <= 16'hB2_0e;  // RSVD        more magic internet values
        34: o_dout <= 16'hB3_80;  // THL_ST
        //begin mystery scaling numbers
        35: o_dout <= 16'h70_3a;  // SCALING_XSC         *Leave as default. No test pattern output.
        36: o_dout <= 16'h71_35;  // SCALING_YSC         *Leave as default. No test pattern output.
        37: o_dout <= 16'h72_11;  // SCALING DCWCTR      *Leave as default. Vertical down sample
by 2. Horizontal down sample by 2.
        38: o_dout <= 16'h73_f0;  // SCALING PCLK_DIV
        39: o_dout <= 16'ha2_02;  // SCALING PCLK DELAY   *Leave as deafult.
        //gamma curve values
        40: o_dout <= 16'h7a_20;  // SLOP
        41: o_dout <= 16'h7b_10;  // GAM1
        42: o_dout <= 16'h7c_1e;  // GAM2
        43: o_dout <= 16'h7d_35;  // GAM3
        44: o_dout <= 16'h7e_5a;  // GAM4
        45: o_dout <= 16'h7f_69;  // GAM5
        46: o_dout <= 16'h80_76;  // GAM6
        47: o_dout <= 16'h81_80;  // GAM7
        48: o_dout <= 16'h82_88;  // GAM8
        49: o_dout <= 16'h83_8f;  // GAM9
        50: o_dout <= 16'h84_96;  // GAM10
        51: o_dout <= 16'h85_a3;  // GAM11
        52: o_dout <= 16'h86_af;  // GAM12
        53: o_dout <= 16'h87_c4;  // GAM13
        54: o_dout <= 16'h88_d7;  // GAM14
        55: o_dout <= 16'h89_e8;  // GAM15
        //AGC and AEC
        56: o_dout <= 16'h13_e0;  // COM8     disable AGC / AEC
```

```
            57: o_dout <= 16'h00_00;  // set gain reg to 0 for AGC
            58: o_dout <= 16'h10_00;  // set ARCJ reg to 0
            59: o_dout <= 16'h0d_40;  // magic reserved bit for COM4
            60: o_dout <= 16'h14_18;  // COM9, 4x gain + magic bit
            61: o_dout <= 16'ha5_05;  // BD50MAX
            62: o_dout <= 16'hab_07;  // DB60MAX
            63: o_dout <= 16'h24_95;  // AGC upper limit
            64: o_dout <= 16'h25_33;  // AGC lower limit
            65: o_dout <= 16'h26_e3;  // AGC/AEC fast mode op region
            66: o_dout <= 16'h9f_78;  // HAECC1
            67: o_dout <= 16'ha0_68;  // HAECC2
            68: o_dout <= 16'ha1_03;  // magic
            69: o_dout <= 16'ha6_d8;  // HAECC3
            70: o_dout <= 16'ha7_d8;  // HAECC4
            71: o_dout <= 16'ha8_f0;  // HAECC5
            72: o_dout <= 16'ha9_90;  // HAECC6
            73: o_dout <= 16'haa_94;  // HAECC7
            74: o_dout <= 16'h13_a7;  // COM8, enable AGC / AEC
            75: o_dout <= 16'h69_06;
            default: o_dout <= 16'hFF_FF;        //mark end of ROM
            endcase
        end
    end
endmodule
```

module cam_config:

```
module cam_config
#(parameter CLK_F = 100_000_000)
    (   input wire         i_clk,
        input wire         i_rstn,
        input wire         i_i2c_ready,
        input wire         i_config_start,
        input wire [15:0]   i_rom_data,
        output reg [7:0]    o_rom_addr,
        output reg         o_i2c_start,
        output reg [7:0]    o_i2c_addr,
        output reg [7:0]    o_i2c_data,
        output reg         o_config_done
    );

    localparam ten_ms_delay  = (CLK_F * 10) / 1000;
    localparam timer_size    =  $clog2(ten_ms_delay);
    reg [timer_size - 1: 0] timer;

