UPPSALA
UNIVERSITET

MASTER THESIS SPECIFICATION

Modeling, Simulation, and Injection of Camera Images/Video to Automotive Embedded ECUs

Image Injection Solution for Hardware-in-the-Loop Testing

*Student:*Nidhi Chakrabhavi Basavaraju
Supervisor: Rubén Calvillo Portales

Reviewer: Bengt Jonsson
Examiner: Pontus Ekberg

Department of Information Technology

Abstract

Early in the development of new cars with advanced control units, it is critical to make sure every part integrates flawlessly. One effective method in this approach is hardware-in-the-loop (HIL) testing. We can effectively simulate and interact with real-world dynamics by combining a Real-Time Simulation Platform, a Sensor Simulation PC, an Interface Unit (IU), and a Vehicle Computing Unit (VCU) into a closed-loop system. By identifying problems early in the development process through iterative testing, verification, and validation, we not only save time and costs but also improve the dependability of the vehicle's safety measures, paving the way for a safer and more efficient production process.

This project, put forth by Volvo Cars, investigates the design and development of a proof-of-concept simulation framework - Image Injection Solution (IIS), intended to insert camera images and videos on the IU of the HIL testing system; in other words, it replaces the testing setup's existing camera system with a simulation environment that offers greater customization. This will be essential for improving modular control over testing the Autonomous Driving Systems (ADS) and Advanced Driver Assistance Systems (ADAS) in cars of the future.

The Image Injection Solution is a computation-intensive solution requiring either a GPU or FPGA for increased performance. The IU at Volvo Cars employs an MPSoC/FPGA architecture, predominantly using Xilinx hardware and software, hence this research follows the same, to keep the hardware changes minimal. The project is segmented into three interconnected parts: input, image processing, and output, carried out by three students respectively.

This thesis will focus on the image processing part where the objective is to translate the incoming 8-bit sRGB data to a 12-bit RGGB Raw image. This is done by reversing the conventional Image Sensor Processing (ISP) pipeline to obtain raw image data from processed inputs. This reversal is essential to emulate the raw sensor outputs required by the VCU.

Concerning this, a thorough literature review and analysis of the complete camera image-generating process have been conducted to examine various strategies for obtaining the unprocessed image information from the processed input with all the unprocessed data intact. We simulated the inverse camera image processing and analyzed the resulting signals.

This thesis presents a partial solution to a complex project focusing on image generation, particularly addressing the challenges of working with the computationally demanding aspects of Color Correction Matrix (CCM) and gamma correction due to their non-linear nature. A complete solution was not achievable within this phase, but significant progress was made in simulating and testing the RGGB Raw data, meeting the initial project requirements. The thesis dives deeply into important theoretical underpinnings of image generation, various strategies for reversing the image pipeline, and their limitations. It also provides simulations of the implemented design. This study helps in further enhancements and refinements in future iterations, promising significant advancements in the design's capabilities.

Acknowledgements

This thesis was carried out at Uppsala University in cooperation with Volvo Cars, Gothenburg.

First and foremost I would like to present my huge appreciation to team manager Siddhant Gupta and supervisor Rubén Calvillo Portales at Volvo Cars, who provided guidance inspiration, and assistance, not only on a professional but also a personal level. Also, thank you to Bengt Jonsson at Uppsala University who guided me through this thesis with useful feedback and information.

A big thank you to my thesis partners Justus Hoffman and Anton Lind who provided a great cooperative environment that, through discussions and assistance, furthered the work we accomplished together.

Another huge thanks to the rest of the workers in the Volvo Cars HIL team who made the work and life outside of work way more enjoyable.

Last but not least, thank you to my family who are always there to support me in any endeavors.

List of Abbreviations

AD - Automated Driving Systems
ADAS - Advanced Driving Assistance Systems
AI- Artificial Intelligence
BRAM -Block Random Access Memory
CCM – Color Correction Matrix
CFA – Color Filter Array
CSI2 - Camera Serial Interface 2
DP – DisplayPort
DSP – Digital Signal Processing
ECU- Electronic Control Unit
FPGA - Field Programmable Gate Array
HIL - Hardware-in-the-Loop
IC - Integrated Circuit
IDE - Integrated Development Environment
ISP – Image Sensor Processing
IP - Intellectual Property
IU - Interface Unit
LUT -Look-up Table
ML – Machine Learning
MPSoC - Multiprocessor System on a Chip
PL -Programmable Logic
PS- Processing System
RTL - Register Transfer Level
sRGB – Standard RGB
SCL - Serial Clock
SDA - Serial Data
TB - Testbench
TRM - Technical Reference Manual
VCU - Vehicle Computing Unit
VDMA - Video Direct Memory Access
VIU - Vehicle Interface Unit
WFW - Waveform Window

Contents

1 Introduction

1.1 Background

Volvo Cars has been a leader in the automotive business since its founding, especially when it comes to safety. Ever since it invented the three-point seatbelt in 1959 to the speed cap in 2020 to stop over-speeding[1], the company has been committed to providing safe and secure conditions for drivers as well as pedestrians. Over the years, this dedication has changed to include active and passive safety features that guard against collisions and safeguard the people in the cars.

It has made significant strides in car safety by incorporating state-of-the-art technologies such as Safe Space Technology[2]. This comprises a variety of sensors, cameras, and radars that offer a full 360-degree view of the vehicle's surroundings, guaranteeing thorough situational awareness. Hardware-in-the-loop (HIL) testing is a crucial method for verifying and validating these advanced systems early in the development process.

HIL testing is a crucial component of these safety advancements. The development and validation of Volvo's ECUs and VCUs depend heavily on HIL testing, ensuring that every part functions perfectly as required before it is installed in real cars. HIL testing enables Volvo to carefully check and confirm the features of its safety systems, such as autonomous driving capabilities, sensor accuracy, and general vehicle response, by mimicking real-world settings. This rigorous testing ensures that Volvo cars not only meet but exceed safety standards, thereby supporting the company's commitment to achieving zero accidents.

Currently, particularly concerning this project, Volvo uses a software solution from an external vendor, for sensor simulation for the HIL testing environment. This approach has several limitations, including long lead times for updates and customizations, which can take several weeks to months. To enhance flexibility and control over the testing environment, Volvo aims to develop an in-house solution. This project seeks to replace the existing camera system in the HIL setup with a simulation environment, similar to what the external vendor provides, that can be customized and updated as needed thereby providing greater control over the testing process.

1.2 Overview of the Hardware-in-the-Loop Testing Environment

Hardware-in-the-loop (HIL) testing is a crucial technique used to verify and validate components and systems during the early stages of development. This method allows real electrical control units (ECUs) and vehicle computing units (VCUs) to be tested in a controlled, repeatable environment. In HIL testing, real components are tested using simulated sensor data, providing a robust method for ensuring system reliability before deployment in actual vehicles.

Hence, to test the ECU, the camera, which feeds visual data to the ECU in real-world scenarios is replaced with a simulation during HIL testing, but the ECU remains untouched. The essence of HIL testing lies in its ability to simulate real-world scenarios. The HIL environment proposed for this project consists of a Real-Time Simulation Platform, a Sensor Simulation PC, and various interface units as shown in Figure 1.1. These components work together to create a comprehensive testing setup.

The Real-Time Simulation Platform generates a real-time, high-resolution 3D environment. It

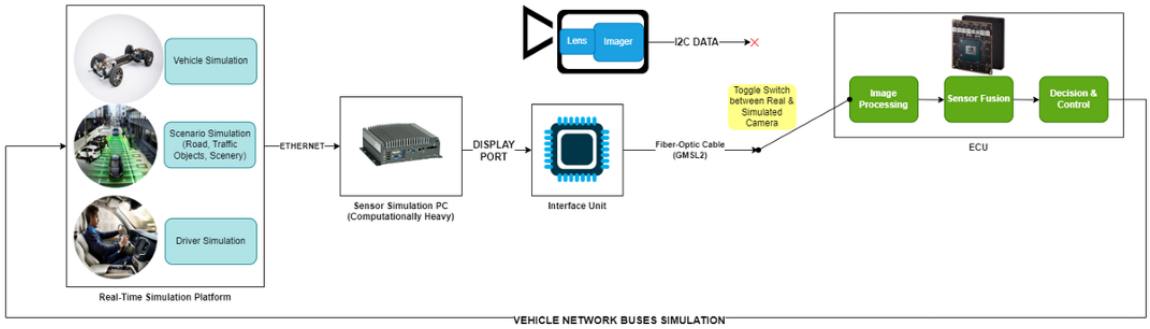

Figure 1: System Layout for Camera Verification on HIL Environment using Real Camera or Simulated Camera

Figure 1.1: HIL environment setup

models many facets of traffic conditions, vehicle dynamics, and sensor data. This environment can be changed to satisfy specific needs and test conditions.

Aurelion produced the required 3D images/videos for image injection from the perspective of the camera, as the camera sensors would sense it. An example of this can be seen in Figure 1.2. Aurelion is from an external supplier which means, Volvo has less control over the updates, custom changes, etc. By developing an in-house solution, Volvo aims to streamline the testing process, allowing for immediate updates and customizations based on specific requirements.

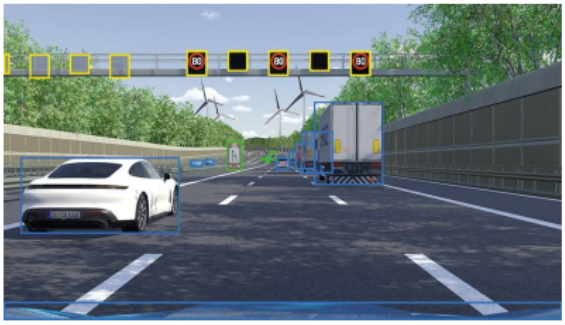

Figure 1.2: Camera perspective from Aurelion simulated 3D environment

The tasks performed by Aurelion, such as creating and interpreting sensor data, are computationally expensive. In the proposed setup, these tasks are managed by a powerful computer known as the Sensor Simulation PC, which is connected to the Real-Time Simulation Platform. It uses many graphics cards to produce the 3D scene and simulate sensor outputs like camera photos. Here, Volvo can generate images/videos as they want for a particular software change. The simulated data is then delivered to the Interface Unit.

The Interface Unit (IU) connects the Sensor Simulation PC with the vehicle's VCU. The output of the Sensor Simulation PC is fed into the display port, which is transferred to the VCU through the GMSL2 port. The VCU processes the image/video data and feeds back the actions necessary, to the Real-Time Simulation Platform essentially forming a closed-loop system. The

VCU is represented as the ECU in Figure 1.1. However, both would work similarly for this thesis.

As described, the practical application of Hardware-in-the-Loop (HIL) testing involves introducing simulated or real recorded data into the ECU to create visuals. For example, real-world situations can be replicated using data from a real camera, while specific scenarios can be assessed using synthetic data. The optical components of the camera, such as the lens and imager, are simulated in the described HIL setup. This makes it possible for the testing environment to replicate the precise point of view the sensors would experience in actual circumstances. These simulated images are produced by the Sensor Simulation PC and sent to the ECU through the Interface Unit. This technique is essential for verifying the camera and related image processing algorithms that the car uses.

1.3 Motivation for the Thesis

So, to recapitulate, the primary motivation for this thesis is to provide the HIL testing team at Volvo with greater control over their testing particularly in visualizing the environment .

The current reliance on Aurelion's external software solution presents several challenges:
1. Delayed Customizations: Every time the team needs to test specific environmental conditions or update the simulation parameters, they must contact Aurelion, resulting in a long wait time.
2. Lack of Flexibility: The current solution limits the ability to quickly adapt the testing environment to new scenarios, which is critical for the iterative development and validation of autonomous driving systems.

By developing an in-house simulation framework, the team can:
1. Reduce Lead Times: Implement and test new scenarios without waiting for external updates.
2. Increase Customization: Tailor the simulation environment to specific needs, enhancing the accuracy and relevance of the tests.
3. Enhance Innovation: Rapidly iterate and improve the testing processes, ultimately leading to more robust and reliable autonomous driving systems.
From the given HIL setup, it was deduced that the major changes need to be done in the Interface Unit, to make the whole system compatible and similar to Aurelion.
In the next sub-section, we will look into what changes/updates/Implementation Volvo expects to be done to the HIL setup corresponding to this project.

1.4 Scope of the Project

The focus of the project is to evaluate and implement the injection of the simulated (raw) image data directly into the ECU. Volvo, when proposing the thesis, collected several areas as mentioned below that need to be evaluated and developed:

1. Emulation of Optical Components: Since the physical camera components are replaced, their functions must be accurately emulated in the simulation.

2. Embedding I2C Data: Necessary communication data must be integrated into the simulation.

3. Timing and Data embedding: Data that does not contain the actual image information needs to be located, adapted, and parameterized depending on the camera used.

I2C is a communication protocol, widely used for exchanging data over short distances(intra-board). It consists of two wires, *Serial Data* (SDA) and *Serial Clock* (SCL), together they control the transmission of the data. It employs a master-slave configuration of data transmission. The master takes control of the bus and then reads/writes a frame containing the slave address and one or more bytes of data. Each data packet transmitted is acknowledged by the slave [3].

In this particular project, generally, the camera system and ECU communicate the camera configuration details such as the image sensor, and setting parameters like exposure time, gain, color balance, and others with each other through I2C. This provides the ECU, control over the settings, including control over altering them. Since we are removing the camera and simulating the image data, it is necessary to include this control interface to fully emulate the camera. The simulation environment needs to generate and respond to the I2C command just as a real camera would.

4. Whether an FPGA or a Microcontroller is best suited for this objective and how they should be programmed to achieve this.

5. The connection between the Sensor Simulation PC and the Interface Unit is facilitated via the DisplayPort output or any other viable interface. It is necessary to enable low-latency and efficient transmission of image data including reliability on the image quality with precise timing. This high-precision timing enables limited data buffering minimizing the lag caused by signal adjustment. The selection of this interface is another research aspect.

6. Evaluation of this prototype system shall be carried out for raw camera data injection at an approximate rate of 10 Gbit/sec and the possibility to support multiple channels of data streams.

This was the original scope of the thesis as developed by Volvo, which proved to be too broad and complex after an initial study was done. Hence, the goals of the project had to be reconsidered and reduced in complexity to make it achievable.

The existing solution had bitstream from the SD card, that was used to program the IU which cannot be reverse-engineered by Volvo Cars, so it was impossible to know what code was being loaded onto the IU. This meant the IU solution had to be built from scratch. Thus, the whole aim was to find a solution to implement the IU that can be customizable while retaining sufficient performance.

The primary objective remained focused on the Interface Unit (IU). However, instead of addressing complex aspects such as I2C data, timing, data embedding, and automation, the practical emphasis shifted towards achieving a basic proof of concept. This shift occurred partly because these aspects were overly complicated and not pertinent until a fundamental solution was established. Consequently, the new focus was to research and develop a straightforward solution for the IU, or its components, that could operate in a simple environment. The overarching goal of the thesis was to demonstrate that the IU could be modified, configured, fully controlled, and eventually produced in-house.

This project and report will focus on creating image data compatible with the ECU. Primarily, the AXI4 stream output of the display port is 8-bit RGB data and the ECU processes RAW12

data in Bayer pattern, which is discussed later. Eventually, it was decided also to figure out if 8-bit sRGB could be changed to RAW12 data, as this would open up the system to different types of image input, such as SD card and not just simulation platform, etc. A detailed block diagram of the system focused on in the thesis is shown in Figure 1.3. The block diagram explains where the camera was initially connected and the newer block that replaces it. Here the camera is replaced by the simulator and the IU.

