VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

LOGIC DESIGN PROJECT (CO3091)

Midterm Report

Image Histogram Equalization

Instructor:	Tran Ngoc Thinh	
Students:	Hoang Minh Cam Tu	2353289
	Ngo Gia Bao	2352094

Ho Chi Minh, November, 2025

Table of contents

1 Introduction

1.1 Background

In digital image processing, histogram equalization is one of the most widely used techniques for contrast enhancement. It works by redistributing the pixel intensity values of an image so that they span the entire range of possible brightness levels. This process enhances image details, especially in dark or overly bright regions, making the image more visually balanced and informative.

While histogram equalization is commonly implemented in software environments such as MATLAB or Python, this project focuses on realizing the same functionality in hardware using **Verilog HDL**. Implementing the algorithm in hardware provides advantages in terms of speed, parallelism, and real-time processing capabilities, which are essential for embedded or FPGA-based image processing systems.

1.2 Tools and Environment

Tool / Environment	Platform	Description / Rationale
Verilog HDL	Universal	Core Hardware Description Language used for IP Core implementation.
Vivado 2024.1	Windows	Official Xilinx suite for synthesis, place-and-route, and timing analysis during final FPGA deployment and verification.
Icarus Verilog (iverilog)	macOS / Linux	Command-line simulator for functional verification and debugging on platforms without Vivado support.
GTKWave	macOS / Linux	Waveform viewer used for graphical inspection of simulation results produced by Icarus Verilog.
Yosys	macOS / Linux	Open-source synthesis tool for preliminary logic optimization and netlist generation to ensure design portability.
Visual Studio Code	Universal	Used for preprocessing and postprocessing scripts, including image-to-.mem format conversion for testbench initialization and debugging process.
FPGA Board (Arty-Z7)	Target Hardware	Physical platform for deploying the system and validating SoC integration.

Table 1: Development Tools and Environments used in the Project

1.3 Product Functions

The proposed Histogram Equalization IP Core is designed to be a high-performance hardware accelerator for image enhancement. Below are the key functional features of the system:

1.3.1 Core Image Processing Functions

- **Automatic Contrast Enhancement:** The system analyzes the input 8-bit grayscale image and redistributes the pixel intensities to cover the full 0255 range. This makes hidden details in dark or bright areas much clearer to the human eye.
- **Two-Pass Sequential Architecture:** The architecture ensures accuracy by processing the image in two stages: in Pass 1 (Analysis), the system scans every pixel to build a histogram and calculates the Cumulative Distribution Function (CDF) along with the Lookup Table (LUT); in Pass 2 (Mapping),

the image is re-read and the pre-computed LUT is used to transform each pixel into its new enhanced value.

1.3.2 System and Integration Functions

- **Dual-Interface Control (SoC Ready):** The system integrates two interfaces: the AXI4-Lite interface allows an external CPU such as RISC-V or ARM to control the hardware by sending start signals and monitoring system status, while the AXI4-Stream interface enables high-speed, continuous data transfer between the IP core and memory via DMA, ensuring efficient operation without being limited by slow CPU-driven I/O.
- **Dynamic Resolution Scaling:** Unlike static designs, this system is flexible. Users can configure the total number of pixels (e.g., from small 64x64 images up to 720p HD) through control registers without needing to rewrite the hardware code.
- **Hardware-Optimized Math:** Instead of using resource-heavy floating-point logic, the system implements Fixed-Point Arithmetic with rounding logic. This ensures mathematical precision while maintaining high speed on the FPGA.

1.3.3 Performance Goals

- **High Throughput:** The design is fully pipelined, meaning it can process 1 pixel per clock cycle during both passes, which is essential for real-time video applications.
- **On-Chip Memory Utilization:** By utilizing FPGA Block RAM (BRAM) to store the Histogram and LUT, the system achieves single-cycle memory access, avoiding external memory bottlenecks during processing.

2 Background Knowledge

2.1 Histogram Equalization Algorithm

2.1.1 Mathematical Foundation

Histogram Equalization (HE) is a popular statistical method used to improve the visual appearance of an image. In many cases, images captured in low-light or overexposed conditions have a narrow "dynamic range", meaning most pixels are crowded into a small range of gray levels. HE works by spreading these gray levels across the entire available spectrum (0 to 255 for an 8-bit image), resulting in a higher contrast.

The algorithm follows a step-by-step mathematical procedure to transform the input image into an equalized output.

Step 1: Histogram Calculation First, we count the number of occurrences for each gray level k (where $k \in [0, 255]$). This is represented as:

$$h(k) = n_k$$

where n_k is the number of pixels with intensity k. In our hardware design, this process is handled by the `histogram_accumulator` module.

Step 2: Cumulative Distribution Function (CDF) The CDF represents the cumulative sum of the histogram values up to a specific gray level k. It shows the distribution of pixels below or at a certain intensity. The formula is:

$$\text{CDF}(k) = \sum_{i=0}^{k} h(i)$$

The CDF is the key to mapping old pixel values to new ones. If the CDF is linear, the image has a perfectly uniform distribution.

Step 3: Transformation Function (Mapping) To get the new intensity value s_k for an original intensity k, we normalize the CDF using the total number of pixels (T) and the maximum gray level, which is $L - 1 = 255$ for 8-bit images):

$$s_k = \text{round}\left(\frac{\text{CDF}(k) - \text{CDF}_{\text{min}}}{T - \text{CDF}_{\text{min}}} \times (L-1)\right)$$

In simplified hardware implementations, to save resources, we often use the general form:

$$s_k = \text{round}\left(\frac{\text{CDF}(k)}{T} \times (L-1)\right)$$

The resulting value s_k represents the new, equalized intensity for any pixel that originally had a value of k. By applying this transformation to every pixel in the image, the intensity distribution is effectively "stretched," ensuring that the image utilizes the full dynamic range available.

2.1.2 Probability Perspective

In digital image processing, the distribution of gray levels can be viewed from a probability perspective. Each pixel intensity can be treated as a random variable, and the image histogram essentially represents the Probability Mass Function (PMF). From this viewpoint, the Probability Density Function (PDF) describes the likelihood that a pixel has a specific gray level k, which is obtained by dividing the number of pixels at that level n_k by the total number of pixels T:

$$P(k) = \frac{n_k}{T}$$

For images with low contrast, this probability distribution is typically concentrated within a narrow range of gray levels, resulting in a ”clumped” PDF. The Cumulative Distribution Function (CDF), which is the cumulative sum of these probabilities, explains why gray levels become more spread out after histogram equalization. Mathematically, the CDF is a non-decreasing function that maps the input range $[0, 1]$ to the output range $[0, 1]$. In an ideal high-contrast image, pixel intensities are evenly distributed across all gray levels, forming a uniform distribution. The goal of histogram equalization is to transform the original PDF toward this uniform distribution. By using the CDF as the transformation function, regions with a high concentration of pixels are effectively stretched, while regions with fewer pixels are compressed, resulting in a more evenly distributed set of gray levels and an overall increase in image contrast.