    localparam SM_IDLE  = 0;
```

```
localparam SM_SEND  = 1;
localparam SM_DONE  = 2;
localparam SM_TIMER = 3;

reg [2:0] SM_state;
reg [2:0] SM_return_state;
reg [1:0] byte_index;

always @(posedge i_clk or negedge i_rstn)
   begin
      if(!i_rstn) begin
            o_config_done <= 0;
            byte_index    <= 0;
            o_rom_addr    <= 0;
            o_i2c_addr    <= 0;
            o_i2c_start   <= 0;
            o_i2c_data    <= 0;
            SM_state      <= SM_IDLE;
         end
      else begin
         case(SM_state)
            SM_IDLE:
               begin
                  SM_state <= (i_config_start) ? SM_SEND : SM_IDLE;
               end
            SM_SEND:
               begin
                  if(i_i2c_ready)
                     case(i_rom_data)
                     16'hFF_FF: SM_state <= SM_DONE;
                     16'hFF_F0: begin
                                   SM_state         <= SM_TIMER;
                                   SM_return_state  <= SM_SEND;
                                   timer            <= ten_ms_delay;
                                   o_rom_addr       <= o_rom_addr + 1;
                                end
                     default:
                           begin
                              SM_state       <= SM_TIMER;
                              SM_return_state <= SM_SEND;
                              timer          <= 1;
                              o_i2c_start    <= 1;
                              o_i2c_addr     <= i_rom_data[15:8];
                              o_i2c_data     <= i_rom_data[7:0];
                              o_rom_addr     <= o_rom_addr + 1;
                           end
```

```
                endcase
            end
          SM_DONE:
            begin
              SM_state       <= SM_IDLE;
              o_config_done   <= 1;
            end
          SM_TIMER:
            begin
              SM_state    <= (timer == 1) ? SM_return_state : SM_TIMER;
              timer       <= (timer == 1) ?        0        : timer - 1;
              o_i2c_start <= 0;
            end
        endcase
      end
    end

endmodule
```

module sccb_master:

```
module sccb_master
#(parameter CLK_F = 100_000_000,
  parameter SCCB_F = 400_000)
  (  input wire         i_clk,
     input wire         i_rstn,

     input wire         i_read,      // I2C Commands. Assume read/write are mutually exclusive
     input wire         i_write,
     input wire         i_start,
     input wire         i_restart,
     input wire         i_stop,

     input wire   [7:0]  i_din,
     input wire   [7:0]  i_addr,

     output wire  [7:0]  o_dout,
     output wire         o_ready,
     output wire         o_done,      // 1-cycle tick when a transaction is completed
     output wire         o_ack,

     inout  wire         io_sda,
     output wire         o_scl
   );

  // Camera Address for writing
```

```
    localparam CAM_ADDR = 7'h21;

    // FSM States
    localparam [3:0] IDLE      = 0,
                     START_1   = 1,
                     START_2   = 2,
                     WAIT      = 3,
                     DATA_1    = 4,
                     DATA_2    = 5,
                     DATA_3    = 6,
                     DATA_4    = 7,
                     DATA_DONE = 8,
                     RESTART   = 9,
                     END_1     = 10,
                     END_2     = 11;

    // CLK_F/SCCB_F is number of clocks in ONE period of the SCCB clock (SCL)
    localparam TIMER_WIDTH = $clog2(CLK_F/SCCB_F);
    localparam HALF        = CLK_F/(2*SCCB_F);
    localparam QUARTER     = HALF/2;

    reg [TIMER_WIDTH - 1: 0] timer;
    reg [3:0]                state;
    reg [8:0]                r_data_bit_index;
    reg [8:0]                r_tx;
    reg [8:0]                r_rx;
    reg [7:0]                r_latched_data, r_latched_addr;
    reg [1:0]                r_byte_index;
    reg                     data_state;
    wire                     i_sda;

    reg r_done;
    reg r_ready;

    // Buffer SCL and SDA lines
    reg r_scl, r2_scl;
    reg r_sda, r2_en_sda;