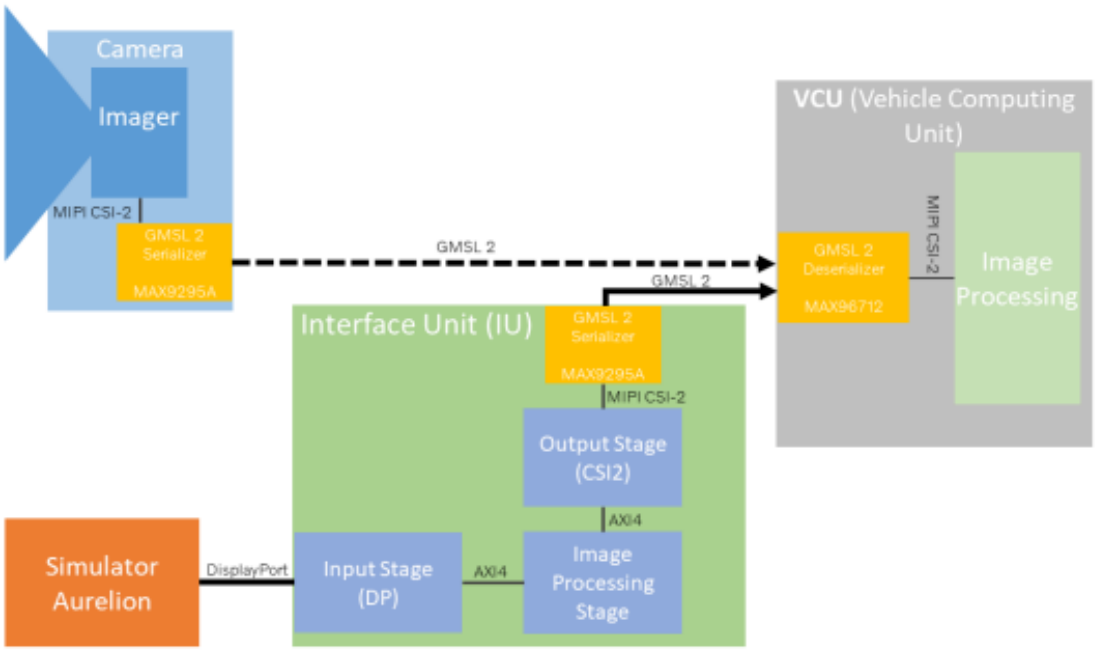

Figure 1.3: Block diagram of the image injection setup

1.5 Division of the Thesis

This thesis was a huge project and hence divided between three students. It was conducted in collaboration with Anton Lind, Justus Hoffmann, and the author of this thesis. Justus handled the input stage while Anton [4] handled the later, output stage while I focused on the image processing stage of the project. The input stage is composed of Justus trying to receive the incoming data stream from the simulation PC through Display Port, by implementing a Display Port receiver on an FPGA. In the output stage, Anton had to transfer the 12-bit raw data to the ECU through the GMSL2. He had to implement the GMSL2 in software for his project. All the work and theory presented in this report was conducted exclusively by me. Their reports will be referenced throughout this thesis when needed.

2 Technical Background and Theory

Vision is one of the most important senses in the world. It provides a lot of information from recognizing the object to its size, shape, distance, color, texture, etc, instantaneously, and hence

has become a crucial component of automobiles for safety considerations. The integration of camera technology in automotive applications has significantly influenced both vehicle performance and safety standards. The cameras aid in forward-collision warning, lane departure warning, traffic sign recognition, rear-view for parking and reversing maneuvers, and overall autonomous driving. These cameras with resolutions typically ranging from 720p to 1080p usually need to process data in real-time, at a speed of at least 60 fps. This requires a powerful parallel computation processor for greater performance and efficiency. Generally, ASIC is used for image processing applications but with greater requirements for computation, we are adopting FPGA along with CPU as it provides better flexibility for parallel processing.

2.1 Camera

The camera lies at the heart of the image processing system. The incident light on the camera lens passes through a series of processes before forming an image. The incident light passing through a sensor is devoid of any color and is a greyscale image with varying intensity that depicts depth. This image is called the raw image, a minimally processed data from the image sensor[5]. A Color Filter Array (CFA) layer consisting of red, green, and blue colors, just below the sensor captures the different intensities and produces the colors, this step is called Demosaicing. Further, it goes through more processing that results in the final image we see as the output of the camera image. Figure 2.1 depicts the general Image Sensor Processing (ISP) pipeline which depicts the several stages involved in image formation. Each block will be explained in detail in the following subsections.

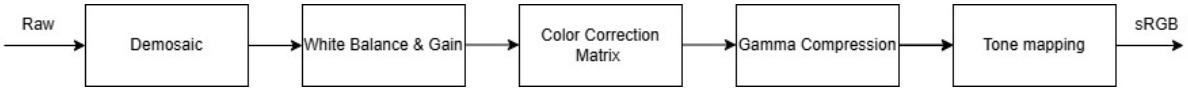

Figure 2.1: Image Sensor Processor Pipeline

2.1.1 Demosaic

Demosaicing is an essential computational process applied to raw image data captured using a Color Filter Array(CFA), where each pixel is overlaid with a specific color filter, allowing it to capture only one of the 3 primary colors - red, green, or blue.

There are several types of CFA, the predominant one being the RGGB Bayer format, which arranges color filters in a repeating 2x2 grid with two green, one red, and one blue filter. This pattern takes advantage of the human eye being more sensitive to green light, thereby improving luminance resolution. The RGGB Bayer Filter consists of alternating rows of red and green pixels on one row and green and blue pixels on the other.

The output from the 2-dimensional Bayer layer called the Bayer image, is interpolated to produce a 3-dimensional RGB image. Various algorithms such as Nearest-Neighbor or Bilinear Interpolation, Adaptive Homogeneity-Directed (AHD), and more, are used to reconstruct the color information, each aiming to hold onto as much detail as possible and keep the image sharp [6, 7].

Figure 2.2 and Figure 2.3 represent the RGGB Bayer layer.

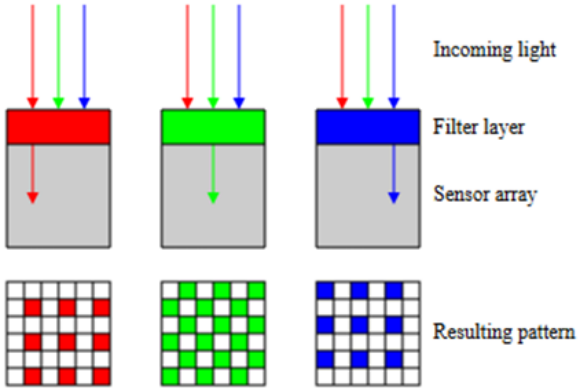

Figure 2.2: graphical representation of a CFA Filter [8]

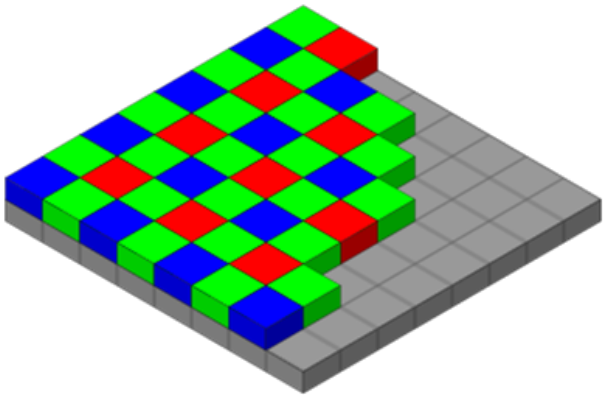

Figure 2.3: RGGB Bayer Pattern CFA [8]

2.1.2 White Balance and Gain

Gain is used to amplify the signal and boost the sensor sensitivity, which allows the capture of the images in lower light conditions. As the gain is image-dependent, finding a standard gain for all the images is difficult. An appropriate approximation based on example image datasets is done to simulate images.

White balance is a correction process used to balance the color cast of an image due to the illumination of the source light so that the objects that appear white in person are rendered white in the photo. Generally, the brightest spot of the image is adjusted to white, and all the other colors are balanced accordingly. Green channels have higher intensities than the other two colors, hence green channel gain is taken as a reference and the other channel-wise gain is calculated accordingly [9].

Figure 2.4: Image (1) raw image, (2) auto white balance, (3) custom white balance [10]

2.1.3 Color Correction Matrix

Color Correction Matrix is used to ensure that the colors captured by the digital sensors align with real-world colors as perceived by the human eye. The image sensors capture colors that often deviate from their actual appearance due to their physical properties and the design of CFA. The CCM employs a mathematical transformation where the RGB values from the previous step are multiplied by a 3x3 matrix. This adjusts the intensity values of each color channel, and the resultant output matches the standard sRGB or Adobe RGB. This matrix is generally camera-dependent. Though for simulation purposes, the coefficients of the matrix are determined through a calibration process, by capturing the standard color chart under controlled lighting conditions. Since it is a matrix operation, this is a highly compute-intensive operation [11, 12]. Figure 2.5 and 2.6 depict the effect off CCM processing.

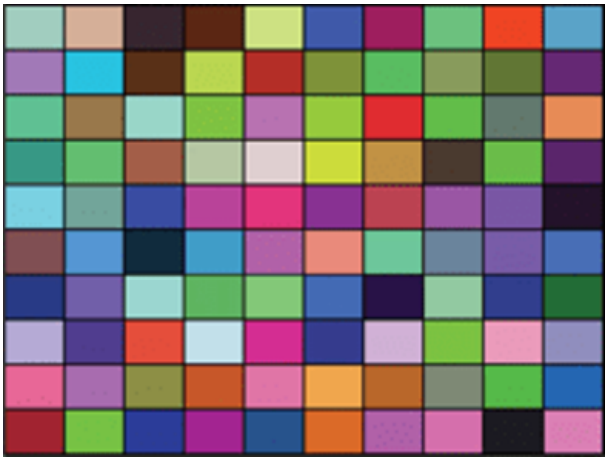

Figure 2.5: Original image without CCM processed [12]

2.1.4 Gamma Compression

Humans perceive light and color in a non-linear manner, i.e., human eyes are more sensitive to changes in dark tones than they are to equivalent changes in lighter tones. The camera senses

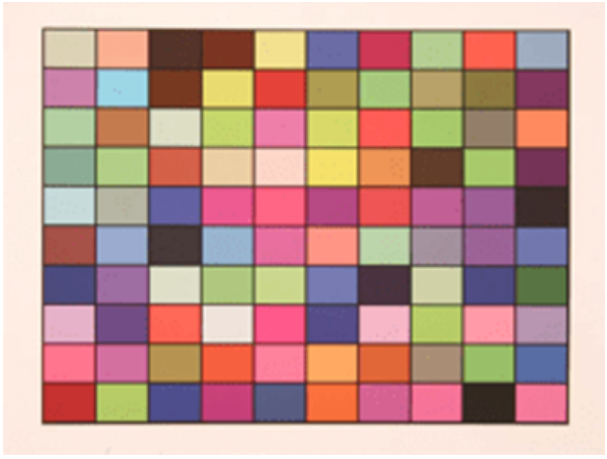

Figure 2.6: Processed CCM Image [12]

the changes linearly and hence requires updating the RGB values obtained after the CCM step logarithmically. There is an established power function where the output pixel, is the input pixel raised to a gamma value, and the gamma value is set around 2.2 in typical devices. This improves contrast and depth in areas more sensitive to the human eye [13].

$$v = u^{(1/gamma)} \tag{1}$$

v is the gamma corrected output and u is the linear input image. Figure 2.7 and 2.8 depict the effect of Gamma processing.

Figure 2.7: Linear Raw Image (gamma= 1) [13]

2.1.5 Tone-mapping

Global tone mapping is done to adapt a more dynamic range in contrast by increasing the high values and darkening the darker tones. The global tone-mapping can be approximated to the cubic polynomial expression 2. The output of this stage is the final processed image in the camera, typically sRGB image [14, 15].

$$f(x) = -2x^3 + 3x^2 \tag{2}$$

Figure 2.8: Gamma Encoded Image (gamma= 0.45) [13]

2.1.6 sRGB v/s RGB

The sRGB is the standard used for consistent color representation across different devices and platforms. It is a standardized version of the RGB color model and includes gamma correction to account for the non-linear way humans perceive brightness and color.

RGB image is the non-standard form where the colors appear differently on different devices, depending on how a device interprets it. It does not carry out other non-linear corrections required for better perception by human eyes. It is the colored image in the simplest form.

2.2 FPGA

The Field-Programmable Gate Arrays (FPGA) are integrated circuits that contain a matrix of programmable logic blocks, and several reconfigurable interconnects connecting them. These reconfigurable interconnects offer a high degree of flexibility in programming the blocks for specific use cases, thus leading to a high level of customization. FPGAs primarily consist of three main elements: Configurable Logic Blocks (CLBs), which perform logic operations; Programmable Interconnects, interconnecting the CLBs; and I/O Blocks, which provide the connections between the I/O pins on the FPGA and internal logic. This parallel architecture of the FPGA can perform multiple operations concurrently, significantly reducing latency and improving compute performance. This feature is highly sought after in complex, real-time operations and for high throughput, like image processing, automotive applications, consumer electronics, etc [16].

2.3 MPSoC

A *System-on-Chip* (SoC), unlike a CPU, is a system composed of all the processors – CPUs, GPUs, Digital Signal Processors, memories, peripherals, etc. on a single chip. A *Multi-Processor System-on-Chip* (MPSoC) includes several SoC processors aiding in handling multiple tasks simultaneously, thus being highly effective in complex embedded systems [17, 18].

2.4 AMD Vivado Design Suite

Vivado is an *Integrated Development Environment* (IDE), used to develop *Register Transfer Level* (RTL) code, analyze, synthesize, and implement FPGA designs. The FPGAs and MPSoCs are primarily coded in Verilog and/or VHDL. Vivado offers an IDE for this and along with Vivado HLS, certain codes can also be written in C/C++ which will be translated to RTL design on compilation. While one can certainly build custom codes, Vivado also offers pre-built designs called *Intellectual Property* (IP) cores, which form a basic building block to develop our design further. These cores are also highly optimized for efficiency [19, 20].

2.5 IP Cores and Subsystems

An IP core is a data or logic block with a reusable design that is pre-programmed to perform a certain function. The AMD Vivado Design Suite allows these IP cores to be configured via a graphical user interface or a scriptable TCL programming interface. This feature improves code reuse and speeds up the design process by using pre-configured, tested modules that interact with AMD (formerly Xilinx) FPGA devices with ease. There are a lot of IP subsystems available in the Vivado environment that combine several IP cores to provide a coherent solution. The AMBA AXI4 communication protocol is commonly used to connect these subsystems, enabling effective interaction amongst the cores. By dragging and dropping these subsystems into a Vivado block diagram, developers can easily incorporate them into their projects and streamline the process of creating an extensive RTL design. This simplified integration technique not only shortens development times but also guarantees consistency, better efficiency, and dependability across projects [21].