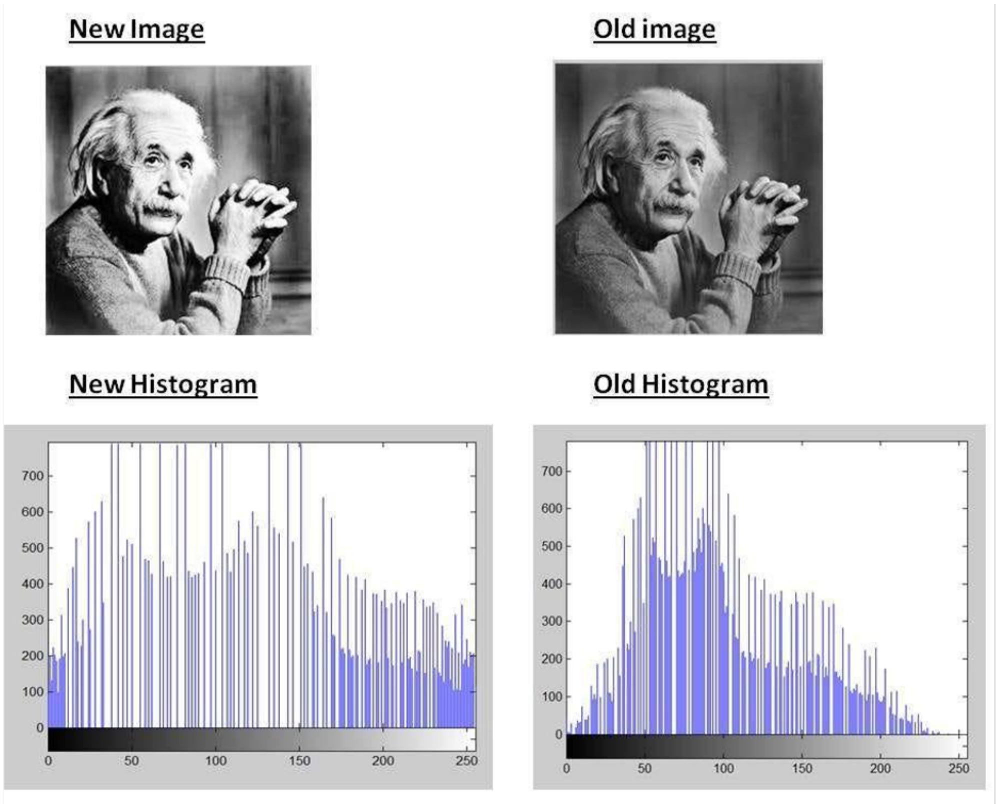


Figure 1: Comparison of original and equalized images with the respective histograms.

2.1.3 Hardware-oriented Mathematics (Fixed-point Arithmetic

When moving from a theoretical algorithm to a Verilog implementation, a major challenge arises from floating-point division. On an FPGA such as the Arty Z7, implementing standard decimal division is both slow and resource-intensive, consuming a large number of logic elements (LUTs). To keep the design efficient, fixed-point arithmetic is used instead. The standard formula for computing the new pixel value is

$$\text{New\_Value} = \frac{\text{CDF}(k) \times 255}{T}$$

In software, the term $1/T$ naturally results in a decimal value, but in hardware, using integer division directly would discard the fractional part, leading to significant rounding errors and a visible posterization effect in the output image. To avoid this issue, the computation is rearranged to preserve precision while remaining hardware-friendly.

First, the bit-width of internal registers is extended (to 40 bits in our implementation) to prevent overflow during multiplication. The multiplication is performed before the division by computing the numerator as $\text{CDF}(k) \times 255$. To achieve proper rounding, a common mathematical trick is applied:

rounding a fraction A/B to the nearest integer can be done using the formula:

$$\left\lfloor \frac{A + B/2}{B} \right\rfloor$$

Following this idea, a rounding factor of $T/2$ is added before the division, resulting in rounded_numerator $=$ $(\text{CDF}(k) \times 255) + (T/2)$. Finally, an integer division by T is performed.

By adding half of the divisor prior to division, the hardware effectively implements rounding to the nearest integer instead of rounding down, significantly improving image quality without the need for a complex floating-point unit.

2.2 AXI DMA and AXI4-Stream Protocol

2.2.1 Overview of AXI Direct Memory Access (AXI DMA)

The AXI DMA is a specialized hardware component (an IP core) designed by AMD to move data at very high speeds between the system memory (like DDR) and other hardware peripherals. In a typical system, the CPU is often too busy with calculations; therefore, we use the AXI DMA to "offload" the heavy task of moving data, which helps the whole system run much faster. The AXI DMA operates using three main interfaces:

- **AXI4-Lite:** A simple interface used by the CPU to send "commands" or check the "status" of the DMA via internal registers.
- **AXI4 Memory Map:** The primary high-speed path used to read from or write to the system memory.
- **AXI4-Stream:** The interface used to talk to our image processing core. It is built for continuous data flow without needing addresses for every single piece of data.

2.2.2 Data Channels: MM2S and S2MM

The AXI DMA works with two independent directions of data movement:

- **MM2S (Memory Map to Stream):** The DMA reads data from the memory and turns it into a "stream" of pixels for our processing unit.
- **S2MM (Stream to Memory Map):** After the image is processed, it comes back as a stream. The DMA then "packs" this stream and writes it back into the memory.

In the project, Direct Register Mode (also called Simple DMA) is used. This is a resource-efficient mode where the CPU only needs to tell the DMA: "Here is the starting address" and "Here is how many bytes to move". Once these registers are written, the DMA handles the rest.

2.2.3 The AXI4-Stream Protocol

The AXI4-Stream protocol is a "point-to-point" protocol, meaning it connects one Transmitter (Source) directly to one Receiver (Destination). Unlike standard memory interfaces, it doesn't use addresses, making it perfect for video or image data where pixels just flow one after another. Key signals that make this work include:

- **TDATA:** The actual payload (in our case, the pixel values).
- **TVALID and TREADY (The Handshake):** This is the most important part of the protocol. A data transfer only happens when the Transmitter says "I have valid data" (TVALID) and the Receiver says "I am ready to take it" (TREADY) in the same clock cycle.

- **TLAST:** This signal marks the end of a "packet". In image processing, we typically use TLAST to signal the end of a frame or the end of the total pixel transfer so the DMA knows when to stop.

Using AXI DMA with AXI4-Stream allows our Histogram Equalization IP to process massive amounts of data (like HD images) without slowing down the CPU. The protocol ensures that every pixel is delivered in the correct order, and the handshake mechanism automatically pauses the data flow if our hardware core is still busy calculating the histogram, preventing any data loss.

3 System Design

Building upon the mathematical foundation from the previous section, this section focuses on the hardware realization of the Histogram Equalization algorithm. Converting theoretical equations into a high-performance digital circuit requires a careful balance between processing speed, resource utilization, and system compatibility. The following sections present the overall architecture, internal functional modules, and control logic that support real-time image enhancement.