    // Register read/write inputs
    reg r_read;
    reg r_write;

   initial begin
       state       = IDLE;
       r_ready     = 1'b1;
       r_scl       = 1'b1;
```

```
    r_sda       = 1'b1;
    r2_scl      = 1'b1;
    r2_en_sda   = 1'b1;
    r_done      = 1'b0;
end

always @(posedge i_clk or negedge i_rstn)
    begin
        if(!i_rstn) begin
            r2_scl    <= 1'b1;
            r2_en_sda <= 1'b1;
        end
        else begin
            r2_scl    <= r_scl;
            r2_en_sda <= r_sda;
        end
    end

always @(posedge i_clk)
    begin
        r_read  <= i_read;
        r_write <= i_write;
    end

// 2-Wire SCCB bus lines
assign i_sda  = (data_state && r_read) || (data_state && r_write && r_data_bit_index == 8);
assign o_scl  = (r2_scl)            ? 1'bZ : 1'b0;
assign io_sda = (i_sda || r2_en_sda) ? 1'bZ : 1'b0;


// State Machine
always @(posedge i_clk)
    begin
        timer <= timer + 1'b1;              // Free Running Counter
        case(state)
            IDLE: begin                     // SDA and SCL line high; Wait for start command
                timer          <= 0;
                r_ready        <= 1'b1;
                r_done         <= 1'b0;
                data_state     <= 1'b0;
                r_data_bit_index <= 9'b0;
                r_byte_index   <= 2'b0;
                r_scl          <= 1'b1;
                r_sda          <= 1'b1;
                r_latched_data <= 8'hZZ;
                r_latched_addr <= 8'hZZ;
```

```
            if(i_start) begin
                state      <= START_1;
                timer      <= 0;
                r_latched_data <= i_din;
                r_latched_addr <= i_addr;
                r_ready    <= 1'b0;
              end
            end
          START_1: begin    // Bring SDA line low; Wait for 1/2 period of SCL
            r_sda       <= 1'b0;
            if(timer == (HALF-1)) begin
              timer <= 0;
              state <= START_2;
            end
          end
          START_2: begin    // Bring SCL line low; Wait for 1/2 period of SCL
            r_scl   <= 1'b0;
            if(timer == (HALF-1)) begin
              timer       <= 0;
              state       <= WAIT;
            end
          end
          WAIT:   begin     // Both SCL/SDA low; Wait for Control Signal (Read or Write)
            r_scl           <= 1'b0;
            r_sda           <= 1'b0;
            timer           <= 0;
            r_data_bit_index <= 0;
            r_byte_index     <=  r_byte_index + 1'b1;
            state           <= (r_byte_index == 3) ? END_1 : DATA_1;
            case(r_byte_index)                          // 3-Phase Write Cycle (no ack in SCCB)
              2'b00: r_tx <= {CAM_ADDR, ~i_write, 1'b1};      //   byte1 = {SLAVE ADDRESS, WR
BIT, Don't Care Bit}
              2'b01: r_tx <= {r_latched_addr, 1'b1};          //   byte2 = {Register Addr,        Don't Care
Bit}
              2'b10: r_tx <= {r_latched_data, 1'b1};          //   byte3 = {Data to Register,     Don't Care
Bit}
              default: r_tx <= {r_latched_data, 1'b1};
            endcase

            if((!i_write) && (!i_read)) begin
              if(i_stop)                    state <= END_1;
              else if(i_restart || i_start)   state <= RESTART;
              else                          state <= IDLE;
            end
          end
```