2.6 AMBA AXI4

The *AMBA Advanced eXtensible Interface 4* (AXI4) protocol is part of the ARM AMBA suite of protocols designed to facilitate communication in complex SoC designs. This protocol exhibits high -bandwidth, low latency, and secure transfer of data. There are three types of AXI protocols. The first one is AXI4, which is used for high-performance memory-mapped requirements, like video processing. For simpler memory-mapped communication, requiring low throughput, there is AXI4-Lite. The third interface is AXI4-Stream, which is used for high-speed streaming data. The AXI4 protocol has a master-slave-based communication architecture. The master initiates the communication, and the slave responds. The address and data channels are separate, and they employ handshaking signals to ensure reliable data transfers. Refer to Figure 2.9 for a better understanding of the protocol.

The data transfers happen only on the rising edge of the clock when the signals VALID and READY are high. In Figure 2.10, it can be observed that the data transfer occurs at the time instance T3.

A simple read transfer occurs when the master initiates a read transaction by sending the address and control signals information to the slave via the Address Read Channel. The slave responds by sending the data at the corresponding address to the master via the Read Data Channel as shown in Figure 2.11.

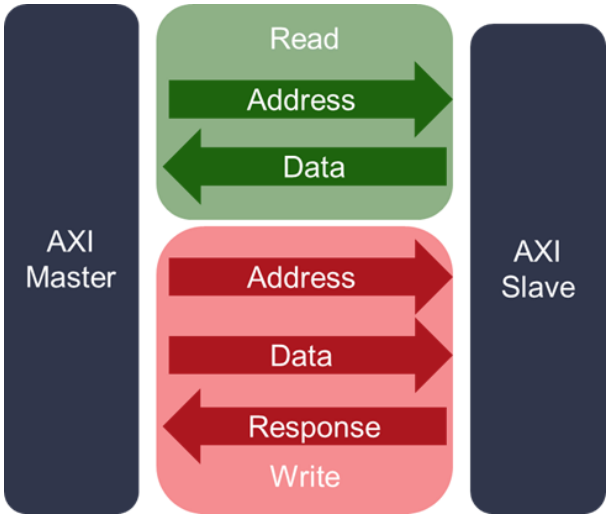

Figure 2.9: AXI interface with five read and write channels [22]

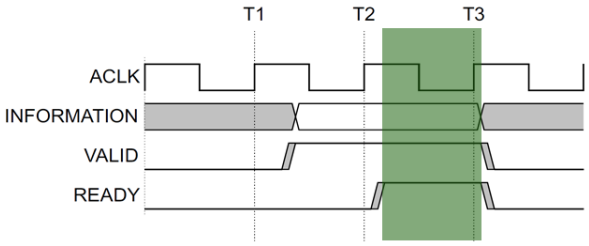

Figure 2.10: AXI4 information transfer with time on the x-axis [22]

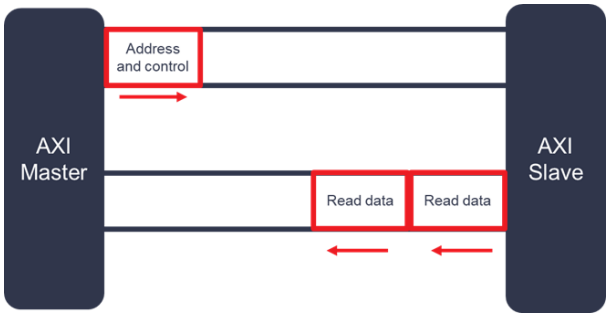

Figure 2.11: AXI4 Read transaction [22]

Similarly, for the Write transaction, Figure 2.12, the master sends the address and control signal information to the slave via the Address Write Channel. The master sends the data to be written at the address to the slave via the Write Data Channel. The slave acknowledges if the transfer was successful or not via the Write Response Channel [23, 22, 24].

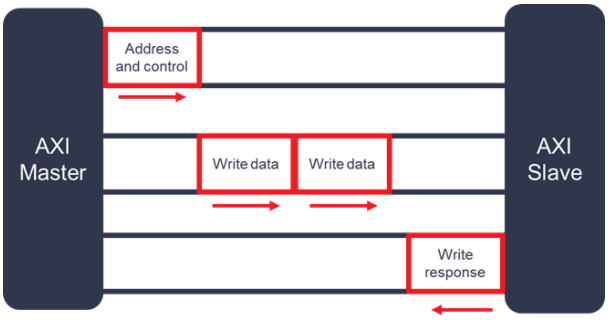

Figure 2.12: AXI4 Write Transaction [22]

3 Method

3.1 Algorithm

Several studies have been conducted to obtain raw images from processed RGB images, resulting in various algorithms. However, most of these studies indicate the need for metadata to be already established, such as the hardware characteristics of the camera. Since the images are simulated, we do not have access to the hardware characteristics. Fortunately, for the current project simulation, the hardware characteristics can belong to any camera for the initial development process. However, finding a comprehensive dataset that documents all the camera properties required for this project was difficult. Eventually, Brooks *et al.* [25] discusses the Darmstadt Noise Dataset. The Darmstadt Noise Dataset consists of 50 high-resolution noisy images taken on four different cameras with varying sensors[26].

Figure 3.1 below represents the algorithm [15] used in this thesis to convert the sRGB images to RAW images. It is a reverse of the ISP pipeline shown in Figure 2.1. Here, each block reverses the processing done initially to achieve RGB and we finally end with a RAW RGGB image. Each step of the algorithm is explained in detail below.

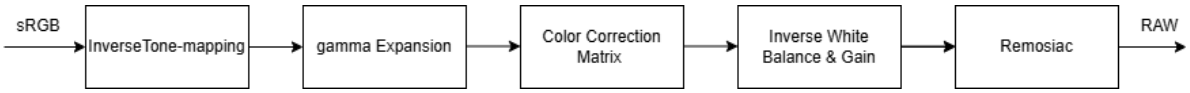

Figure 3.1: Reversed ISP Pipeline

Phases of the algorithm:

1. Input Preparation:
   (a) Input: obtain an 8-bit RGB image, 1920x1080 (typically in JPEG)
   (b) Normalization: Normalize the RGB image by scaling the pixel values to the range [0,1].

2. Inverse Tone-Mapping:
   (a) Function: Inverse of the tone-mapping curve used during the original image processing.

(b) Implementation: Apply the inverse of the function used for the tone-mapping curve [15], i.e.

$$x \;=\; \frac{1}{2} \;-\; \sin\left(\frac{\sin^{-1}\left(1-2x\right)}{3}\right) \tag{3}$$

The proof that this is the inverse of the cubic polynomial tone-mapping equation(2) is given in Appendix D.

3. Gamma expansion:

   (a) Function: Revert the gamma compression to retrieve the linear representation of the image.

   (b) Implementation: Apply the power function with the inverse gamma value, typically

$$\gamma \;=\; 2.2$$

$$u = \; v^{\gamma} \tag{4}$$

   where u is the linear image and v is the gamma-corrected image.

4. Color Space Inversion:

   (a) Load Color Correction Matrix: Choose the appropriate CCM from the dataset.

   (b) Invert CCM: Apply the inverse of the CCM to convert the sRGB color values back to the camera's original color space.

5. White Balance and Gain Inversion:

   (a) Global Gain: Apply the inverse of the global digital gain used in the initial processing pipeline.

   (b) Channel-Specific Gains: Estimate the inverse gains for the red and blue channels based on typical values or specific dataset parameters. Use these to adjust the color balance.

6. Remosaicing:

   (a) Bayer Pattern Application: Remosaic the three-channel RGB image by applying a Bayer filter pattern (e.g., RGGB), which results in a single-channel output with sparse color samples.

3.2 Hardware

The type of hardware depends on the type of computation needed to be done. Typically for image/video processing one uses GPU or microcontroller or the latest FPGA. GPUs help in faster processing due to their parallel architecture, making it highly convenient for real-time applications. Microcontrollers are useful in relatively simpler applications as they do not possess high computing power. Over the past decade, FPGAs have emerged as the main hardware for image processing as they are highly parallelizable and highly customizable. They can be built

exactly per specification and in several scenarios work a lot better than GPUs. In this project, as the company, Volvo Cars are already using a FPGA-based setup in their HIL testing, it was decided that FPGA is the go-to option.

3.2.1 Zynq™ UltraScale+ MPSoC

This project utilizes a particular MPSoC called the Zynq™ UltraScale+™ MPSoC [27] which is developed by AMD Xilinx. This MPSoC is an FPGA-based system consisting of a Processing System (PS) and Programmable Logic (PL). The PS is like a CPU, it runs the code sequentially while the PL is the array fabric used for parallel logic. The PS and PL can also be programmed individually in different languages like C/C++ and Verilog/VHDL respectively. The detailed image of the Zynq™ UltraScale+™ MPSoC can be seen in the below Figure 3.2.

The TEB0911 UltraScale+ MPSoC Board by Trenz Electronics is the System-on-Module (SOM) used in this project. This module hosts the required MPSoC. This module was chosen as it was already used at Volvo Cars for other applications. More information on the board will be given as we go further into the project. The TEB0911 module can be seen in Figure 3.3.

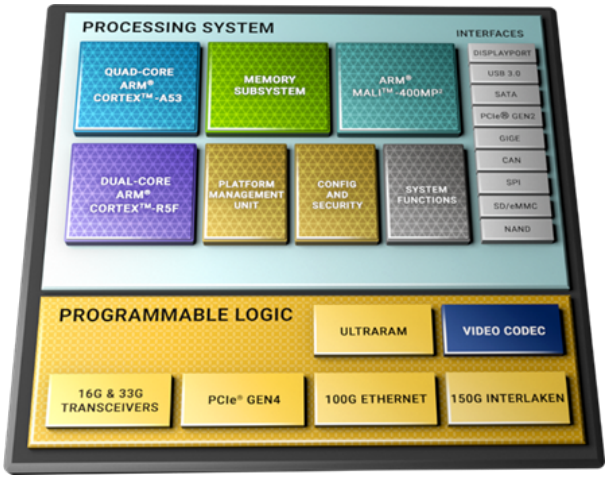

Figure 3.2: Zynq UltraScale+ MPSoC composition [27]

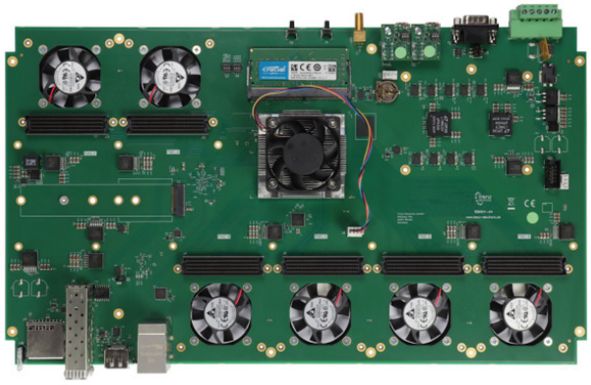

Figure 3.3: TEB0911 UltraRack+ MPSoC SOM [28]

3.2.2 FPGA Resource Estimate

A rough estimate was made to identify how many resources each of the blocks might require and assess if it is possible to implement on the given FPGA.
Few terminologies before the calculations:
Look-Up-Table (LUT): It is a data structure used to map input values to corresponding output values based on predefined relationships. The function is precomputed for all possible input values, and the results are stored in a table. These ensure fast lookups, making real-time image processing feasible.
DSP slice: they are specialized hardware blocks in FPGAs that perform high-speed arithmetic operations. they make up the adders, multipliers, accumulators, etc. They are efficient and can be parallelized to handle large volumes of data simultaneously.
BRAM: They are on-chip memory blocks in FPGA used for storing data. They are used to implement large memory structures, loo-up tables, and buffers.

1. Normalization and Clipping

   LUTs: Minimal, mainly for the implementation of control logic.

   DSP Slices: Typically, normalization can be handled using LUTs or DSP slices if done in fixed-point arithmetic. The estimated DSP Slices: 0-50, if fixed-point is used for accuracy.

   BRAM: Storage for the frame; The estimated BRAM: 2-4.

2. Gamma Expansion

   LUTs: Moderate to high, due to the non-linear function (power function) computation. It could be implemented using LUT-based approximations or a small lookup table. The estimated LUTs: 500-1000.

   DSP Slices: High, for repeated power calculations per pixel. The estimated DSP Slices: 200-300.

   BRAM: Utilized for storing intermediate results if processed in chunks. The estimated BRAM: 2-4.

3. Color Correction Matrix (CCM) Application

   LUTs: Moderate, for implementing matrix multiplication logic. The estimated LUTs: 300-600.

   DSP Slices: High usage due to the multiplication and addition of three channels per pixel. The estimated DSP Slices: 300-400.

   BRAM: This may require additional buffers for line or frame storage. Estimated BRAM: 4-6.

4. Inversion of White Balance and Gains

LUTs: Minimal to moderate. The estimated LUTs: 100-200.

DSP Slices: Moderate, as this step involves straightforward multiplications for gain adjustments. The estimated DSP Slices: 100-150.

BRAM: Intermediate storage if processed separately. The estimated BRAM: 2-3.

5. Mosaicing

LUTs: Moderate, primarily for data rearrangement and selection logic. The estimated LUTs: 200-400.

DSP Slices: Minimal to none, as this process is more about data organization than computation. The estimated DSP Slices: 0-10.

BRAM: Requires handling of the full frame in a specific pattern, potentially needing additional buffering. The estimated BRAM: 2-4.

The overall resource summary:
Total LUTs: Roughly 1100 to 2200 LUTs.
Total DSP Slices: About 600 to 910 DSP slices.
Total BRAMs: Approximately 12 to 21 blocks.

These were the rough estimates made initially before the design in Vivado. However, there were considerable changes in the design due to several issues faced during the implementation.

3.3 Software

3.3.1 MatLab

MatLab is an excellent development environment for images as they are represented as matrices. Hence, it was used in the initial stage for verifying the algorithm. This enabled rapid prototyping and debugging, ensuring each step of the algorithm produced the expected outputs.

3.3.2 Vivado

AMD provides the software support for their MPSoC. The Xilinx Vivado serves as an IDE for designing and developing the hardware code to be programmed on this MPSoC. Vivado, version 2019.2 was used as it was the latest version that supported TEB0911 and it had a few examples of designs helpful for the input display port design which sends the data for image processing.

3.3.3 MPSoC

Within Vivado, there is an option to choose the hardware on which the specific code will be eventually uploaded. It is crucial to choose the right board, as things like pins, ports, clocks, timings, etc. are specific to the hardware and the software will follow the same. In this project, the board "ZYNQ-UltraScale+ TEB0911-09EG-1E. SPRT PCB: REV03, REV02. (xczu9eg-ffvb1156-1-e)" is used. The name within the parentheses refers to the serial number and version of the board.

As mentioned previously, the MPSoC consists of two parts, *Processor System* (PS) which is the processor that can be coded in C/C++, and the *Programmmable Logic* (PL) which is the FPGA fabric and needs to be coded in *Hardware Description Languages* (HDL) like Verilog or VHDL. In this project, the PL is coded in Verilog. The PL can handle complex code written at the register level. It gives a lot of autonomy over how the hardware interacts with data.

3.3.4 Vitis IDE

Vitis is an extension of Vivado, a part of Vivado Design Suite. It is an IDE to work on the PS of the MPSoC. This IDE offers an easier way of writing and uploading code to the PS using C/C++. The hardware design of the FPGA/PL is also uploaded to Vitis to bring together the PS and PL and operate both simultaneously. Within this project, the Vitis version is the same as Vivado, i.e. 2019.2 as it is necessary for compatibility.