3.1 Top-Level System Architecture

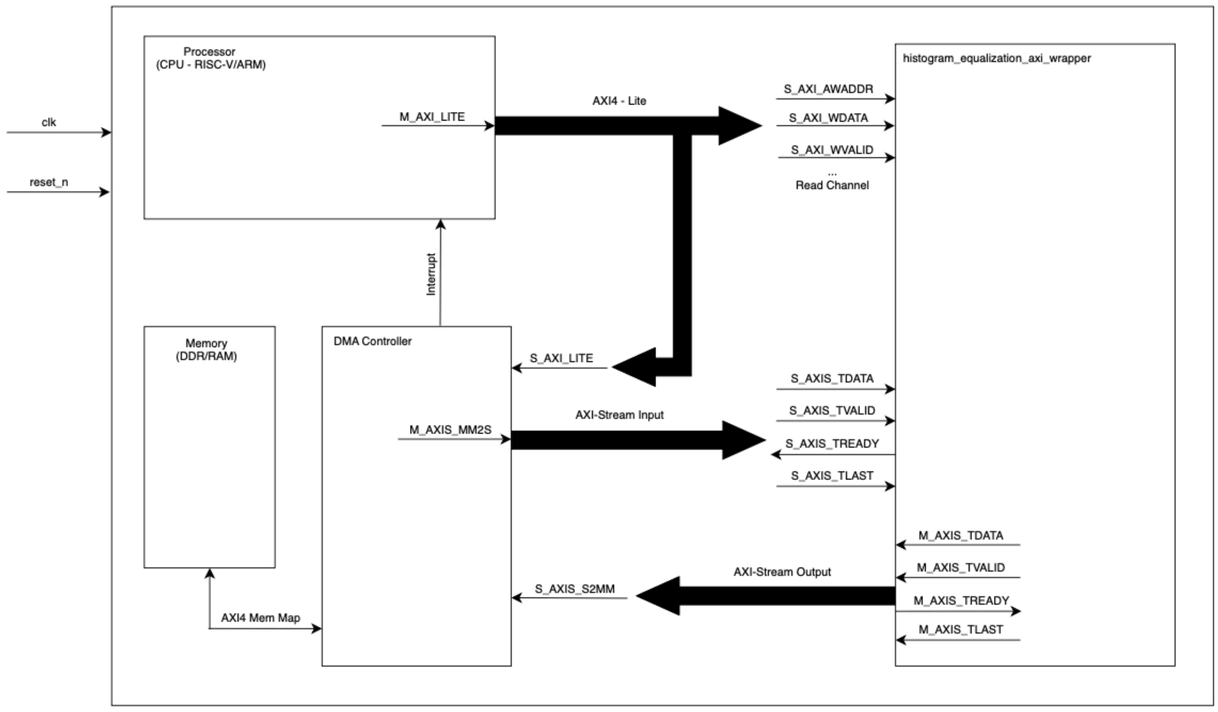


Figure 2: Top-Level Block Diagram of the Histogram Equalization IP Core.

The system is designed as a high-performance hardware accelerator integrated into a System-on-Chip (SoC) environment. Its overall architecture is organized around three main components that work together to achieve efficient, real-time image processing: a Processor (CPU), a Direct Memory Access (DMA) Controller, and a custom Histogram Equalization IP Core `histogram_equalization_axi_wrapper`. Each component has a distinct role within the system, and their interaction is coordinated through standard AXI-based interfaces.

1. **Processor (RISC-V/ARM):** The processor acts as the system master and is responsible for overall system control. It configures both the IP Core and the DMA controller via the AXI4-Lite bus, initiates the histogram equalization process, and monitors status flags to ensure correct operation.
2. **DMA Controller:** The DMA controller handles high-speed data transfers between the external memory (DDR/RAM) and the Histogram Equalization IP Core. By using the AXI4-Stream protocol, it offloads the intensive data movement tasks from the CPU, allowing the processor to focus on control logic while enabling efficient, real-time image processing.
3. **Histogram Equalization IP Core:** This custom hardware block performs the core histogram equalization computation. It receives raw pixel data through the `S_AXIS` interface, processes the data internally, and streams the equalized pixel values back to memory through the `M_AXIS` interface.

3.2 Signal Specifications and Bus Widths for Top-Level Interface

Signal Name	Direction	Width	Type	Description
S_AXI_ACLK	Input	1	Wire	Global clock signal (125 MHz typical).
S_AXI_ARESETN	Input	1	Wire	Active-low system reset.
S_AXI_AWADDR	Input	4	Wire	Write address bus for AXI-Lite control registers.
S_AXI_WDATA	Input	32	Wire	Write data bus for AXI-Lite control registers.
S_AXI_WVALID	Input	1	Wire	Valid signal for AXI-Lite write channel.
S_AXIS_TDATA	Input	8	Wire	Input pixel stream from DMA (MM2S).
S_AXIS_TVALID	Input	1	Wire	Signal indicating valid input data.
S_AXIS_TREADY	Output	1	Wire	Ready signal to apply back-pressure to the DMA.
S_AXIS_TLAST	Input	1	Wire	Indicates the last pixel of an input frame.
M_AXIS_TDATA	Output	8	Wire	Processed pixel stream sent to DMA (S2MM).
M_AXIS_TVALID	Output	1	Wire	Signal indicating valid output data.
M_AXIS_TREADY	Input	1	Wire	Ready signal from the DMA receiver.
M_AXIS_TLAST	Output	1	Wire	Indicates the end of the processed frame.

Table 2: Top-Level Interface Signals of the AXI Wrapper

3.3 Control and Configuration Registers (AXI-Lite)

Register Name	Address	Width	Type	Bit / Field Description
slv_reg0	0x0	32	Reg	Bit [0]: `start_pulse` (Self-clearing start signal).
slv_reg1	0x4	32	Reg	`T_TOTAL_PIXELS` (Configure image size up to 4G pixels).

Table 3: Control and Configuration Registers (AXI-Lite)

3.4 Detailed Functional Block Decomposition

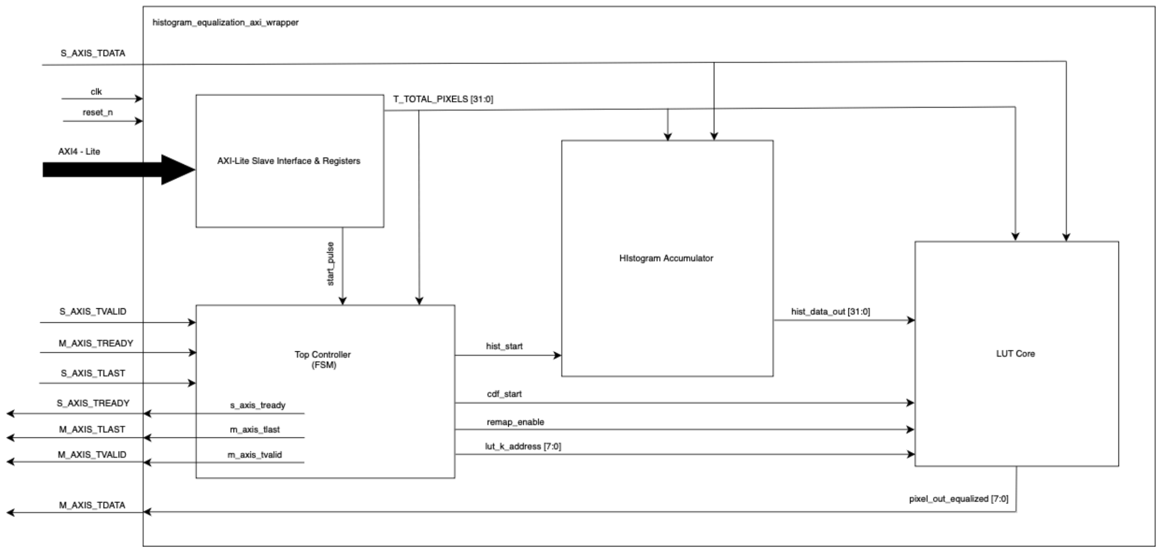


Figure 3: Top-Level Block Diagram of the Histogram Equalization IP Core.