```
DATA_1: begin   // Load Data Bit to SDA before sampled by SCL
   r_sda      <= r_tx[8];
   r_scl      <= 1'b0;
   data_state  <= 1'b1;
   if(timer == (QUARTER-1)) begin
      timer <= 0;
      state <= DATA_2;
   end
end
DATA_2: begin   // SCL Samples the Data Bit (Shift in for read/Shift out for write)
   r_sda <= r_tx[8];
   r_scl <= 1'b1;
   // Sample read bits in the middle of SCL being HIGH for sample reads
   if(timer == (QUARTER-1)) begin
      timer <= 0;
      state <= DATA_3;
      r_rx  <= {r_rx[7:0], io_sda};
   end
end
DATA_3: begin   // Wait another quarter SCL cycle of it being HIGH
   r_sda <= r_tx[8];
   r_scl <= 1'b1;
   if(timer == (QUARTER-1)) begin
      timer <= 0;
      state <= DATA_4;  // Shift Data In
   end
end
DATA_4: begin   // Bring SCL Low again; Wait another quarter of a cycle
   r_sda      <= r_tx[8];
   r_scl      <= 1'b0;
   if(timer == (QUARTER-1)) begin
      timer <= 0;
      if(r_data_bit_index == 8) begin
         state     <= DATA_DONE;
         r_done    <= 1'b1;    // Set done signal HIGH
         data_state <= 1'b0;
      end
      else begin
         r_tx           <=  {r_tx[7:0], 1'b0};
         r_data_bit_index <=  r_data_bit_index + 1'b1;
         state          <=  DATA_1;
      end
   end
end
DATA_DONE: begin
   r_done <= 1'b0;    // Set done signal LOW since it's a tick
```

```
                r_sda  <= 1'b0;
                r_scl  <= 1'b0;
                if(timer == (QUARTER-1)) begin
                    timer <= 0;
                    state <= WAIT;
                end
            end
            RESTART: begin
                if(timer == (HALF-1)) begin
                    timer <= 0;
                    state <= START_1;
                end
            end
            END_1: begin
                // SCL low, SDA low,        SCL high, SDA high
                // [ Done in WAIT state]
                r_scl <= 1'b1;
                r_sda <= 1'b0;
                if(timer == (HALF-1)) begin
                    timer <= 0;
                    state <= END_2;
                end
            end
            END_2: begin
                r_scl <= 1'b1;
                r_sda <= 1'b1;
                if(timer == (HALF-1)) begin
                    timer <= 0;
                    state <= IDLE;
                end
            end
        endcase
    end

    // Assign Output from SCCBA Reads ( don't read ACK bit )
    assign o_dout = r_rx[8:1];
    // ACK (from slave in writes should be '0')
    assign o_ack = r_rx[0];

    // Assign I2C Master Status Signals
    assign o_ready = r_ready;
    assign o_done  = r_done;

endmodule
```

module cam_capture:

```
module cam_capture
   (   input wire        i_pclk,
       input wire        i_vsync,
       input wire        i_href,
       input wire  [7:0]  i_D,
       input wire        i_cam_done,
       output reg  [18:0] o_pix_addr,
       output reg  [11:0] o_pix_data,
       output reg        o_wr
   );

   // Negative/Positive Edge Detection of vsync for frame start/frame done signal
   reg        r1_vsync,    r2_vsync;
   wire       frame_start, frame_done;

   initial { r1_vsync, r2_vsync } = 0;
   always @(posedge i_pclk)
         {r2_vsync, r1_vsync} <= {r1_vsync, i_vsync};

   assign frame_start = (r1_vsync == 0) && (r2_vsync == 1);    // Negative Edge of vsync
   assign frame_done  = (r1_vsync == 1) && (r2_vsync == 0);    // Positive Edge of vsync

   // FSM for capturing pixel data in pclk domain
   localparam [1:0] WAIT   = 2'd0,
                    IDLE   = 2'd1,
                    CAPTURE = 2'd2;

   reg       r_half_data;
   reg [1:0]  SM_state;
   reg [3:0]  pixel_data;

   always @(posedge i_pclk)
      begin
         r_half_data        <= 0;
         o_wr              <= 0;
         o_pix_data         <= o_pix_data;
         o_pix_addr         <= o_pix_addr;
         SM_state           <= WAIT;
         case(SM_state)
            WAIT:
               begin
                  // Skip the first two frames on start-up
                  SM_state    <= (frame_start && i_cam_done) ? IDLE : WAIT;
               end
```