3.4 Programming the MPSoC

3.4.1 HDL Wrapper

Once the RTL design is completed for the new project, an HDL wrapper needs to be generated. This HDL wrapper consists of the hardware code that describes the RTL design. It is generally in VHDL or Verilog. This project will use Verilog as it is more comfortable to use.

3.4.2 Synthesis and Implementation

The HDL wrapper of the RTL code is next synthesized and then implemented onto the hardware, here, the FPGA. During the synthesis phase, the HDL code (VHDL/Verilog) is translated into a set of logical gates and interconnects. During the synthesis phase, Vivado parses the HDL files and libraries, checks for syntactical issues, builds a hierarchical design database, and optimizes the design by removing redundant logic, combining the gates, and restructuring the design for better performance and lower power consumption. The output of this stage is called a *netlist*, which will be used to implement FPGA Hardware. In the Implementation phase, the components of the netlist are assigned specific physical locations on the FPGA and routed to create a pathway for data flow between the components. This is also a good time to do timing analysis to check if the design meets the required timing specifications and *Design Rule Checking* (DRC), to check if other design constraints are satisfied. Once all the checks are done, a bitstream can be generated which describes the whole system with all the connections, voltage levels, timings, etc. This file is uploaded to the FPGA. Here the whole programming process ends if only the FPGA had to be programmed. If there is a need to utilize the PS, then the bitstream needs to be exported to the Vitis IDE through Hardware Specifications in Vivado. Within Vitis, it is possible to code the PS along the already described FPGA/PL [29, 30].

3.4.3 Vitis IDE and Hardware Manager

Vitis IDE streamlines the development process by enabling the creation of *Application Project* based on the *Hardware Specification* files including the bitstream. The projects can be modified according to specific functionalities in C/C++, to be executed on PS. Once the necessary changes are made, the project needs to be built using the build project command on the GCC/G++ compiler. The compiled software is then linked with the board support package (BSP).

Post-build, the project is deployed directly on the hardware using the command "*Debug As →Launch on Hardware* (Single Application Debug)" with the error-free code being uploaded on both PS and PL.

If the project only involves PL and not PS code or needs no more changes in the PS code, then the code is directly uploaded onto the board in Vivado. Within Vivado, the Hardware manager identifies a new target as "*Hardware Target*". The target is the board connected to the PC through a JTAG. The hardware target offers several options to the users on how to interact with the FPGA, such as, directly uploading the code, monitoring, and debugging in real-time through the waveform window, adjust settings and configurations directly on the FPGA without the need for re-compilation of the entire project [31].

3.5 MatLab Implementation

The whole purpose of this project is to obtain an unprocessed raw image from a processed sRGB image. This involves editing the image at the pixel level. Given that an image is a matrix of numbers representing the image intensity at each index, MatLab provides an excellent development environment to manipulate the images at the indices level. MatLab provides powerful computational and visualization capabilities, so in this project, it is used to prototype and refine the algorithm essential in converting the sRGB to raw RGGB image. This approach helped to understand the algorithm better and ensured that the foundational logic was accurate and functional before transitioning to FPGA.

3.5.1 MatLab Functions

The whole MatLab code for the ISP can be found in Appendix B. Here, let us walk through the main functions in the code. The code follows the algorithm mentioned in Section 3.1.

The process begins with the normalization of the RGB 1920x1080p image and calling the *unprocess* function. Normalization is crucial as it scales the pixel values to a standard range between 0 and 1, ensuring consistency and compatibility with subsequent processing steps. This is particularly important because image processing algorithms often assume input data within a specific range for correct functionality. The *unprocess* function calls the rest of the functions involved in inverse ISP.

The *unprocess* function calls the other functions further. The functions are called as per the algorithm mentioned, starting with:
1. *inverse_smoothstep* which handles the inverse tone-mapping by restoring the image's linear light intensity values.
2. *gamma_expansion*, as the name suggests handles the inversion of the gamma compression and leads to linearization of the perpetual brightness of the image.
3. *apply_ccm*, here the image undergoes inverse color correction.
4. *safe_invert_gains* handles the white balance and digital gain. The blue and red gains are estimated with respect to the green gain which usually lies within the range [0.5, 1.1]. In this project, we randomly choose the gain within this range.
5. *mosaic*, where the RGB values are re-arranged into the desired pattern to simulate the raw sensor output.

While all the functions are straightforward, computing the CCM involves several substeps:

1. Load Color Correction Matrix: The CCM is obtained from the dataset. The matrix values are derived from a color calibration process involving standard color charts.
2. Invert the CCM: The CCM used during the image processing needs to be inverted. The inversion maps the color values from the processed sRGB space back to the raw sensor space. Both, this and the previous step are carried out by the function *random_ccm*.
3. Apply the Inverted CCM: The inverted matrix is applied to the image to correct the color values. This process involves matrix multiplication, which adjusts the RGB values according to the inverse of the color correction applied during the initial processing This step is executed by the function *apply_ccm*.

3.6 Vivado Implementation Specifics

Vivado provides various IP cores and subsystems for image/video processing projects. But this project, which needs inversion of the image pipeline does not have any existing code examples. This is because it is extremely unpopular to reverse an image pipeline to Raw format as it contains lots of data, which increases the storage memory and includes lots of redundant data. The output of a typical digital camera is a processed image such as sRGB image. This meant that the entire image pipeline had to be created in Vivado. The effort was made to use certain pre-existing individual IP cores to see if they could function as intended in the inverted pipeline.

In this project, we aim to achieve the conversion of RGB to Raw and sRGB to Raw. As sRGB is just an improvement by adding a few extra steps to the design targeting RGB, the following will discuss the sRGB-based pipeline, and the differences will be discussed later in the result section.

Figure 3.4 below illustrates the overall RTL design of the reverse ISP pipeline. The Zynq processor at the center drives the whole system by providing the clock signals. The interconnects handle communicating the master XI4 signals to different cores. the first interconnect initiates the VTPG, to generate patterns, which is next buffered in the Video Direct Memory Access (VDMA). This stores the incoming stream of signals before transferring it to the image processing modules. The next stage is the inverse gamma IP core, in this module going by the design, since only the gamma value changes, we can repurpose the pre-built IP core available from Xilinx. The output of this stage is carried forward to the CCM IP core. Here too, a pre-built CCM core was used, within this module, the inverse white balance and digital gain values are also processed. Since the gains values are constant for a given camera, they do not vary for a system, hence to reduce extra computational blocks, they were calculated and included in the CCM matrix. The output of this stage reaches the final stage, the Bayer IP module (or the remosaic module). This module was coded in Verilog and this module rehashes the RGB data into the desired RGGB format and then the signal gets multiplied by 4 and sampled to obtain 12-bit RGGB data. While this overall design shows a standard RTL design, this design did not function as expected. More details on the respective video blocks, the results, and the issues faced will be discussed in the following sections.

Here is the general design of the reverse ISP pipeline.

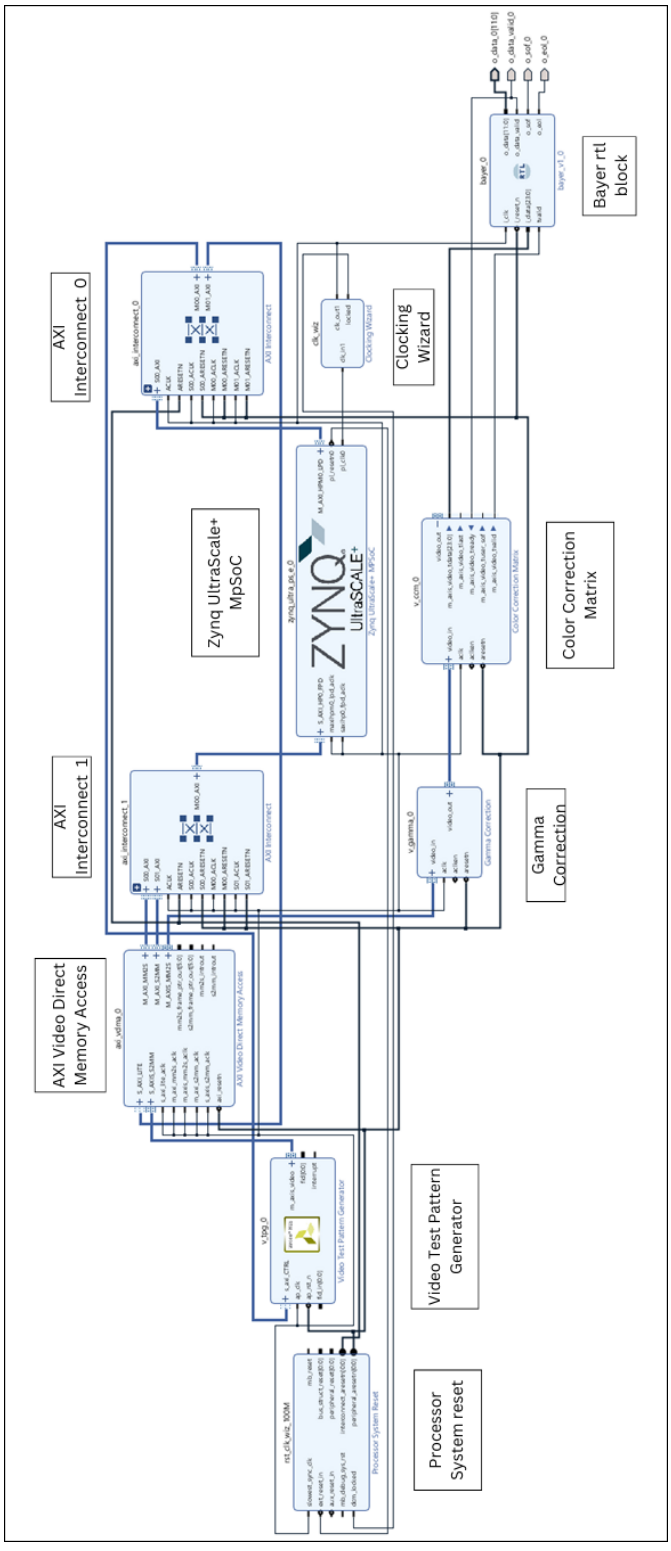

Figure 3.4: sRGB to RAW12 Design

3.6.1 Clocking Wizard

The Clocking Wizard IP [32] is seen in Figure 3.5. This IP takes in input clock signal from a processor or other IPs generating clock signal and provides the clock signals of necessary frequency to drive the rest of the design. The clock takes in a single input (clk_in1) and can generate 7 individual output clocks (clk_out1, clk_out2,. . . , clk_out7) of varying frequency to drive 7 different components. The clocking wizard is connected to the external ports reset_rtl_0 and clk_100MHz. These are the global reset and clock signals of the entire RTL design. During the synthesis and implementation phase, these ports are mapped to physical pins on the FPGA. Simulations of these ports can be handled within a testbench or directly on the Waveform Window.

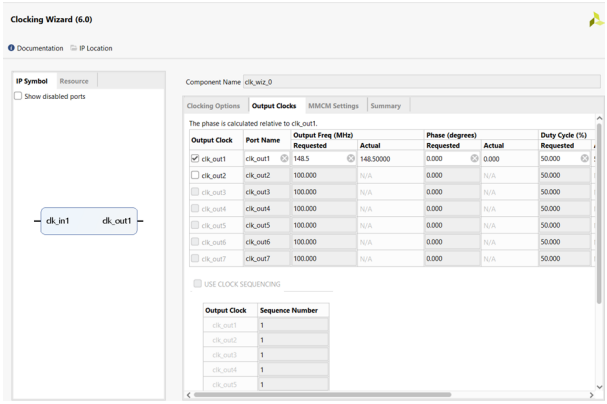

Figure 3.5: Clocking Wizard

3.6.2 Processing System Reset

The Processor System Reset IP [33] is seen in Figure 3.6. This IP generates all the resets in the RTL design and ensures that it is propagated in a controlled and orderly manner throughout the design ensuring stability and preventing errors. It interfaces directly with both the PS and PL, facilitating a unified approach to system resets. This IP, which connects the peripherals, interconnects, and processor, can be customized for desired reset and delays. This IP is often included in the first stage of the block design.

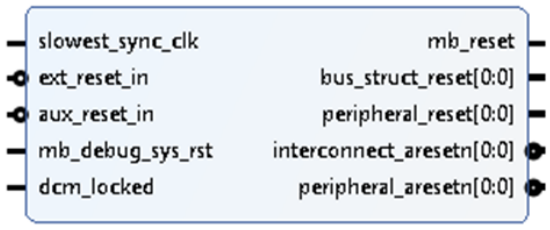

Figure 3.6: Processing System Reset

3.6.3 AXI Interconnect

The AXI Interconnect IP [34] can be seen in Figure 3.7. This IP connects one or more AXI master devices to multiple slave devices. In the design Appendix A.2, it can be observed that the two AXI interconnects connect The Zynq PS with the other peripherals like AXI Video Direct Memory Access (VDMA), and VTPG. The input clocks and resets are present for synchronization and are associated with the respective master and slave.

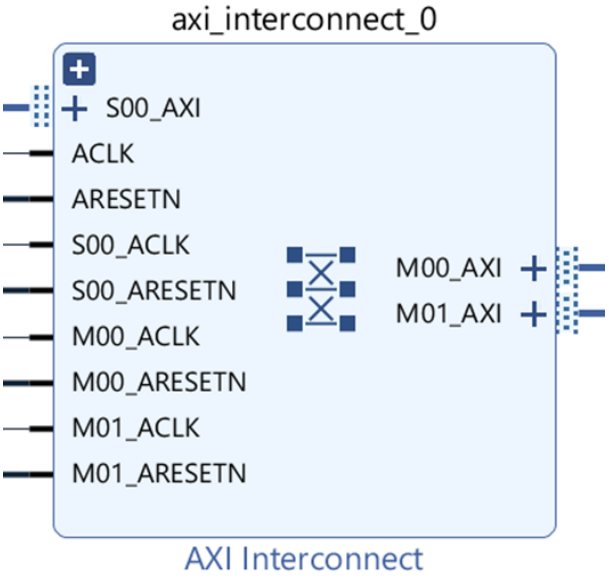

Figure 3.7: AXI Interconnect

3.6.4 Video Test Pattern Generator

The Video Test Pattern Generator (VTPG) IP [35] as shown in Figure 3.8, is used to generate video test patterns for video system evaluations, debugging, and setup. The VTPG is helpful in testing and validation of color and motion handling of video processing systems. It can be customized to produce various test patterns, including color bars, zone plates, and motion patterns, which are used to analyze and debug the quality of the system. This provides the necessary patterns as input to the image processing block within this project. As VTPG supports a range of color formats, including RGB, YUV444, etc, we can select the required pattern within the VTPG's configuration GUI. The clock (ap_clk) and the reset (ap_rst_n) are controlled through the clocking wizard and the processor system reset. The input is the AXI4-Lite into the s_axi_CTRL and m_axis_video is the output transmitting the necessary test pattern. The expanded version m_axis_video_TDATA[23:0] is the output from the system showing that the output AXI4 interface is 24-bit wide. Additionally, TREADY, TVALID, TLAST, etc, provide the necessary synchronization between the VTPG and the rest of the image processing block.