The internal architecture of the `histogram_equalization_axi_wrapper` is modularized to ensure maximum throughput and deterministic control.

3.4.1 AXI-Lite Slave Interface and Registers

This block is responsible for handling low-speed control and configuration communication with the CPU through a memory-mapped AXI4-Lite interface. It provides a set of programmable control registers that allow the processor to configure system parameters and manage the execution flow of the hardware accelerator.

- `T_TOTAL_PIXELS`: A programmable register that specifies the total number of pixels in a frame. This parameter allows the IP Core to support a wide range of image resolutions, from small test images to high-resolution frames, including Full HD and up to 4K resolution.
- `start_pulse`: A single-bit control signal used to trigger the start of the processing sequence. When asserted by the CPU, this signal notifies the top-level controller to begin the histogram equalization operation for the current frame.

3.4.2 Top Controller (FSM)

The Top Controller is implemented as a Finite State Machine (FSM) and acts as the central coordination unit of the entire hardware accelerator. Its primary responsibility is to supervise the two-pass Histogram Equalization workflow while ensuring correct synchronization with the AXI-Stream protocol.

Specifically, the FSM manages the streaming handshaking signals (`TVALID`, `TREADY`, and `TLAST`) to guarantee reliable data transfer between the processing stages. Based on the current processing phase, it generates a set of control signals that activate and synchronize the internal functional modules:

- `hist_start`: Asserted during Pass 1 to enable histogram accumulation from the incoming pixel stream.
- `cdf_start`: Issued after histogram completion to initiate the CDF computation and LUT generation process.
- `remap_enable`: Activated in Pass 2 to allow pixel intensity remapping using the precomputed LUT.
- `lut_k_address`: Generates sequential addresses from 0 to 255, providing indexed access during the LUT calculation phase.

Through this centralized control mechanism, the FSM ensures orderly phase transitions, correct data flow, and deterministic real-time operation of the Histogram Equalization pipeline.

3.4.3 Histogram Accumulator

The Histogram Accumulator performs the statistical analysis required in Pass 1 of the Histogram Equalization algorithm. It processes the incoming pixel stream delivered through `S_AXIS_TDATA` and updates the corresponding intensity bin within its internal histogram memory.

The module is carefully optimized for high-throughput operation, ensuring that each pixel is processed without stalling the data stream:

- **Single-cycle Update:** The frequency counter associated with each gray-level bin is updated within a single clock cycle, enabling a sustained throughput of one pixel per cycle.
- **Data Output:** After the histogram accumulation phase is completed, the module outputs the accumulated bin values as `hist_data_out` [31:0], which are forwarded to the LUT Core for subsequent CDF computation.

This design guarantees accurate histogram construction while fully preserving the real-time streaming performance of the system.

3.4.4 LUT Core

The LUT Core serves as the mathematical processing unit of the Histogram Equalization IP Core. Positioned between the two processing passes, this block is responsible for converting the raw histogram data into a deterministic intensity mapping table.

During the intermediate computation phase, the LUT Core sequentially processes histogram values to construct the Cumulative Distribution Function (CDF). Based on this result, it generates a complete lookup table that defines the transformation from original pixel intensities to equalized values. The main functions of this block are summarized as follows:

- **CDF Calculation:** Histogram bin values are accumulated to form the Cumulative Distribution Function, which represents the cumulative pixel distribution across all intensity levels.
- **Hardware-Friendly Mapping:** The core implements the fixed-point division and rounding scheme described in Section 2, avoiding costly floating-point operations while ensuring accurate generation of the final 8-bit equalized pixel values.
- **Pixel Remapping:** In Pass 2, the LUT Core operates as a high-speed lookup table. The original pixel value received from `S_AXIS_TDATA` is used directly as the LUT address, allowing the corresponding equalized output `pixel_out_equalized` to be produced with minimal latency.

This approach enables efficient hardware implementation while maintaining both computational accuracy and real-time processing capability.

3.5 Signal Specifications and Bus Widths for Internal Processing and Control

Signal Name	Module	Width	Type	Description
current_state	FSM	2	Reg	Current state of the IP (`IDLE`, `PASS1`, `CDF`, `PASS2`).
addr_counter	FSM	32	Reg	Keeps track of processed pixels ($0 \rightarrow T-1$).
lut_counter	FSM	8	Reg	Counter for iterating through 256 gray levels.
hist_start	FSM	1	Wire	Enable signal for the Histogram Accumulator.
cdf_start	FSM	1	Wire	Trigger signal for the CDF and LUT calculation logic.
remap_enable	FSM	1	Wire	Enable signal for reading the LUT and remapping pixels.
hist	Accumulator	32	Reg Array	256×32-bit BRAM array storing pixel counts.
lut	LUT Core	8	Reg Array	256×8-bit BRAM array storing mapping values.
cdf_reg	LUT Core	32	Reg	Accumulator register for calculating the CDF sequentially.
numerator_comb	LUT Core	40	Wire	Result of $\text{CDF}(k) \times 255$ (bit-extended to prevent overflow).

Table 4: Internal Processing and Control Signals

3.6 Design Considerations and Optimization

To ensure system robustness and scalability toward high-resolution image processing, several deliberate design decisions were made regarding bus widths and data type selection. These choices aim to balance hardware safety, future extensibility, and computational accuracy.

1. **32-bit Histogram Accumulators:** This design adopts 32-bit registers for all histogram bins as well as the total pixel counter. This conservative sizing guarantees safe operation for significantly larger image resolutions, including Full HD and 4K frames, where the total pixel count can exceed several million samples. By using 32-bit accumulators, the IP Core can process high-resolution images without any risk of counter overflow, ensuring long-term scalability and reusability of the design.

2. **40-bit Intermediate Calculation (`numerator_comb`):** During the LUT generation phase, the CDF value must be multiplied by the scaling factor 255 to map the cumulative probability into an 8-bit output range. Since the CDF itself is stored as a 32-bit value, this multiplication can theoretically produce a result of up to 40 bits $(32+8)$. To preserve numerical correctness, a bit-extension technique is employed by declaring a 40-bit wire for this intermediate product. This approach prevents arithmetic overflow and maintains the precision required for accurate histogram equalization, especially when operating on large image frames such as 4K resolutions.

3. **Usage of `reg` vs. `wire`:** Signal data types in the design are carefully assigned according to their hardware behavior:

   - `wire`: Used for purely combinational logic, direct signal connections, and input/output ports that do not need to retain values across clock cycles.

   - `reg`: Used for sequential logic elements that must store state information, including Finite State Machine (FSM) registers, address and pixel counters, as well as the Block RAM (BRAM) arrays that implement the Histogram and LUT memories.

This disciplined separation between combinational and sequential data types improves code clarity, synthesizability, and overall design reliability.

4 Implementation

4.1 Overview of the Implementation

The hardware design for the **Image Histogram Equalization** system was implemented using **Verilog HDL**. The design logic is encapsulated within a top-level **AXI** wrapper, allowing it to function as a standard peripheral within a larger **System-on-Chip (SoC)** environment. The implementation focuses on maximizing throughput using a pipelined data path while minimizing resource usage by leveraging on-chip **Block RAM (BRAM)** for histogram and **Lookup Table (LUT)** storage.