```
            IDLE:
               begin
                  SM_state   <= (frame_start) ? CAPTURE : IDLE;
                  o_pix_addr <= 0;
                  o_pix_data <= 0;
               end
            CAPTURE:
               begin
                  SM_state   <= (frame_done) ? IDLE : CAPTURE;
                  o_pix_addr <= (r_half_data) ? o_pix_addr + 1'b1 : o_pix_addr;
                  if(i_href)
                     begin
                        // Register first byte
                        if(!r_half_data)
                           pixel_data <= i_D[3:0];
                        r_half_data    <= ~r_half_data;
                        o_wr           <= (r_half_data) ? 1'b1 : 1'b0;
                        o_pix_data     <= (r_half_data) ? {pixel_data, i_D} : o_pix_data;
                     end
               end
         endcase
      end

endmodule
```

module mem_bram:

```
module mem_bram
#(parameter WIDTH = 11,
   parameter DEPTH = 640*480)
   (   input wire                  i_wclk,
      input wire                  i_wr,
      input wire [$clog2(DEPTH)-1:0]  i_wr_addr,

      input wire                  i_rclk,
      input wire                  i_rd,
      input wire [$clog2(DEPTH)-1:0]  i_rd_addr,

      input wire                  i_bram_en,
      input wire [WIDTH-1:0]         i_bram_data,
      output reg [WIDTH-1:0]         o_bram_data
   );

   // Infer dual-port BRAM with dual clocks
   // https://docs.xilinx.com/v/u/2019.2-English/ug901-vivado-synthesis (page 126)
   reg [WIDTH-1:0] ram [0:DEPTH-1];
```

```
    always @(posedge i_wclk)
    if(i_bram_en)
       if(i_wr)
          ram[i_wr_addr] <= i_bram_data;

    always @(posedge i_rclk)
    if(i_rd)
       o_bram_data <= ram[i_rd_addr];

  endmodule
```

module vga_top:

```
  module vga_top
    (   input wire         i_clk25m,
       input wire         i_rstn_clk25m,

       // VGA driver signals
       output wire [9:0]   o_VGA_x,
       output wire [9:0]   o_VGA_y,
       output wire         o_VGA_vsync,
       output wire         o_VGA_hsync,
       output wire         o_VGA_video,
       output wire [3:0]   o_VGA_red,
       output wire [3:0]   o_VGA_green,
       output wire [3:0]   o_VGA_blue,

       // VGA read from BRAM
       input  wire [11:0] i_pix_data,
       output reg  [18:0] o_pix_addr
    );

    vga_driver
    #(  .hDisp(640),
       .hFp(16),
       .hPulse(96),
       .hBp(48),
       .vDisp(480),
       .vFp(10),
       .vPulse(2),
       .vBp(33)              )
    vga_timing_signals
    (   .i_clk(i_clk25m        ),
       .i_rstn(i_rstn_clk25m   ),
```

```
    // VGA timing signals
    .o_x_counter(o_VGA_x    ),
    .o_y_counter(o_VGA_y    ),
    .o_video(o_VGA_video    ),
    .o_vsync(o_VGA_vsync    ),
    .o_hsync(o_VGA_hsync    )
);

reg [3:0]   r_VGA_R;
reg [3:0]   r_VGA_G;
reg [3:0]   r_VGA_B;
reg [1:0]   r_SM_state;
localparam [1:0]    WAIT_1  = 0,
                    WAIT_2  = 'd1,
                    READ    = 'd2;

always @(posedge i_clk25m or negedge i_rstn_clk25m)
if(!i_rstn_clk25m)
begin
    r_SM_state <= WAIT_1;
    o_pix_addr <= 0;
end
else
    case(r_SM_state)
    // Skip two frames
    WAIT_1: r_SM_state <= (o_VGA_x == 640 && o_VGA_y == 480) ? WAIT_2 : WAIT_1;
    WAIT_2: r_SM_state <= (o_VGA_x == 640 && o_VGA_y == 480) ? READ : WAIT_2;
    READ: begin
        // Currently active video
        if((o_VGA_y < 480) && (o_VGA_x < 639))
            o_pix_addr <= (o_pix_addr == 307199) ? 0 : o_pix_addr + 1'b1;
        else begin
        // Next clock is active video
        if( (o_VGA_x == 799) && ( (o_VGA_y == 524) || (o_VGA_y < 480) ) )
            o_pix_addr <= o_pix_addr + 1'b1;
        // Next clock not active video
        else if(o_VGA_y >= 480)
            o_pix_addr <= 0;
        end
    end
    endcase