Figure 3.8: VTPG

3.6.5 Color Correction Matrix

The Color Correction Matrix IP in Figure 3.9, is, as the name suggests, used to perform color correction operations like adjusting white balance, color bias, brightness, and contrast through a 3x3 matrix of programmable coefficients. The core supports 8,10,12, and 16 bits per color component, and clipping and clamping controls to prevent color values from exceeding desired limits. The clock (aclk) and reset (arestn) are used to control the block. This IP also has _tlast, _tready, and _tvalid for synchronicity. m_axis_video_tdata[23:0] signal provides the output to the next stage.

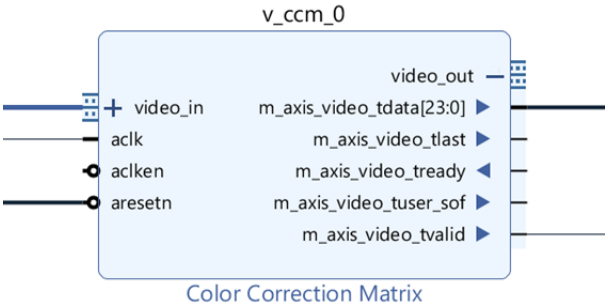

Figure 3.9: CCM IP Core

3.6.6 Gamma Correction

The Gamma Correction IP core [36] in Figure 3.10 is a hardware block used to manipulate image data to match the non-linear characteristics of the human eye through gamma encoding. It consists of a look-up table (LUT) which can be dynamically updated to adjust the gamma values. It can also be updated with predefined values. This system takes in an AXI4 input and outputs 24-bit AXI4 data. The aclk and aresetn are used to control the block.

3.6.7 Bayer RTL Block

The Bayer RTL block is a custom block built for this project as there does not exist any IP core converting RGB to RGGB image. The RTL block consists of different state machines to sort the incoming AXI4 data into the required pattern. It has support signals such as clock, reset, _tdata, and _tvalid synchronization. This IP is a modification of the open-source code available from Vipin K Menon [37].

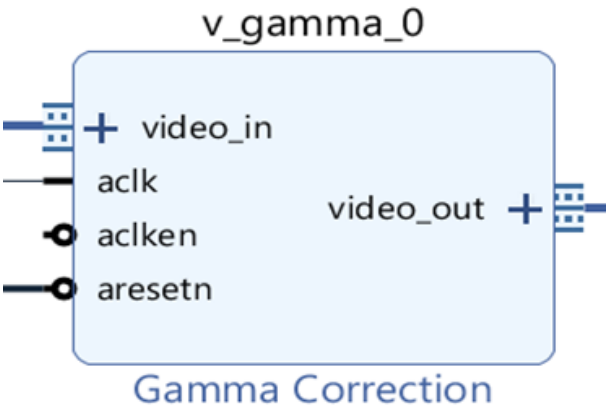

Figure 3.10: Gamma IP Core

4 Simulation Setup

Hardware design is a complex process involving several components. Hence, it is crucial to simulate the whole design in simulation software before deploying it onto the hardware to ensure the design performs as per requirements. Knowing how simulations work in any software is important as it enables easy debugging. After the RTL design is developed for synthesis, and implemented, and a bitstream is generated, it can be flashed onto the hardware. A simulation setup does not require the design to be synthesizable, implemented, or a bitstream generation. It requires only a valid RTL design. This reduces a lot of development time and helps to debug the design early into the development phase. Vivado provides a separate executable tool for Simulations.

Since it is a simulation, the input ports cannot be mapped to any physical pins, requiring software inputs. The software input can be achieved in two ways. One is to directly configure the input signals in the waveform window (WFW) of the Vivado simulation system and the other is to generate a testbench. A testbench is a Verilog/VHDL-based file that defines how the input signals need to behave and how the ports are interconnected. In this project, the signals are directly configured in the waveform window due to time constraints.

5 Results and Discussion

5.1 MatLab Execution

The initial step in the execution of the project was to make sure that the algorithm to convert the sRGB to Raw RGGB worked as intended. MatLab is one of the best IDEs for Matrix manipulation required for image processing. MatLab offers several toolboxes to achieve various image processing functions, but in this project, no toolboxes were used as it was necessary to learn how the code works at every line. This helps to understand the code better when implementing it in HDL. This subsection presents a series of images after every step of inverting the image pipeline. Figure 5.1 displays the example image used in this project for testing. It is a 1080x1920p RGB888 image.

The very first step was to normalize the image within [0,1].

Figure 5.1: Example RGB Image

5.1.1 Inverse Tone Mapping

Tone mapping, which is done to enhance contrasts is a sine curve as discussed in the section 2.1.5. Here, in Figure 5.2, it can be observed how inverting the tone reduces the dynamics in the image. To achieve this, the inverse of the polynomial function equation(2) is used. The inverse tone function is referenced here in equation(3).

Figure 5.2: Inverse Tone Mapped Image

5.1.2 Gamma Expansion

The standard value to operate the gamma expansion is the gamma value to be 2.2, but while working on MatLab, it was not possible to raise the value to a rational number. It would throw an error and exit the program. So, a whole number, 2, was chosen as a suitable substitute as it still falls within the range [2, 8]. Choosing a whole number also reduces complexity while working at the register level. Figure 5.3 refers to the image after the inverse tone-mapped image

is also Gamma corrected.

Figure 5.3: Gamma Expansion

It can be observed from the original sRGB image, this image has a more linear color gradient.

5.1.3 Inverse Camera Color Matrix

As the matrix is camera-specific, it was hard to design the matrix for the camera used at Volvo Cars, due to lack of raw data availability. After discussion, it was decided to go with any example camera in the prototype phase. The Darmstadt Noise dataset, with data from four different consumer cameras, provides four different sets of CCM values. While Brooks et al, suggested using a random convex combination of these four cameras, it led to a lot of errors within the MatLab environment and a lot of calculations. Multiplications and additions are highly resource-intensive operations within the FPGA. Keeping this in focus, only one of the four CCMs was taken and inverted to achieve inverse CCM. Figure 5.4 is the output of the Inverse CCM phase.

5.1.4 Inverse White Balance And Digital Gain

The white balance is achieved by applying channel-specific gains to the red, green, and blue channels of the image, compensating for any color cast introduced by the lighting. And, for the unprocessing, we are inverting the white balance gains. The gains are estimated for the red and blue channels, as the whole gain is normalized to the green channel. The red channel gain lies in the range [1.9, 2.4] and the blue channel gain lies in the range [1.5,1.9]. The overall gain to amplify the overall brightness is applied uniformly across all the channels with a standard value of around 0.8. Figure 5.5, is the image after following this step.

5.1.5 Remosaic

The output of the previous stage is an RGB image. This image can now be converted to the RGGB Bayer pattern, Figure 5.6, by just omitting two-thirds of the color pixels. It was straightforward and the following image is the final Raw output in Bayer form. This satisfies the requirements of the project. The only other necessary step is converting the 8-bit Raw

Figure 5.4: Image after Inverse CCM

Figure 5.5: Inverse White Balance And Digital Gain

output to 12-bit Raw output. This can easily be taken care of in Verilog by multiplying the image data by 4. Though this image looks distorted here, the image contains all the information required for the ECU to process. The ECU expects an RGGB 12-bit image. Also, here, within the document, the image looks dark, almost black at certain places. This is not the case with the actual image, though dark, it still contains lots of information that we are unable to see clearly.

5.2 Verilog Implementation

5.2.1 Discussion

While the MatLab implementation helped to identify a few issues and enabled simpler solutions, The RTL implementation was a lot more difficult for several reasons such as issues in dealing with certain IP cores, memory constraints, and just the complexity of the whole project. As discussed before, there is an under-representation of research where a processed RGB image

Figure 5.6: RGGB Bayer output

needs to be unprocessed to obtain a raw image. This implies that there are not many supporting blocks for this within Verilog. This meant creating new IP blocks as the conversions required.

As discussed in the scope of the project, there are two different inversions we are looking at. One is the conversion of RGB to Raw RGGB image format and the other is the conversion of sRGB to Raw RGGB image format. Since RGB to Raw is a subproblem of sRGB to RGGB, the whole discussion up until this point concentrated on the whole conversion. In the following, it will be discussed separately as RTL implementation of the whole system was difficult and only the RGB to Raw conversion was successful. The output 8-bit raw image is extrapolated to a 12-bit raw image by multiplying by 4.

5.2.2 sRGB to RAW12

The implementation of sRGB to RAW12 is composed of several parts as shown in Appendix A.2. As can be seen in the design, the inverse tone mapping block was removed as the sine functions used in the design were using more resources than on the FPGA. Since this processing is for automobiles and not the photography industry, a call was made to avoid the block. However, more research can be done to optimize this design to include that block. The CCM IP provided by Xilinx was repurposed in the design, as it does the required 3x3 matrix multiplication. Since our requirement was for an inverse matrix, the data was calculated manually and entered into the matrix to represent the inverse matrix. During the calculation for the inverse matrix, the inverse White Balance gain and inverse gain were also included to avoid usage of the resources in calculating them. This is possible as the values are standard for the given camera and remain constant throughout the design.

It was hard to get the system running during the simulation. The implementation faced issues due to the VTPG not working. It would not trigger ON. A separate design, Appendix A.1, to get the VTPG working was implemented, yet the result was the same. Several changes were made but the issue would not be resolved. So, to get the system running for test purposes, a simple custom RTL code was written to generate input signals. The value can be changed during simulation in the WFW. The RTL code, named datagen, is used in the main design as an RTL block. The design can be viewed in Appendix A.3.

Figures 5.8 and 5.7 are generated for the design in Appendix A.3. Initially, the output of the CCM block was zero for all the inputs. After further examination, the initial error was more human, as a mistake was made in calculating the inverse of the matrix. Necessary changes were made and the system still showed zero output. The error seemed to be because the pre-built CCM block has an upper bound and lower bound on the index values of the matrix. The Inverse matrix values which were calculated were way over the bounds on either side. Since few of the values were extremely small, they were truncated, almost equal to the bounds, or zero. This meant that repurposing the CCM block was not a good idea. A new custom block had to be made. Due to time constraints, it was not possible.

5.2.3 RGB888 to RAW12 image

The RGB888 to RAW12 Bayer image conversion is straightforward. Here the RGB image is devoid of any other processing discussed above. This conversion is nothing but the remosaicing phase. Here the incoming stream of 24 bits, consisting of 8 bits of Red, Green, and Blue is sampled and based on the Bayer layout required, in this project - RGGB, is reproduced. Every odd row of RAW image consists of alternating Red and Green and even consists of Green and Blue. Another requirement is the color intensity, which is denoted by the number of bits used to represent the color, the color samples are multiplied by 4 to make them 12 bits.

In Figure 5.7, it can be observed that the signal coming through the signal m_axis_video_tdata[23:0], the output from the CCM block is the input to the bayer_v1_0 block, which is a custom Bayer IP Block.

In Figure 5.8 the signal o_data_0[11:0] denotes the final 12-bit output. This is the output from the final Bayer IP core. Here in this example, it can be observed that the CCM output is a 24-bit stream consisting of RGB888. The m_axis_video_tdata[23:0] stream data is 6300ff, a hex number, with the first two MSBs denoting red, the next two, green, and the last two LSBs denoting blue. This is extracted and the resulting output at o_data_0[11:0] alternating between 37C hex number and 000 is the odd row with alternating red and green channels respectively. The linePixelCounter[31:0] represents each row number of the image. Within this example it is 0xd7, representing the odd row. The state[1:0] in state 1 represents that the data transfer has been enabled.

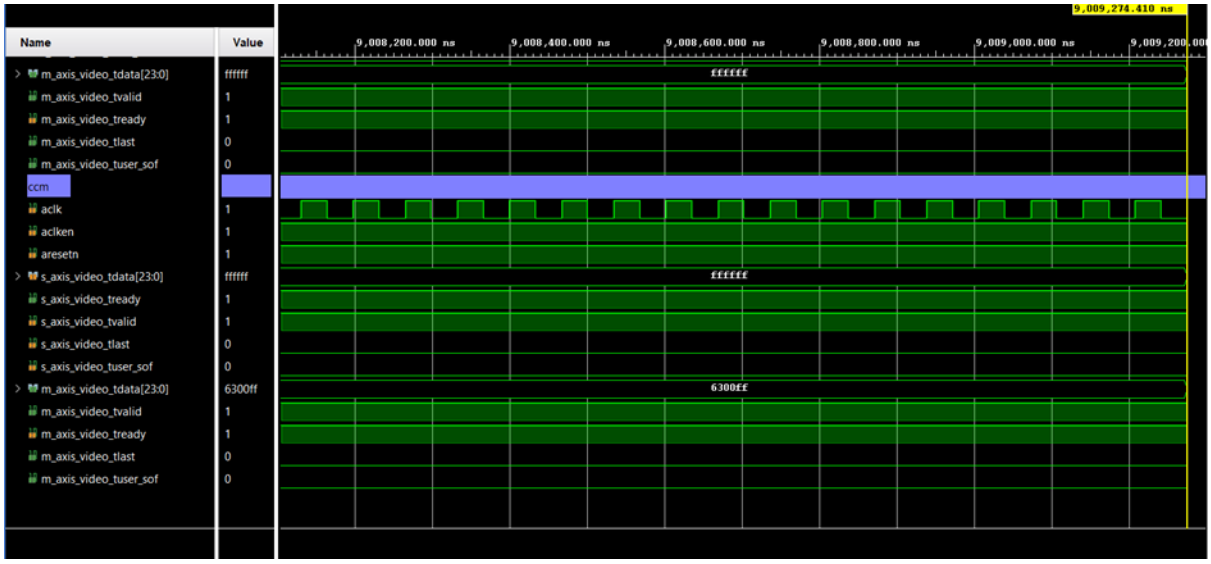

Figure 5.7: instace of stream data showing the CCM output m_axis_video_tdata[23:0]

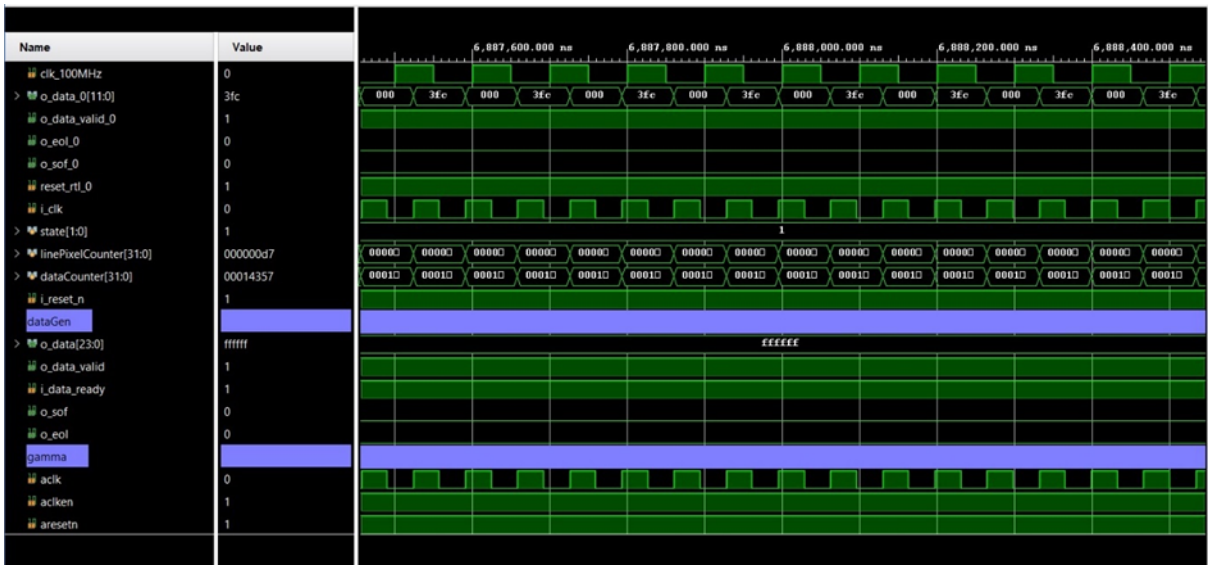

Figure 5.8: Simulated instance of the stream data with the final output o_data_0[11:0]

6 General Discussion

While the algorithm seems to work correctly in MatLab, it was not possible to reproduce the same in the RTL design. Though one part of the project, the basic RGB to Raw RGGB conversion, worked as expected, a complete solution could not be found. Several issues were faced along the way. The VTPG could not produce any signals and failed to identify certain files during the hardware wrapper phase. After several tries at debugging, it was decided to generate custom data. However, manually developing the whole design without optimized IP cores used up more resources than were available. The tone mapping, when coded on Vitis, involving the sine function was overshooting the available registers by almost 37% as it generates a Look-Up-Table for the sine function for all value ranges, taking up more memory space. The CCM involves a lot of multipliers and adders which requires a lot of computing resources too.