4.2 Memory Interface and Data Flow

To ensure compatibility with standard **DMA (Direct Memory Access)** controllers, the design implements the **AXI4** protocol, defined in the wrapper module `histogram_equalization_axi_wrapper`.

- **AXI4-Lite Interface:** This interface is used for configuration. The host processor writes the total number of pixels to register `slv_reg1` and asserts the start bit in `slv_reg0`.
- **AXI4-Stream Interface:** Image data is transferred via high-speed streaming.
  - **Input (S_AXIS):** Receives raw pixels from memory. The `axis_push_ok` signal ensures processing occurs only when valid data is present and the core is ready.
  - **Output (M_AXIS):** Transmits processed pixels back to memory. The logic ensures that `m_axis_tvalid` is synchronized with the processing pipeline.

4.3 Control Flow and FSM Operation

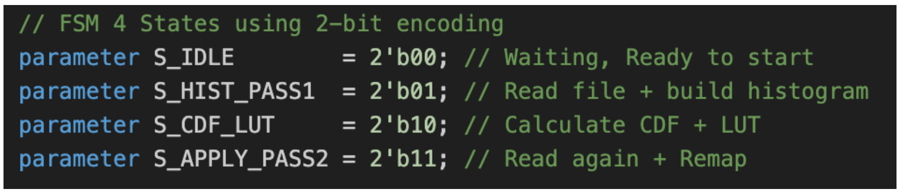
The system operation is governed by a **Finite State Machine (FSM)** with four distinct states, encoded in the `top_controller_fsm` module:

- **S_IDLE:** The system waits for the `start_equalization` signal from the control register.
- **S_HIST_PASS1:** The system accepts the incoming pixel stream. The `s_axis_tready_out` signal is asserted, allowing data to flow into the Histogram Accumulator. An address counter tracks the number of received pixels; once the count reaches `T_TOTAL_PIXELS`, the state transitions.
- **S_CDF_LUT:** The data stream is paused (`s_axis_tready` goes low). The system iterates through the 256 histogram bins to compute the CDF and populate the LUT in the `lut_core`.
- **S_APPLY_PASS2:** The system resumes the data stream. The FSM asserts `remap_enable`, allowing the LUT Core to map incoming pixels to the output stream (`m_axis_tdata`) in real-time.

4.4 Module Integration

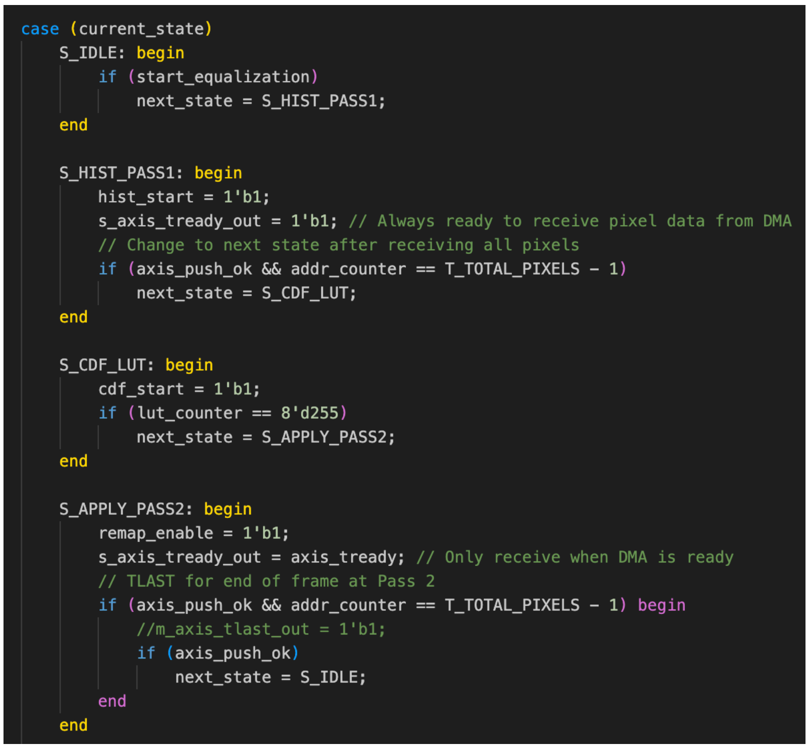
The internal architecture of the system is divided into three primary functional units, coordinated by a central controller. These modules are instantiated within the `histogram_equalization_ip` module.

1. **Top Controller FSM** (`top_controller_fsm.v`): This module serves as the central control unit of the system and is implemented as a Finite State Machine (FSM) with four explicitly defined states using 2-bit binary encoding. The FSM governs the entire processing flow by sequentially switching between the histogram accumulation, CDF/LUT generation, and pixel remapping phases.



```
// FSM 4 States using 2-bit encoding
parameter S_IDLE        = 2'b00; // Waiting, Ready to start
parameter S_HIST_PASS1  = 2'b01; // Read file + build histogram
parameter S_CDF_LUT     = 2'b10; // Calculate CDF + LUT
parameter S_APPLY_PASS2 = 2'b11; // Read again + Remap
```

Figure 4: 2-bit encoding for FSM.

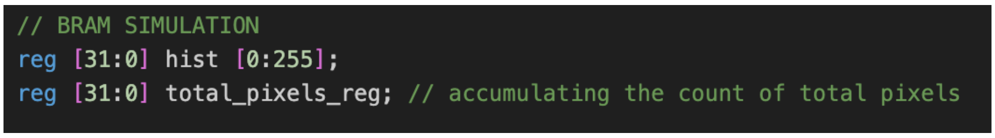


```
case (current_state)
    S_IDLE: begin
        if (start_equalization)
            next_state = S_HIST_PASS1;
    end

    S_HIST_PASS1: begin
        hist_start = 1'b1;
        s_axis_tready_out = 1'b1; // Always ready to receive pixel data from DMA
        // Change to next state after receiving all pixels
        if (axis_push_ok && addr_counter == T_TOTAL_PIXELS - 1)
            next_state = S_CDF_LUT;
    end

    S_CDF_LUT: begin
        cdf_start = 1'b1;
        if (lut_counter == 8'd255)
            next_state = S_APPLY_PASS2;
    end

    S_APPLY_PASS2: begin
        remap_enable = 1'b1;
        s_axis_tready_out = axis_tready; // Only receive when DMA is ready
        // TLAST for end of frame at Pass 2
        if (axis_push_ok && addr_counter == T_TOTAL_PIXELS - 1) begin
            //m_axis_tlast_out = 1'b1;
            if (axis_push_ok)
                next_state = S_IDLE;
        end
    end
```

Figure 5: Case switching between states of FSM.