// Valid Video selects between a black RGB Pixel and BRAM pixel data
always @(*)
    begin
        if(o_VGA_video)
```

```
            begin
               r_VGA_R = i_pix_data[11:8];
               r_VGA_G = i_pix_data[7:4];
               r_VGA_B = i_pix_data[3:0];
            end
         else begin
               r_VGA_R = 0;
               r_VGA_G = 0;
               r_VGA_B = 0;
         end
      end

   assign o_VGA_red    = r_VGA_R;
   assign o_VGA_green  = r_VGA_G;
   assign o_VGA_blue   = r_VGA_B;

endmodule
```

module vga_driver:

```
module vga_driver
#(parameter hDisp  = 640,
   parameter hFp    = 16,
   parameter hPulse = 96,
   parameter hBp    = 48,
   parameter vDisp  = 480,
   parameter vFp    = 10,
   parameter vPulse = 2,
   parameter vBp    = 33)
   (   input wire         i_clk,
      input wire         i_rstn,
      output wire [9:0]   o_x_counter,
      output wire [9:0]   o_y_counter,
      output wire        o_video,
      output wire        o_hsync,
      output wire        o_vsync
   );

    // Horizonal timing     hEND = 800
    localparam hEND       = hDisp + hFp + hPulse + hBp;
    localparam hSyncStart  = hDisp + hFp;
    localparam hSyncEnd    = hDisp + hFp + hPulse;

    // Vertical timing      vEND = 524
    localparam vEND       = vDisp + vFp + vPulse + vBp;
    localparam vSyncStart  = vDisp + vFp;
```

```
    localparam vSyncEnd    = vDisp + vFp + vPulse;

    reg [9:0] hc;
    reg [9:0] vc;

    always@(posedge i_clk or negedge i_rstn)
      begin
        if(!i_rstn) begin
           hc      <= 0;
           vc      <= 0;
        end
        else begin
           if(hc == hEND-1)
           begin
              hc <= 0;
              if(vc == vEND-1)
              vc <= 0;
              else
                 vc <= vc + 1'b1;
           end
           else
              hc <= hc + 1'b1;
        end
      end

    // Output (x,y) coordinates of the pixel and timing signals
    assign o_x_counter = hc;
    assign o_y_counter = vc;
    assign o_video     = ((hc >= 0) && (hc < hDisp) && (vc >= 0)  && (vc < vDisp));
    assign o_hsync     = ~((hc >= hSyncStart) && (hc < hSyncEnd));
    assign o_vsync     = ~((vc >= vSyncStart) && (vc < vSyncEnd));

  endmodule
```

CONSTRAINT_FILE:

```
  set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { i_top_clk }];
  #IO_L12P_T1_MRCC_35 Sch=clk100mhz
  create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports { i_top_clk }];
  set_property -dict {PACKAGE_PIN H17 IOSTANDARD LVCMOS33} [get_ports { o_top_cam_done
  }]
  set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { i_top_rst }];
  #IO_L4N_T0_D05_14 Sch=btnu
  set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports {
  i_top_cam_start }];
  set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 } [get_ports { o_top_pwdn
```