During the project, one of the other students, working on the output stage figured out that he could not program his GMSL2 design on the other FPGA on the MpSoC, which was decided initially, and had to move the code to this chip. This considerably reduced the available resources.

7 Conclusion and Future work

This thesis aimed to develop a proof-of-concept simulation framework for hardware-in-the-loop (HIL) testing, specifically focusing on injecting camera images and videos into the ECU of an automotive embedded system. The primary objective was to translate 8-bit sRGB data to 12-bit RGGB raw images by reversing the conventional image sensor processing pipeline.

The project successfully demonstrated a partial solution to this complex problem through extensive research and development. The implementation of the algorithm in MATLAB confirmed its feasibility, while the Vivado-based design highlighted the challenges and resource constraints of hardware implementation. Despite these challenges, the project achieved its initial goal of converting 8-bit RGB images to 12-bit RAW format, providing a foundational framework for future enhancements.

However, the project faced several limitations, including the complexity of implementing an inverse tone mapping function and the resource-intensive nature of color correction matrix operations. These limitations underline the need for further optimization and research to achieve a fully integrated and efficient solution.
In general, having access to the raw image gives better control over further image processing. However, there are only a few papers that discussed obtaining a raw image through a processed image and it required pre-existing knowledge about the specific camera properties. This effectively reduced the usability of several algorithms and datasets. Few papers, for example, [25, 38, 39], suggest employing the Machine Learning model to fasten the process, where the inference can run on the FPGA. The biggest hurdle for this is again the lack of sufficient Raw data for training. There are few datasets available, but they need to be processed further as this project not just requires conversion but also bit-width manipulation. Another issue with this is that ML models use ML-specific frameworks which cannot be directly used in Vivado or Vitis.

The above problem is not an impossible task but would require lots of time to sift through lots of data to create the necessary dataset and to convert the ML code to C/C++ to make it compatible with Vitis.

The project proved to be complex with broad goals to be achieved. Even after dividing the thesis among three, the solution required was huge to be implemented. It requires more time and knowledge about the system to complete the project. During a meeting with the company currently providing the IU solution, it was realized that it took their team of expert engineers nearly five years to obtain the final solution, and this speaks about the complexity of the project.

The contributions of this thesis are significant in the context of HIL testing for autonomous driving systems. This work lays the groundwork for more reliable and comprehensive testing of ADS and ADAS technologies by providing a customizable image injection solution. Access to raw image data helps in better image/ video processing in general; hence, more research is being done into it currently. This will lead to better knowledge of critical image properties,

datasets, and algorithms. Given that FPGAs are a cornerstone in Hardware Accelerators, going the AI/ML route to find a solution will provide the best output. Additionally, latency should be considered when concentrating on the throughput.