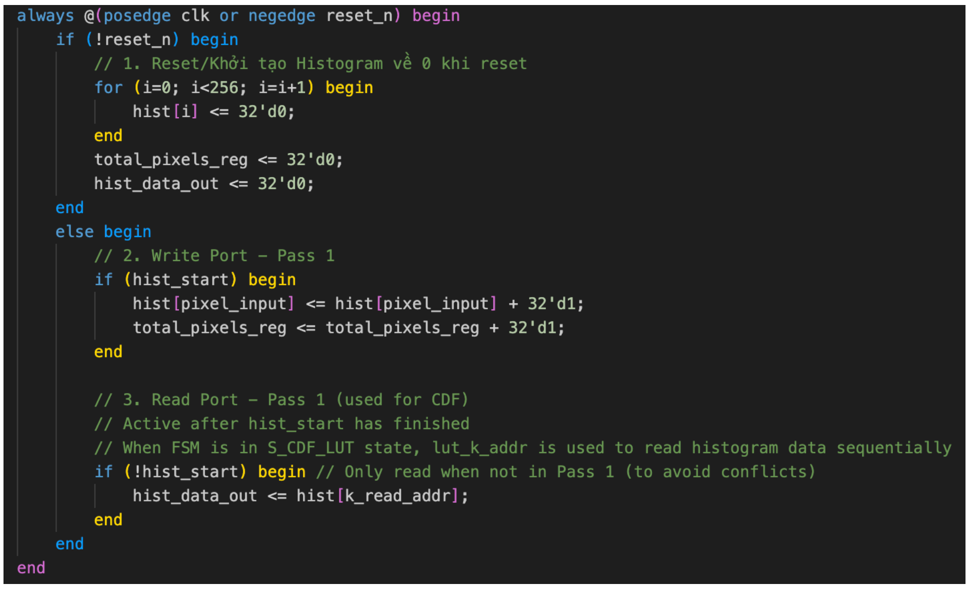
The control logic is realized using a combination of a sequential state register and a combinational `case` statement, which clearly defines the behavior of each state and the corresponding transition conditions. State transitions are primarily driven by AXI-Stream handshaking signals and internal counters.
During each state, the FSM asserts specific internal control signals to activate the appropriate functional modules. In parallel, it manages the AXI-Stream protocol to ensure correct data flow and synchronization with the DMA.

2. **Histogram Accumulator** (`histogram_accumulator.v`): Active during the first pass, this module contains a 256-entry memory array. It reads incoming pixel values and uses them as addresses to increment the corresponding frequency count stored in the internal BRAM. It supports a 32-bit counter width to accommodate high-resolution images without overflow.



```
// BRAM SIMULATION
reg [31:0] hist [0:255];
reg [31:0] total_pixels_reg; // accumulating the count of total pixels
```

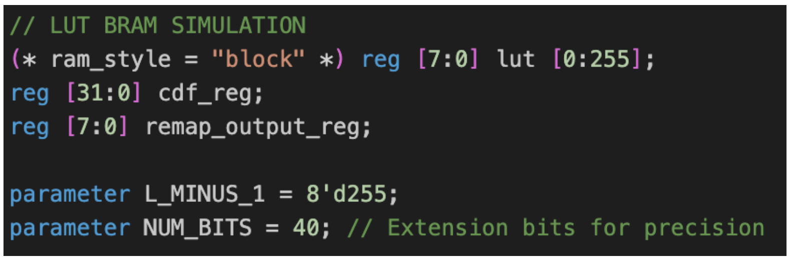
Figure 6: The B-RAM Simulation.



```
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        // 1. Reset/Khởi tạo Histogram về 0 khi reset
        for (i=0; i<256; i=i+1) begin
            hist[i] <= 32'd0;
        end
        total_pixels_reg <= 32'd0;
        hist_data_out <= 32'd0;
    end
    else begin
        // 2. Write Port - Pass 1
        if (hist_start) begin
            hist[pixel_input] <= hist[pixel_input] + 32'd1;
            total_pixels_reg <= total_pixels_reg + 32'd1;
        end

        // 3. Read Port - Pass 1 (used for CDF)
        // Active after hist_start has finished
        // When FSM is in S_CDF_LUT state, lut_k_addr is used to read histogram data sequentially
        if (!hist_start) begin // Only read when not in Pass 1 (to avoid conflicts)
            hist_data_out <= hist[k_read_addr];
        end
    end
end
```

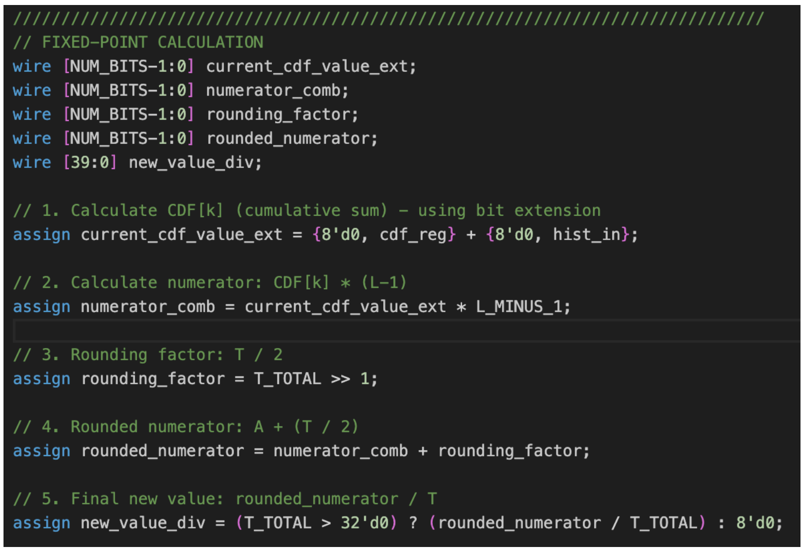
Figure 7: Sequential block for the module.

3. **LUT Core** (`lut_core.v`): This module handles the mathematical computation of the Cumulative Distribution Function (CDF) and the generation of the Lookup Table. It implements fixed-point arithmetic to perform the normalization division $(CDF \times (L-1)/T)$ without requiring floating-point hardware. During the second pass, it acts as a remapper, replacing input pixels with the equalized values stored in its internal RAM.



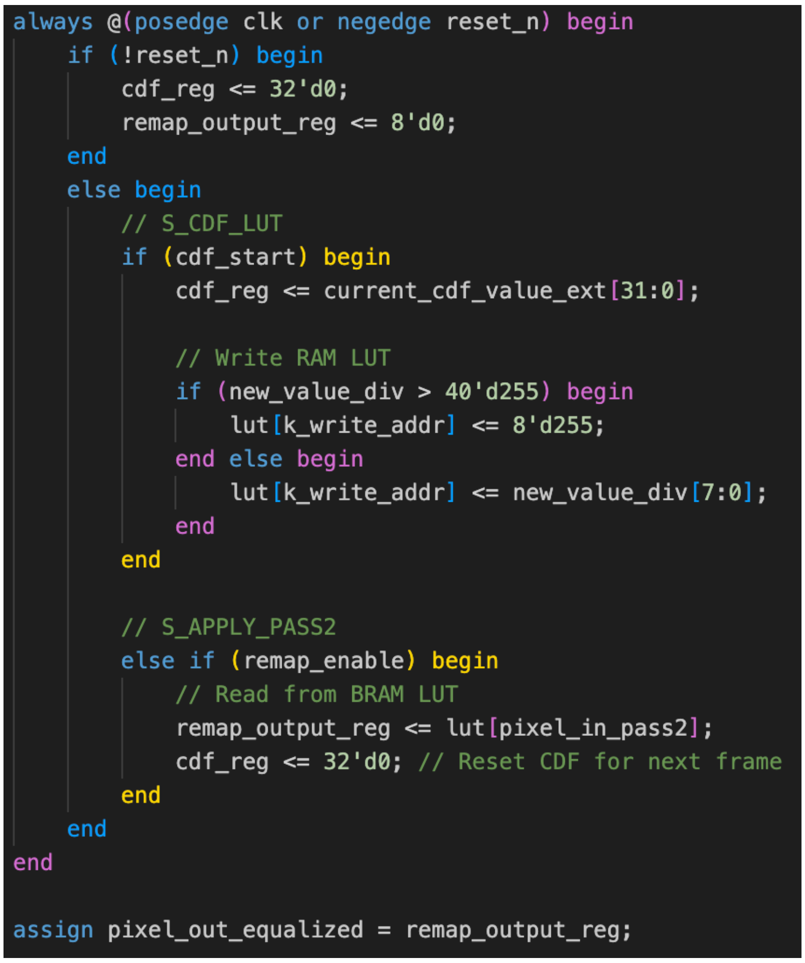
```
// LUT BRAM SIMULATION
(* ram_style = "block" *) reg [7:0] lut [0:255];
reg [31:0] cdf_reg;
reg [7:0] remap_output_reg;

parameter L_MINUS_1 = 8'd255;
parameter NUM_BITS = 40; // Extension bits for precision
```