```
}]; #IO_L20N_T3_A19_15 Sch=ja[1]
set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports {
i_top_pix_byte[0] }]; #IO_L21N_T3_DQS_A18_15 Sch=ja[2]
set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports {
i_top_pix_byte[2] }]; #IO_L21P_T3_DQS_15 Sch=ja[3]
set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports {
i_top_pix_byte[4] }]; #IO_L18N_T2_A23_15 Sch=ja[4]
set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports { o_top_reset }];
#IO_L16N_T2_A27_15 Sch=ja[7]
set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports {
i_top_pix_byte[1] }]; #IO_L16P_T2_A28_15 Sch=ja[8]
set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports {
i_top_pix_byte[3] }]; #IO_L22N_T3_A16_15 Sch=ja[9]
set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports {
i_top_pix_byte[5] }];
set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 } [get_ports {
i_top_pix_byte[6] }]; #IO_L1P_T0_AD0P_15 Sch=jb[1]
set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports { o_top_xclk }];
#IO_L14N_T2_SRCC_15 Sch=jb[2]
set_property -dict { PACKAGE_PIN G16   IOSTANDARD LVCMOS33 } [get_ports { i_top_pix_href
}]; #IO_L13N_T2_MRCC_15 Sch=jb[3]
set_property -dict { PACKAGE_PIN H14   IOSTANDARD LVCMOS33 } [get_ports { o_top_siod }];
#IO_L15P_T2_DQS_15 Sch=jb[4]
set_property -dict { PACKAGE_PIN E16   IOSTANDARD LVCMOS33 } [get_ports {
i_top_pix_byte[7] }]; #IO_L11N_T1_SRCC_15 Sch=jb[7]
set_property -dict { PACKAGE_PIN F13   IOSTANDARD LVCMOS33 } [get_ports {
i_top_pix_vsync }]; #IO_L5P_T0_AD9P_15 Sch=jb[8]
set_property -dict { PACKAGE_PIN G13   IOSTANDARD LVCMOS33 } [get_ports { o_top_sioc }];
#IO_0_15 Sch=jb[9]
set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports { i_top_pclk }];
set_property -dict { PACKAGE_PIN A3    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_red[0] }]; #IO_L8N_T1_AD14N_35 Sch=vga_r[0]
set_property -dict { PACKAGE_PIN B4    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_red[1] }]; #IO_L7N_T1_AD6N_35 Sch=vga_r[1]
set_property -dict { PACKAGE_PIN C5    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_red[2] }]; #IO_L1N_T0_AD4N_35 Sch=vga_r[2]
set_property -dict { PACKAGE_PIN A4    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_red[3] }]; #IO_L8P_T1_AD14P_35 Sch=vga_r[3]

set_property -dict { PACKAGE_PIN C6    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_green[0] }]; #IO_L1P_T0_AD4P_35 Sch=vga_g[0]
set_property -dict { PACKAGE_PIN A5    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_green[1] }]; #IO_L3N_T0_DQS_AD5N_35 Sch=vga_g[1]
set_property -dict { PACKAGE_PIN B6    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_green[2] }]; #IO_L2N_T0_AD12N_35 Sch=vga_g[2]
set_property -dict { PACKAGE_PIN A6    IOSTANDARD LVCMOS33 } [get_ports {
```

```
o_top_vga_green[3] }]; #IO_L3P_T0_DQS_AD5P_35 Sch=vga_g[3]

set_property -dict { PACKAGE_PIN B7    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_blue[0] }]; #IO_L2P_T0_AD12P_35 Sch=vga_b[0]
set_property -dict { PACKAGE_PIN C7    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_blue[1] }]; #IO_L4N_T0_35 Sch=vga_b[1]
set_property -dict { PACKAGE_PIN D7    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_blue[2] }]; #IO_L6N_T0_VREF_35 Sch=vga_b[2]
set_property -dict { PACKAGE_PIN D8    IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_blue[3] }]; #IO_L4P_T0_35 Sch=vga_b[3]

set_property -dict { PACKAGE_PIN B11   IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_hsync }]; #IO_L4P_T0_15 Sch=vga_hs
set_property -dict { PACKAGE_PIN B12   IOSTANDARD LVCMOS33 } [get_ports {
o_top_vga_vsync }]; #IO_L3N_T0_DQS_AD1N_15 Sch=vga_vs
```