Appendices

A Block Diagrams

A.1 VTPG System Design

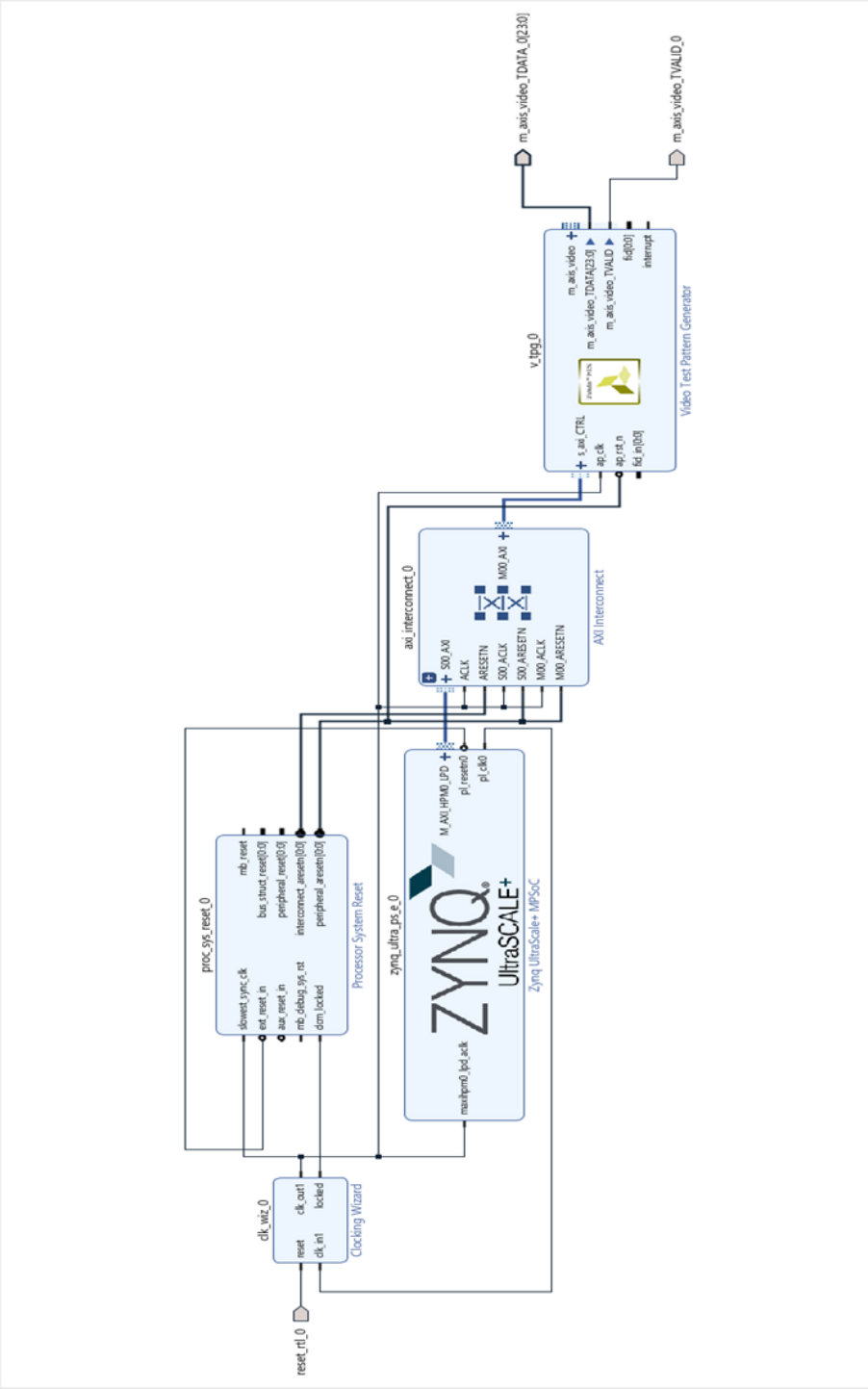

Figure .1: VTPG System Design

A.2 sRGB to RAW12 Design with VTPG

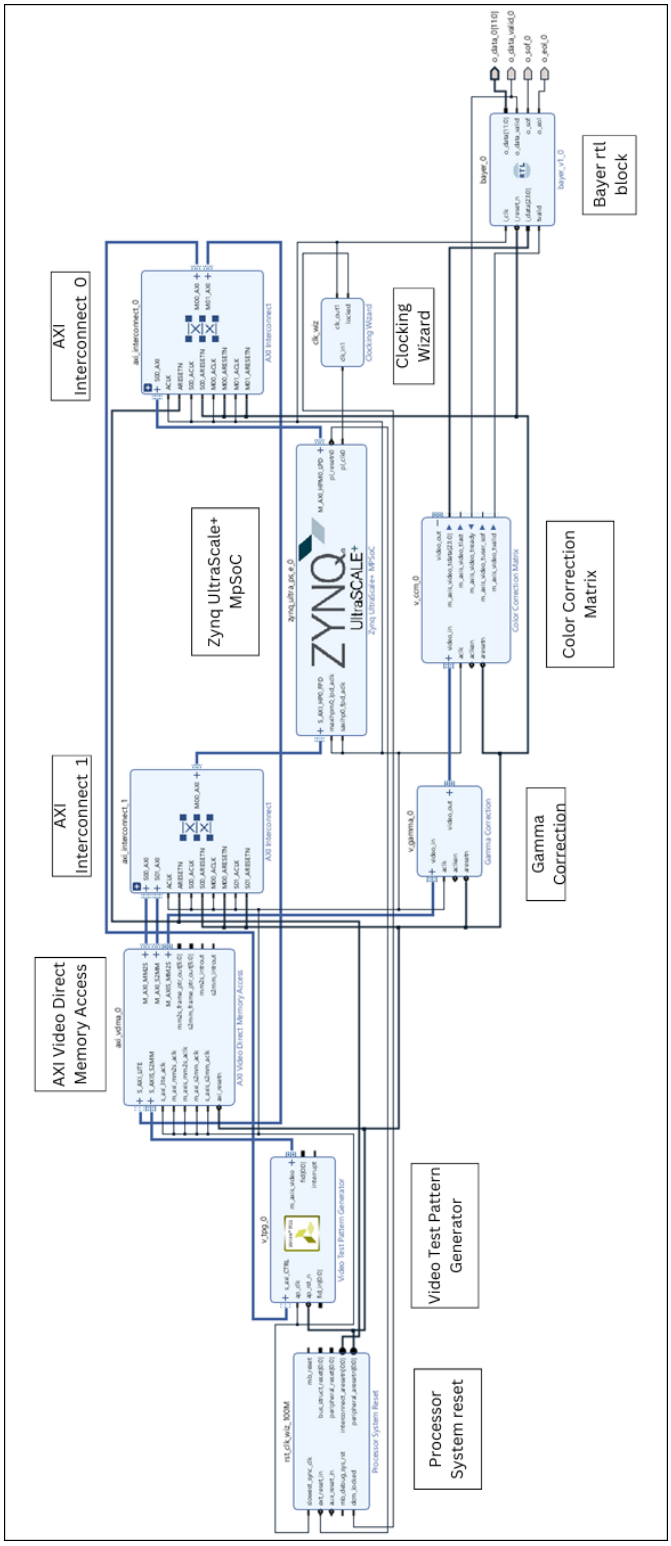

Figure .2: sRGB to RAW12 Design with VTPG

A.3 sRGB to RGB without VTGP and with Datagen

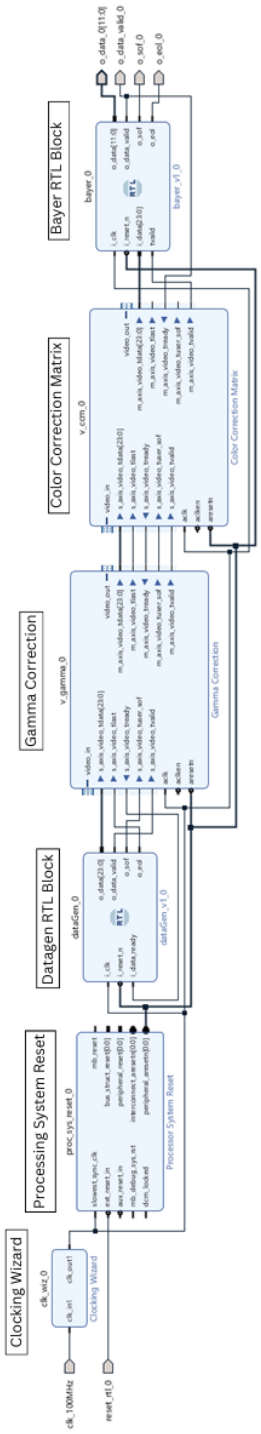

Figure .3: sRGB to RGB without VTGP and with Datagen

B MatLab Code

```
originalImage = imread("Image.jpg");
cropRegion = [1, 1, 1920, 1080];
image = originalImage(cropRegion(2):(cropRegion(2) + cropRegion(4) - 1),
    cropRegion(1):(cropRegion(1) + cropRegion(3) - 1), :);
figure;
imshow(image);
title('8-bit RGB Image');
custom_function(image);

function output_image = custom_function(image)
    % Normalize image values to the range [0, 1]
    image = double(image) / 255.0;

    % Call unprocess function
    output_image = unprocess(image);

end


function [unimage] = unprocess(image)
    rgb2cam = random_ccm();
    [rgb_gain, red_gain, blue_gain] = random_gains();
    unimage = inverse_smoothstep(image);
    unimage = gamma_expansion(unimage);
    unimage = apply_ccm(unimage, rgb2cam);
    unimage = safe_invert_gains(unimage, rgb_gain, red_gain, blue_gain);
    unimage(unimage < 0.0) = 0.0;
    unimage(unimage > 1.0) = 1.0;
    unimage = mosaic(unimage);

end

function tone_image = inverse_smoothstep(image)
    % Approximately inverts a global tone mapping curve.
        image(image < 0.0) = 0.0;
        image(image > 1.0) = 1.0;
        tone_image = 0.5 - sin(asin(1.0 - 2.0 * image) / 3.0);
        imwrite(tone_image, 'tone_image.jpg');
        figure;
        imshow(tone_image);
        title('Invtone RGB Image');
end

function invGamma = gamma_expansion(unimage)
    invGamma = max(unimage, 1e-8) .^ 2.2;
    imwrite(invGamma, 'invGamma.jpg');
    figure;
    imshow(invGamma);
    title('InvGamma RGB Image');
end

function rgb2cam = random_ccm()
    xyz2cams = [1.0234, -0.2969, -0.2266;
                -0.5625, 1.6328, -0.0469;
                -0.0703, 0.2188, 0.6406];
    num_ccms = size(xyz2cams, 1);
    weights = rand(num_ccms, 1, 1) * (1e8 - 1e-8) + 1e-8;
    weights_sum = sum(weights, 1);
```

```
    xyz2cam = sum(xyz2cams .* weights, 1) / weights_sum;

    % Multiplies with RGB -> XYZ to get RGB -> Camera CCM.
    rgb2xyz = [0.4124564, 0.3575761, 0.1804375;
               0.2126729, 0.7151522, 0.0721750;
               0.0193339, 0.1191920, 0.9503041];
    rgb2cam = xyz2cam * rgb2xyz;

    % Normalizes each row.
    rgb2cam = rgb2cam ./ sum(rgb2cam, 2);
end


function ccm_image = apply_ccm(image, ccm)
    % Applies a color correction matrix.
    shape = size(image); % Get the shape of the original image
    num_pixels = numel(image) / 3; % Calculate the number of pixels (assuming
    image is in RGB format)

    % Reshape the image into a 2D array (rows = number of pixels, columns = 3
    color channels)
    image_reshaped = reshape(image, num_pixels, 3);

    % Transpose the color correction matrix for proper matrix multiplication
    ccm_transposed = ccm';

    % Apply color correction by matrix multiplication
    corrected_image_reshaped = image_reshaped * ccm_transposed;

    % Reshape the corrected image back to its original shape
    ccm_image = reshape(corrected_image_reshaped', [1080, 1920]);
    % Transpose back after reshaping

    imwrite(ccm_image, 'ccm_image.jpg');
    figure;
    imshow(ccm_image);
    title('ccm_image RGB Image');
end

function [rgb_gain, red_gain, blue_gain] = random_gains()
    % Generates random gains for brightening and white balance.
    % RGB gain represents brightening.
    rgb_gain = 1.0 / (0.8 + 0.1 * randn);

    % Red and blue gains represent white balance.
    red_gain = 1.9 + (2.4 - 1.9) * rand;
    blue_gain = 1.5 + (1.9 - 1.5) * rand;
end


function InvGain = safe_invert_gains(image, rgb_gain, red_gain, blue_gain)
    % Inverts gains while safely handling saturated pixels.
    gains = [1.0 / red_gain, 1.0, 1.0 / blue_gain] / rgb_gain;
    gains = reshape(gains, [1, 1, 3]);

    % Prevents dimming of saturated pixels by smoothly masking gains near white
    .
    gray = mean(image, 3);
    inflection = 0.9;
    mask = max(gray - inflection, 0.0) / (1.0 - inflection);
```

```
    mask = mask .^ 2.0;
    safe_gains = max(mask + (1.0 - mask) .* gains, gains); % Element-wise
    multiplication

    % Apply safe gains to the image
    InvGain = image .* safe_gains;

    imwrite(InvGain, 'InvGain.jpg');
    figure;
    imshow(InvGain);
    title('InvGain RGB Image');
end

function remosaicedImage = mosaic(image)
    % Extracts RGGB Bayer planes from an RGB image.
    [~, ~, channels] = size(image);
    assert(channels == 3, 'Input image must have 3 channels (RGB)');

    [height, width, ~] = size(image);
    remosaicedImage = zeros(size(image));

    remosaicedImage(1:2:height, 1:2:width, 1) = image(1:2:height, 1:2:width, 1)
    ;
    remosaicedImage(1:2:height, 2:2:width, 2) = image(1:2:height, 2:2:width, 2)
    ;
    remosaicedImage(2:2:height, 1:2:width, 2) = image(2:2:height, 1:2:width, 2)
    ;
    remosaicedImage(2:2:height, 2:2:width, 3) = image(2:2:height, 2:2:width, 3)
    ;

   % image = cat(3, red, green_red, green_blue, blue);
   % remosaicedImage = reshape(image, [height/2, width/2, 4]);
    imwrite(remosaicedImage, 'remosaicedImage.jpg');
    figure;
    imshow(remosaicedImage);
    title('remosaicedImage RGGB Image');
end
```

C Verilog Codes and wrappers

C.1 Bayer RTL Code

```
`timescale 1ns / 1ps
//
    //////////////////////////////////////////////////////////////////////////////////

// Company: Volvo Cars
// Engineer: Nidhi Chakrabhavi Basavaraju
//
// Create Date: 08/08/2023 02:53:37 PM
// Design Name: bayer
// Module Name: bayer
// Project Name:
// Target Devices: xczu9eg-ffvb1156-1-e
// Tool Versions: Vivado 2019.2
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//
    //////////////////////////////////////////////////////////////////////////////////

`define lineSize 1920
`define frameSize 1920*1080

module bayer(
input    i_clk,
input    i_reset_n,
input    [23:0] i_data,
output reg [11:0] o_data,
output reg o_data_valid,
//input    i_data_ready,
output reg o_sof,
output reg o_eol,
input tvaild

    );

     reg [1:0] state;
     reg row= 0;
localparam   IDLE = 'd0,
             SEND_DATA = 'd1,
             END_LINE = 'd2;

integer linePixelCounter;
integer dataCounter;

always @(*)
if(state == SEND_DATA)
begin

    if(linePixelCounter %2 == 0 && row==0)
    begin
        o_data <= (i_data[23:16])*4;
```

```
53      end
54      else if((linePixelCounter %2 == 1 && row==0)|| (linePixelCounter %2 == 0 &&
        row==1))
55      begin
56          o_data <= (i_data[15:8])*4;
57      end
58      else begin
59          o_data <= (i_data[7:0])*4;
60      end
61  end
62
63   always @(posedge i_clk)
64   begin
65      if(!i_reset_n)
66      begin
67          state <= IDLE;
68          linePixelCounter <= 0;
69          dataCounter <= 0;
70          o_data_valid <= 1'b0;
71          o_sof <= 1'b0;
72          o_eol <= 1'b0;
73      end
74      else
75      begin
76          case(state)
77          IDLE: begin
78          o_sof <= 1'b1;
79          o_data_valid <= 1'b1;
80          state <= SEND_DATA;
81           end
82          SEND_DATA:begin
83              if(tvaild)
84                  begin
85                      o_sof <= 1'b0;
86                      linePixelCounter <= linePixelCounter+1;
87                      dataCounter <= dataCounter+1;
88                  end
89              if(linePixelCounter == `lineSize-2)
90                  begin
91                      o_eol <= 1'b1;
92                      state <= END_LINE;
93                  end
94              end
95              END_LINE:begin
96                  if(tvaild)
97                  begin
98                      o_eol <= 1'b0;
99                      linePixelCounter <= 0;
100                     row <= !row;
101                     dataCounter <= dataCounter+1;
102                 end
103                 if(dataCounter == `frameSize-1)
104                 begin
105                     state <= IDLE;
106                     o_data_valid <= 1'b0;
107                     dataCounter <= 0;
108                 end
109                 else
110                 begin
111                     state <= SEND_DATA;
```

```
112                     end
113                 end
114             endcase
115         end
116
117 end
118 endmodule
```

C.2 NetList

```
1 //Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
2 //
    ----------------------------------------------------------------------------------

3 //Tool Version: Vivado v.2019.2 (win64) Build 2708876 Wed Nov  6 21:40:23 MST
    2019
4 //Date        : Fri Aug 8 04:30:20 2023
5 //Host        : 5CG8505XF7 running 64-bit major release  (build 9200)
6 //Command     : generate_target rawSystem.bd
7 //Design      : rawSystem
8 //Purpose     : IP block netlist
9 //
    ----------------------------------------------------------------------------------

10 `timescale 1 ps / 1 ps
11
12 (* CORE_GENERATION_INFO = "rawSystem,IP_Integrator,{x_ipVendor=xilinx.com,
    x_ipLibrary=BlockDiagram,x_ipName=rawSystem,x_ipVersion=1.00.a,x_ipLanguage=
    VERILOG,numBlks=6,numReposBlks=6,numNonXlnxBlks=0,numHierBlks=0,maxHierDepth
    =0,numSysgenBlks=0,numHlsBlks=0,numHdlrefBlks=2,numPkgbdBlks=0,bdsource=USER
    ,da_board_cnt=3,da_clkrst_cnt=1,synth_mode=OOC_per_IP}" *) (* HW_HANDOFF = "
    rawSystem.hwdef" *)
13 module rawSystem
14    (clk_100MHz,
15     o_data_0,
16     o_data_valid_0,
17     o_eol_0,
18     o_sof_0,
19     reset_rtl_0);
20   (* X_INTERFACE_INFO = "xilinx.com:signal:clock:1.0 CLK.CLK_100MHZ CLK" *) (*
      X_INTERFACE_PARAMETER = "XIL_INTERFACENAME CLK.CLK_100MHZ, CLK_DOMAIN
      rawSystem_clk_100MHz, FREQ_HZ 100000000, INSERT_VIP 0, PHASE 0.000" *) input
       clk_100MHz;
21   output [11:0]o_data_0;
22   output o_data_valid_0;
23   output o_eol_0;
24   output o_sof_0;
25   (* X_INTERFACE_INFO = "xilinx.com:signal:reset:1.0 RST.RESET_RTL_0 RST" *) (*
       X_INTERFACE_PARAMETER = "XIL_INTERFACENAME RST.RESET_RTL_0, INSERT_VIP 0,
      POLARITY ACTIVE_LOW" *) input reset_rtl_0;
26
27   wire [11:0]bayer_0_o_data;
28   wire bayer_0_o_data_valid;
29   wire bayer_0_o_eol;
30   wire bayer_0_o_sof;
31   wire clk_100MHz_1;
32   wire clk_wiz_0_clk_out1;
33   wire [23:0]dataGen_0_o_data;
34   wire dataGen_0_o_data_valid;
35   wire dataGen_0_o_eol;
36   wire dataGen_0_o_sof;
37   wire [0:0]proc_sys_reset_0_peripheral_aresetn;
38   wire reset_rtl_0_1;
39   wire [23:0]v_ccm_0_m_axis_video_tdata;
40   wire v_ccm_0_m_axis_video_tvalid;
41   wire v_ccm_0_s_axis_video_tready;
42   wire [23:0]v_gamma_0_m_axis_video_tdata;
43   wire v_gamma_0_m_axis_video_tlast;
```

```
44  wire v_gamma_0_m_axis_video_tuser_sof;
45  wire v_gamma_0_m_axis_video_tvalid;
46  wire v_gamma_0_s_axis_video_tready;
47
48  assign clk_100MHz_1 = clk_100MHz;
49  assign o_data_0[11:0] = bayer_0_o_data;
50  assign o_data_valid_0 = bayer_0_o_data_valid;
51  assign o_eol_0 = bayer_0_o_eol;
52  assign o_sof_0 = bayer_0_o_sof;
53  assign reset_rtl_0_1 = reset_rtl_0;
54  rawSystem_bayer_0_0 bayer_0
55       (.i_clk(clk_wiz_0_clk_out1),
56        .i_data(v_ccm_0_m_axis_video_tdata),
57        .i_reset_n(proc_sys_reset_0_peripheral_aresetn),
58        .o_data(bayer_0_o_data),
59        .o_data_valid(bayer_0_o_data_valid),
60        .o_eol(bayer_0_o_eol),
61        .o_sof(bayer_0_o_sof),
62        .tvaild(v_ccm_0_m_axis_video_tvalid));
63  rawSystem_clk_wiz_0_0 clk_wiz_0
64       (.clk_in1(clk_100MHz_1),
65        .clk_out1(clk_wiz_0_clk_out1));
66  rawSystem_dataGen_0_0 dataGen_0
67       (.i_clk(clk_wiz_0_clk_out1),
68        .i_data_ready(v_gamma_0_s_axis_video_tready),
69        .i_reset_n(proc_sys_reset_0_peripheral_aresetn),
70        .o_data(dataGen_0_o_data),
71        .o_data_valid(dataGen_0_o_data_valid),
72        .o_eol(dataGen_0_o_eol),
73        .o_sof(dataGen_0_o_sof));
74  rawSystem_proc_sys_reset_0_0 proc_sys_reset_0
75       (.aux_reset_in(1'b1),
76        .dcm_locked(1'b1),
77        .ext_reset_in(reset_rtl_0_1),
78        .mb_debug_sys_rst(1'b0),
79        .peripheral_aresetn(proc_sys_reset_0_peripheral_aresetn),
80        .slowest_sync_clk(clk_wiz_0_clk_out1));
81  rawSystem_v_ccm_0_0 v_ccm_0
82       (.aclk(clk_wiz_0_clk_out1),
83        .aclken(1'b1),
84        .aresetn(proc_sys_reset_0_peripheral_aresetn),
85        .m_axis_video_tdata(v_ccm_0_m_axis_video_tdata),
86        .m_axis_video_tready(1'b1),
87        .m_axis_video_tvalid(v_ccm_0_m_axis_video_tvalid),
88        .s_axis_video_tdata(v_gamma_0_m_axis_video_tdata),
89        .s_axis_video_tlast(v_gamma_0_m_axis_video_tlast),
90        .s_axis_video_tready(v_ccm_0_s_axis_video_tready),
91        .s_axis_video_tuser_sof(v_gamma_0_m_axis_video_tuser_sof),
92        .s_axis_video_tvalid(v_gamma_0_m_axis_video_tvalid));
93  rawSystem_v_gamma_0_0 v_gamma_0
94       (.aclk(clk_wiz_0_clk_out1),
95        .aclken(1'b1),
96        .aresetn(proc_sys_reset_0_peripheral_aresetn),
97        .m_axis_video_tdata(v_gamma_0_m_axis_video_tdata),
98        .m_axis_video_tlast(v_gamma_0_m_axis_video_tlast),
99        .m_axis_video_tready(v_ccm_0_s_axis_video_tready),
100       .m_axis_video_tuser_sof(v_gamma_0_m_axis_video_tuser_sof),
101       .m_axis_video_tvalid(v_gamma_0_m_axis_video_tvalid),
102       .s_axis_video_tdata(dataGen_0_o_data),
103       .s_axis_video_tlast(dataGen_0_o_eol),
```

```
104         .s_axis_video_tready(v_gamma_0_s_axis_video_tready),
105         .s_axis_video_tuser_sof(dataGen_0_o_sof),
106         .s_axis_video_tvalid(dataGen_0_o_data_valid));
107 endmodule
```

C.3 Wrapper

```
//Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
//
    ----------------------------------------------------------------------------------

//Tool Version: Vivado v.2019.2 (win64) Build 2708876 Wed Nov  6 21:40:23 MST
    2019
//Date        : Fri Aug 08 04:30:20 2023
//Host        : 5CG8505XF7 running 64-bit major release  (build 9200)
//Command     : generate_target rawSystem_wrapper.bd
//Design      : rawSystem_wrapper
//Purpose     : IP block netlist
//
    ----------------------------------------------------------------------------------

`timescale 1 ps / 1 ps

module rawSystem_wrapper
   (clk_100MHz,
    o_data_0,
    o_data_valid_0,
    o_eol_0,
    o_sof_0,
    reset_rtl_0);
  input clk_100MHz;
  output [11:0]o_data_0;
  output o_data_valid_0;
  output o_eol_0;
  output o_sof_0;
  input reset_rtl_0;

  wire clk_100MHz;
  wire [11:0]o_data_0;
  wire o_data_valid_0;
  wire o_eol_0;
  wire o_sof_0;
  wire reset_rtl_0;

  rawSystem rawSystem_i
       (.clk_100MHz(clk_100MHz),
        .o_data_0(o_data_0),
        .o_data_valid_0(o_data_valid_0),
        .o_eol_0(o_eol_0),
        .o_sof_0(o_sof_0),
        .reset_rtl_0(reset_rtl_0));
endmodule
```