Figure 8: LUT B-RAM Simulation.



```
//////////////////////////////////////////////////////////////////////////////
// FIXED-POINT CALCULATION
wire [NUM_BITS-1:0] current_cdf_value_ext;
wire [NUM_BITS-1:0] numerator_comb;
wire [NUM_BITS-1:0] rounding_factor;
wire [NUM_BITS-1:0] rounded_numerator;
wire [39:0] new_value_div;

// 1. Calculate CDF[k] (cumulative sum) - using bit extension
assign current_cdf_value_ext = {8'd0, cdf_reg} + {8'd0, hist_in};

// 2. Calculate numerator: CDF[k] * (L-1)
assign numerator_comb = current_cdf_value_ext * L_MINUS_1;

// 3. Rounding factor: T / 2
assign rounding_factor = T_TOTAL >> 1;

// 4. Rounded numerator: A + (T / 2)
assign rounded_numerator = numerator_comb + rounding_factor;

// 5. Final new value: rounded_numerator / T
assign new_value_div = (T_TOTAL > 32'd0) ? (rounded_numerator / T_TOTAL) : 8'd0;
```

Figure 9: Fixed-point Calculation Implementation.



```
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        cdf_reg <= 32'd0;
        remap_output_reg <= 8'd0;
    end
    else begin
        // S_CDF_LUT
        if (cdf_start) begin
            cdf_reg <= current_cdf_value_ext[31:0];

            // Write RAM LUT
            if (new_value_div > 40'd255) begin
                lut[k_write_addr] <= 8'd255;
            end else begin
                lut[k_write_addr] <= new_value_div[7:0];
            end
        end

        // S_APPLY_PASS2
        else if (remap_enable) begin
            // Read from BRAM LUT
            remap_output_reg <= lut[pixel_in_pass2];
            cdf_reg <= 32'd0; // Reset CDF for next frame
        end
    end
end

assign pixel_out_equalized = remap_output_reg;
```

Figure 10: CDF_LUT phase and Remapping.

4.5 Verification Methodology and Testbench Analysis

The functional correctness of the Histogram Equalization IP Core is verified using a comprehensive Verilog testbench. The testbench is designed to simulate a real-world System-on-Chip (SoC) environment by modeling the interaction between a Host Processor, a DMA Controller, and the External Memory.

4.5.1 Simulation Environment and Data Initialization

To ensure that the simulation accurately reflects realistic image processing scenarios, the testbench is configured with the following setup. First, the testbench utilizes the `$readmemh` system task to load a pre-processed hexadecimal memory file (`image_data.mem`) into an internal memory array. This approach enables the IP Core to be verified using real image pixel values rather than randomized or synthetic data.

In addition, an output file (`output_image.txt`) is initialized using the `$fopen` system task to log the equalized pixel values generated during simulation. This file-based logging mechanism allows for post-simulation visual inspection and validation of the image enhancement results.

4.5.2 AXI-Lite Control Sequence (CPU Emulation)

The testbench first emulates the behavior of a CPU by issuing AXI-Lite write transactions to configure the IP Core. During the parameter configuration phase, the testbench writes the `T_TOTAL` value (e.g., 262,144 for a 512×512 image) to address `4’h4` (`slv_reg1`). This register defines the total number of pixels to be processed in a single frame.

After the configuration is completed, the system is triggered by writing `32'h1` to address `4'h0` (`slv_reg0`), which generates the `start_pulse` signal. The testbench continuously monitors the `s_axi_bvalid` signal to ensure that the write transaction has been successfully acknowledged before proceeding to the data streaming phase.

4.5.3 Dual-Pass Streaming Logic (DMA Emulation)

The testbench automates the required two-pass processing flow by emulating the behavior of a DMA controller through the AXI4-Stream protocol.

Pass 1 (Statistical Analysis): During the first pass, the testbench sequentially iterates through the `image_mem` array, asserting `s_axis_tvalid` while supplying pixel data to the IP Core. A strict handshaking mechanism is enforced using the construct `while (!s_axis_tready) @(posedge clk);`, ensuring that pixel data is transmitted only when the IP Core is ready to accept it. The `s_axis_tlast` signal is asserted at the final pixel to explicitly indicate the end of the frame to the Histogram Accumulator.

Inter-Pass Latency: Between the two processing passes, the testbench introduces a controlled delay using `repeat(400) @(posedge clk);`. This latency period is essential to allow the FSM to remain in the `S_CDF_LUT` state long enough to complete the 256-cycle CDF and Lookup Table generation process.

Pass 2 (Pixel Remapping): In the second pass, the testbench streams the same original image data once again. During this phase, the IP Core performs real-time pixel remapping based on the precomputed LUT. Concurrently, a separate `always` block monitors the Master AXI-Stream interface. Whenever both `m_axis_tvalid` and `m_axis_tready` are asserted, the testbench captures the corresponding `m_axis_tdata` value and writes it to the output file for post-simulation analysis.

4.5.4 Verification Features

The testbench also incorporates deterministic checks to ensure overall system reliability and correct protocol operation. First, handshake integrity is verified by continuously monitoring both the `TVALID` and `TREADY` signals. This confirms that the data stream maintains a sustained throughput of one pixel per clock cycle without any data loss or unintended stalls.

In addition, frame completion is validated by detecting the assertion of `m_axis_tlast` on the Master AXI-Stream interface. The presence of this signal indicates that the entire image frame has been fully processed and successfully equalized.

4.6 Simulation Setup and Test Scenarios

To validate the scalability and robustness of the proposed design, simulations were conducted on four distinct image resolutions. For each resolution, Python scripts were employed to preprocess the input images and convert them into memory initialization files (in `.mem` or `.txt` format), which were subsequently used as inputs for the hardware simulation.

After the simulation process, the output data were generated as text-based memory dump files. These files were then post-processed using Python to reconstruct the resulting images, enabling a direct visual and quantitative comparison between the pre-processed input images and the post-processed output images. This workflow ensures a consistent and automated verification pipeline between software-based preprocessing and hardware-level simulation results.

1. **64x64:** Minimal load for basic functional verification.
2. **512x512:** Standard resolution to verify counter widths.
3. **720x720:** Intermediate square resolution.
4. **1024x720:** HD Widescreen resolution to verify the handling of common rectangular video aspect ratios.