C.4 Datagen

```
`timescale 1ns / 1ps
//
    //////////////////////////////////////////////////////////////////////////////////

// Company: Volvo Cars
// Engineer: Nidhi Chakrabhavi Basavaraju
//
// Create Date: 04/08/2023 03:22:24 PM
// Design Name: dataGen
// Module Name: dataGen
// Project Name:
// Target Devices: xczu9eg - ffvb1156 -1 - e
// Tool Versions: vivado 2019.2
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//
    //////////////////////////////////////////////////////////////////////////////////

`define lineSize 1920
`define frameSize 1920*1080

module dataGen(
input   i_clk,
input   i_reset_n,
output reg [23:0] o_data,
output reg o_data_valid,
input   i_data_ready,
output reg o_sof,
output reg o_eol
    );
 reg [1:0] state;

localparam  IDLE = 'd0,
            SEND_DATA = 'd1,
            END_LINE = 'd2;

integer linePixelCounter;
integer dataCounter;

always @(*)
begin
    if(linePixelCounter >= 0 && linePixelCounter < 1980)
        o_data <= 24'hffffff;

end

 always @(posedge i_clk)
 begin
    if(!i_reset_n)
    begin
        state <= IDLE;
        linePixelCounter <= 0;
```

```
55          dataCounter <= 0;
56          o_data_valid <= 1'b0;
57          o_sof <= 1'b0;
58          o_eol <= 1'b0;
59      end
60      else
61      begin
62          case(state)
63          IDLE: begin
64          o_sof <= 1'b1;
65          o_data_valid <= 1'b1;
66          state <= SEND_DATA;
67           end
68          SEND_DATA:begin
69              if(i_data_ready)
70                  begin
71                      o_sof <= 1'b0;
72                      linePixelCounter <= linePixelCounter+1;
73                      dataCounter <= dataCounter+1;
74                  end
75              if(linePixelCounter == `lineSize-2)
76                  begin
77                      o_eol <= 1'b1;
78                      state <= END_LINE;
79                  end
80              end
81              END_LINE:begin
82                  if(i_data_ready)
83                  begin
84                      o_eol <= 1'b0;
85                      linePixelCounter <= 0;
86                      dataCounter <= dataCounter+1;
87                  end
88                  if(dataCounter == `frameSize-1)
89                  begin
90                      state <= IDLE;
91                      o_data_valid <= 1'b0;
92                      dataCounter <= 0;
93                  end
94                  else
95                  begin
96                      state <= SEND_DATA;
97                  end
98              end
99          endcase
100     end
101
102 end
103 endmodule
```

D Proof for Inverse Tone-mapping Function

To derive the inverse tone mapping function, let's start with the forward tone mapping function and work through the steps to find its inverse:

1. Forward tone-mapping function:
$$f(x) = -2x^3 + 3x^2$$
2. To find the inverse, we need to solve the equation:
$$y = -2x^3 + 3x^2$$
3. Rearrange the equation:
$$2x^3 - 3x^2 + y = 0$$
4. This is a cubic equation. To solve it, we can use trigonometric substitution.
5. Let $x = \cos(\theta)$. Then $x^2 = \cos^2(\theta)$ and $x^3 = \cos^3(\theta)$.
6. Substitute these into the equation:
$$2\cos^3(\theta) - 3\cos^2(\theta) + y = 0$$
7. Use the trigonometric identity: $\cos^3(\theta) = \frac{3}{4}\cos(\theta) + \frac{1}{4}\cos(3\theta)$
8. Substitute this into our equation:
$$2\left[\frac{3}{4}\cos(\theta) + \frac{1}{4}\cos(3\theta)\right] - 3\cos^2(\theta) + y = 0$$
9. Simplify:
$$\frac{3}{2}\cos(\theta) + \frac{1}{2}\cos(3\theta) - 3\cos^2(\theta) + y = 0$$
10. Use another identity: $\cos^2(\theta) = \frac{1}{2}(1 + \cos(2\theta))$
11. Substitute and simplify:
$$\frac{3}{2}\cos(\theta) + \frac{1}{2}\cos(3\theta) - \frac{3}{2}(1 + \cos(2\theta)) + y = 0$$
$$\frac{3}{2}\cos(\theta) + \frac{1}{2}\cos(3\theta) - \frac{3}{2} - \frac{3}{2}\cos(2\theta) + y = 0$$
12. Rearrange:
$$\frac{1}{2}\cos(3\theta) - \frac{3}{2}\cos(2\theta) + \frac{3}{2}\cos(\theta) + y - \frac{3}{2} = 0$$
13. This equation has the form: $A\cos(3\theta) + B\cos(2\theta) + C\cos(\theta) + D = 0$ where $A = \frac{1}{2}$, $B = -\frac{3}{2}$, $C = \frac{3}{2}$, and $D = y - \frac{3}{2}$
14. For this form of equation, the solution is:
$$\theta = \frac{1}{3}\arccos(1 - 2y)$$

15. Remember that we substituted $x = \cos(\theta)$. So the inverse function is:

$$x = \cos\left(\frac{1}{3}\arccos(1-2y)\right)$$

16. To simplify notation, we can write this as:

$$f^{-1}(y) = \cos\left(\frac{1}{3}\arccos(1-2y)\right)$$

17. The reference research paper for this thesis uses the form:

$$f^{-1}(y) = \frac{1}{2} - \sin\left(\frac{\arcsin(1-2y)}{3}\right)$$

18. So, to reach there from our derived inverse, first, let's consider the argument of arccos in our derived function:

$$1 - 2y$$

This is the same as the argument of arcsin in the paper's form.

19. arccos and arcsin related by the identity:

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x)$$

20. Applying this to the derived function:

$$f^{-1}(y) = \cos\left(\frac{1}{3}\left(\frac{\pi}{2} - \arcsin(1-2y)\right)\right)$$

21. Based on the cosine of a difference identity:

$$\cos(A-B) = \cos(A)\cos(B) + \sin(A)\sin(B)$$

22. $A = \frac{\pi}{6}$ and $B = \frac{1}{3}\arcsin(1-2y)$:

$$f^{-1}(y) = \cos\left(\frac{\pi}{6}\right)\cos\left(\frac{1}{3}\arcsin(1-2y)\right) + \sin\left(\frac{\pi}{6}\right)\sin\left(\frac{1}{3}\arcsin(1-2y)\right)$$

23. We know that:

$$\cos\left(\frac{\pi}{6}\right) = \frac{\sqrt{3}}{2} \quad \text{and} \quad \sin\left(\frac{\pi}{6}\right) = \frac{1}{2}$$

24. Substituting these values:

$$f^{-1}(y) = \frac{\sqrt{3}}{2}\cos\left(\frac{1}{3}\arcsin(1-2y)\right) + \frac{1}{2}\sin\left(\frac{1}{3}\arcsin(1-2y)\right)$$

25. Using the identity:

$$\cos\left(\frac{1}{3}\arcsin(x)\right) = \frac{\sqrt{4-3x^2}}{2}$$

26. Applying this to the function:

$$f^{-1}(y) = \frac{\sqrt{3}}{2} \cdot \frac{\sqrt{4 - 3(1-2y)^2}}{2} + \frac{1}{2} \sin\left(\frac{1}{3} \arcsin(1-2y)\right)$$

27. Simplify the first term:

$$\begin{aligned} f^{-1}(y) &= \frac{\sqrt{3}}{4} \sqrt{4 - 3(1 - 4y + 4y^2)} + \frac{1}{2} \sin\left(\frac{1}{3} \arcsin(1-2y)\right) \\ &= \frac{\sqrt{3}}{4} \sqrt{1 + 12y - 12y^2} + \frac{1}{2} \sin\left(\frac{1}{3} \arcsin(1-2y)\right) \end{aligned}$$

28. The term under the square root simplifies to 1:

$$f^{-1}(y) = \frac{\sqrt{3}}{4} + \frac{1}{2} \sin\left(\frac{1}{3} \arcsin(1-2y)\right)$$

29. Finally, simplifying:

$$\begin{aligned} f^{-1}(y) &= \frac{1}{2} - \left(\frac{1}{2} - \frac{\sqrt{3}}{4}\right) + \frac{1}{2} \sin\left(\frac{1}{3} \arcsin(1-2y)\right) \\ &= \frac{1}{2} - \sin\left(\frac{\pi}{6}\right) + \frac{1}{2} \sin\left(\frac{1}{3} \arcsin(1-2y)\right) \\ &= \frac{1}{2} - \sin\left(\frac{\pi}{6} - \frac{1}{3} \arcsin(1-2y)\right) \end{aligned}$$

30. Using the sine of a difference identity and the fact that $\sin(\frac{\pi}{6}) = \frac{1}{2}$, we arrive at:

$$f^{-1}(y) = \frac{1}{2} - \sin\left(\frac{\arcsin(1-2y)}{3}\right)$$

This is the form presented in the paper. The transformation involves several non-trivial trigonometric identities and algebraic manipulations, which is why the final form looks quite different from our initial derivation.

References

[1] "A heritage of safety innovations." [Online]. Available: https://www.volvocars.com/intl/v/car-safety/safety-heritage

[2] "Volvo cars, safe space technology." [Online]. Available: https://www.volvocars.com/in/v/safety/features

[3] "Rohde schwarz, "understanding i2c"." [Online]. Available: https://www.youtube.com/watch?v=CAvawEcxoPUt=14s

[4] "Anton lind, "model, simulation, and injection of camera images/videos to automotive embedded ecu"." [Online]. Available: https://uu.diva-portal.org/smash/get/diva2:1773330/FULLTEXT01.pdf

[5] "Wikipedia, "raw image format"." [Online]. Available: https://en.wikipedia.org/wiki/Raw_image_format

[6] "Rawpedia, "demosaicing"." [Online]. Available: https://rawpedia.rawtherapee.com/Demosaicing

[7] "Arrow, "introduction to bayer filters"." [Online]. Available: https://www.arrow.com/en/research-and-events/articles/introduction-to-bayer-filters

[8] "Wikipedia, "bayer filter"." [Online]. Available: https://en.wikipedia.org/wiki/Bayer_filter

[9] I. Shopovska, A. Stojkovic, J. Aelterman, D. Van Hamme, and W. Philips, ""high-dynamic-range tone mapping in intelligent automotive systems"," *Sensors*, vol. 23, no. 12, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/12/5767

[10] "Image-engineering, "white balance and gain"." [Online]. Available: https://www.image-engineering.de/library/image-quality/factors/1079-white-balance

[11] "Imatext, "color correction matrix(ccm)"." [Online]. Available: https://www.imatest.com/docs/colormatrix/

[12] S. Wolf, "Color correction matrix for digital still and video imaging systems," 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:60914677

[13] "Cambridgeincolor, "understanding gamma correction"." [Online]. Available: https://www.cambridgeincolour.com/tutorials/gamma-correction.htm

[14] R. C. Bilcu and M. Vehvilainen, "A novel tone mapping method for image contrast enhancement," in *2007 5th International Symposium on Image and Signal Processing and Analysis*, 2007.

[15] "Ipol, "image unprocessing: A pipeline to recover raw data from srgb images"." [Online]. Available: https://www.ipol.im/pub/art/2022/438/article.pdf

[16] "Latticesemi, "what is an fpga"." [Online]. Available: https://www.latticesemi.com/en/What-is-an-FPGA

[17] "baesystems, "what is a system-on-a-chip"." [Online]. Available: https://www.baesystems.com/en-us/definition/what-is-a-system-on-a-chip

[18] “reflexces, ”xilinx zynq ultrascale+ mpsoc”.” [Online]. Available: https://www.reflexces.com/modules/xilinx-zynq-ultrascale-mpsoc

[19] “Doulos, ”rtl coding”.” [Online]. Available: https://www.doulos.com/knowhow/vhdl/rtl-coding/

[20] “Xilinx, “rtl-to-bitstream design flow”.” [Online]. Available: https://docs.xilinx.com/r/en-US/ug892-vivado-desig n-flows-overview/RTL-to-Bitstream-Design-Flow

[21] “Xilinx, ”intellectual property”.” [Online]. Available: https://www.xilinx.com/products/intellectual-property.html

[22] “Xilinx, ”axi basics 1 - introduction to axi”.” [Online]. Available: https://support.xilinx.com/s/article/1053914language=en_US

[23] “Xilinx, ”amba axi4 interface protocol”.” [Online]. Available: https://www.xilinx.com/products/intellectual-property/axi.html

[24] “Mohammad s. sadri, ”zynq training - session 01 - what is axi?”.” [Online]. Available: https://www.youtube.com/watch?v=0Dt8rWJdiJo

[25] T. Brooks, B. Mildenhall, T. Xue, J. Chen, D. Sharlet, and J. T. Barron, “Unprocessing images for learned raw denoising,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[26] S. R. Tobias Plötz, “Benchmarking denoising algorithms with real photographs,” *CVPR*, 2017. [Online]. Available: https://download.visinf.tu-darmstadt.de/papers/2017-cvpr-ploetz-benchmarking_denoising_algorithms-preprint.pdf

[27] “Xilinx, ”zynq ultrascale+ mpsoc”.” [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[28] “Trenz electronic, ”teb0911 ultrarack+ mpsoc board with amd zynq ultrascale+ zu9, 6 fmc connectors.”.” [Online]. Available: https://shop.trenz-electronic.de/en/TEB0911-04-9BEX1MA-TEB0911-UltraRack-MPSoC-Board-with-AMD-Zynq-UltraScale-ZU9-6-FMC-Connectors.

[29] “Xilinx, ”logic synthesis”.” [Online]. Available: https://www.xilinx.com/products/designtools/vivado/implementation.html#synthesis

[30] “Xilinx, ”implementation”.” [Online]. Available: https://www.xilinx.com/products/designtools/vivado/implementation.html#implementation

[31] “Xilinx, ”vitis unified software platform documentation: Embedded software development (ug1400).”.” [Online]. Available: https://docs.xilinx.com/viewer/book-attachment/PHwa3sDuk6G_sjX_v9CnHw/4Yl0KRfcZ5H6NcC7s7Ye9g

[32] “Xilinx, ”clocking wizard”.” [Online]. Available: https://www.xilinx.com/products/intellectual-property/clocking_wizard.html

[33] “Xilinx, ”processing system reset module”.” [Online]. Available: https://www.xilinx.com/products/intellectual-property/proc_sys_reset.html

[34] “Xilinx, ”axi interconnect v2.1 logicore ip product guide”.” [Online]. Available: https://docs.amd.com/r/en-US/pg059-axi-interconnect

[35] “Xilinx, ”test pattern generator”.” [Online]. Available: https://www.xilinx.com/products/intellectual-property/tpg.html

[36] “Xilinx, ”gamma lut”.” [Online]. Available: https://www.xilinx.com/products/intellectual-property/v_gamma_lut.html

[37] “Vipin k menon, ” spatial filter”.” [Online]. Available: https://github.com/vipinkmenon/SpatialFilter

[38] Z. Q. Yazhou Xing and Q. Chen, “Cvpr, ”invertible image signal processing”.” [Online]. Available: https://openaccess.thecvf.com/content/CVPR2021/papers/Xing_Invertible_Image_Signal_Processing_CVPR_2021_paper.pdf

[39] J. L. Woohyeok Kim, Geonu Kim, “Cvpr, ” paramisp: Learned forward and inverse isps using camera parameters”.” [Online]. Available: https://openaccess.thecvf.com/content/CVPR2024/papers/Kim_ParamISP_Learned_Forward_and_Inverse_ISPs_using_Camera_Parameters_CVPR_2024_paper.pdf