5. **2736x1824:** High-resolution test case (approx. 5 megapixels) to stress-test the 32-bit pixel counters and throughput.
6. **3840x2160:** 4K High-resolution test case.

4.7 Simulation Results

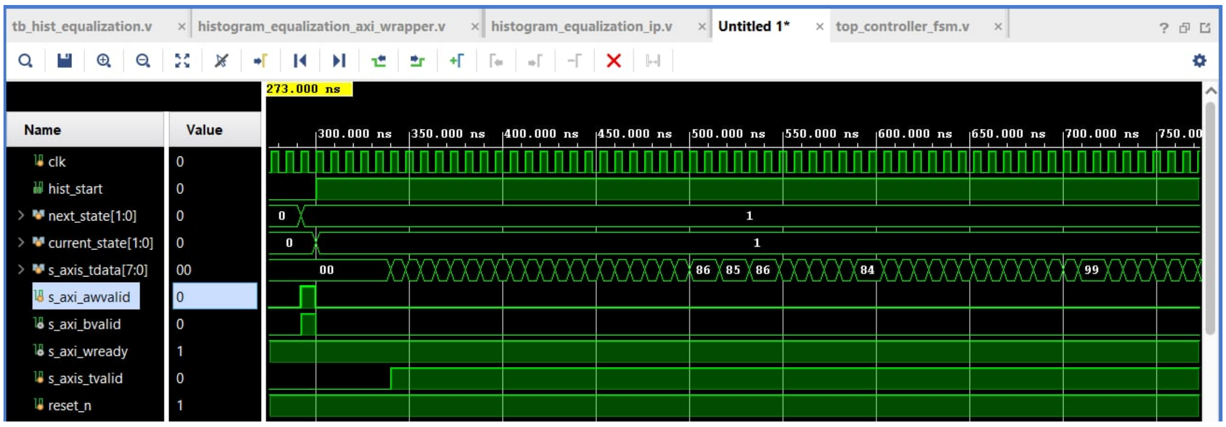


Figure 11: Waveform at the beginning of Pass 1.

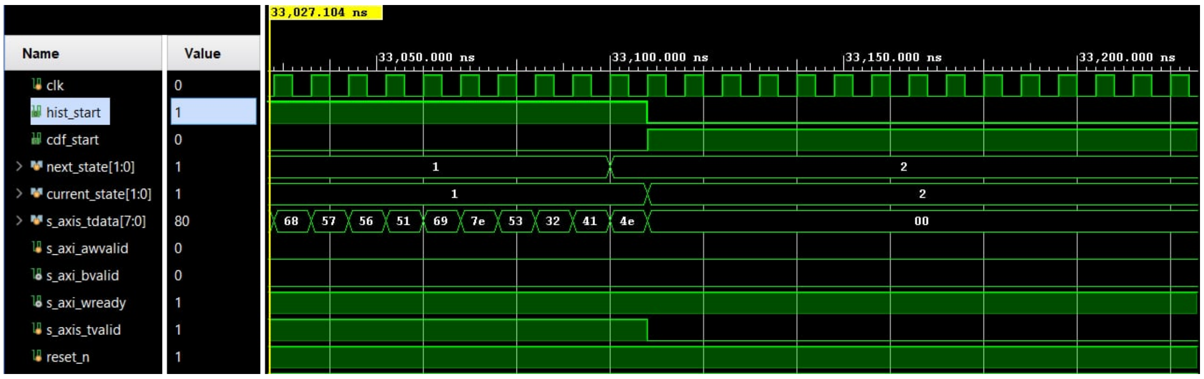


Figure 12: Waveform at the end of Pass1/CDF.

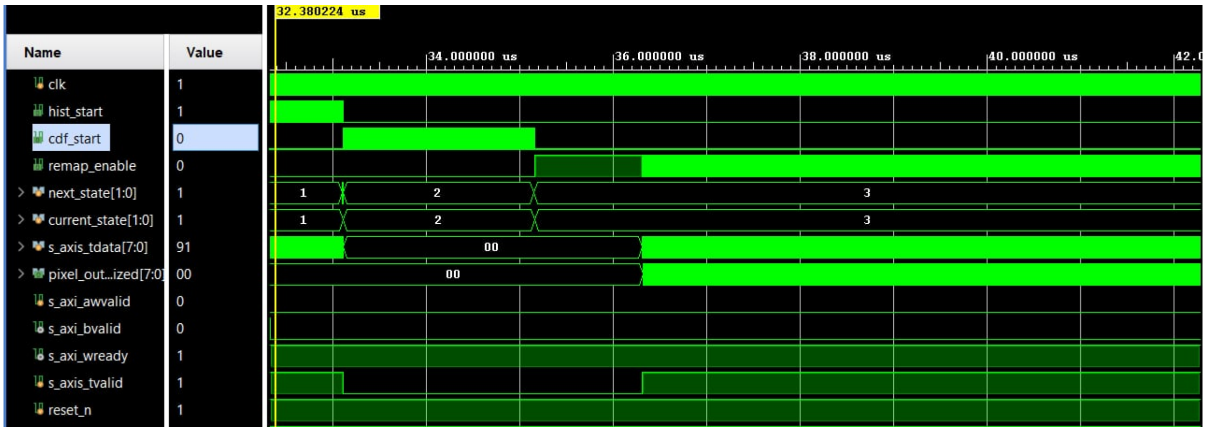


Figure 13: Waveform at the beginning of CDF_LUT calculation.

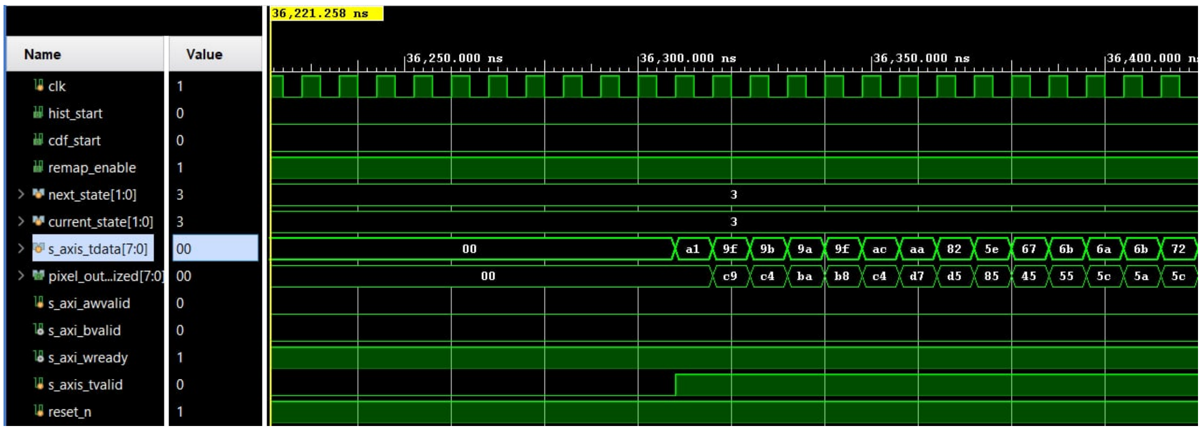


Figure 14: Starting the CDF_LUT calculation.

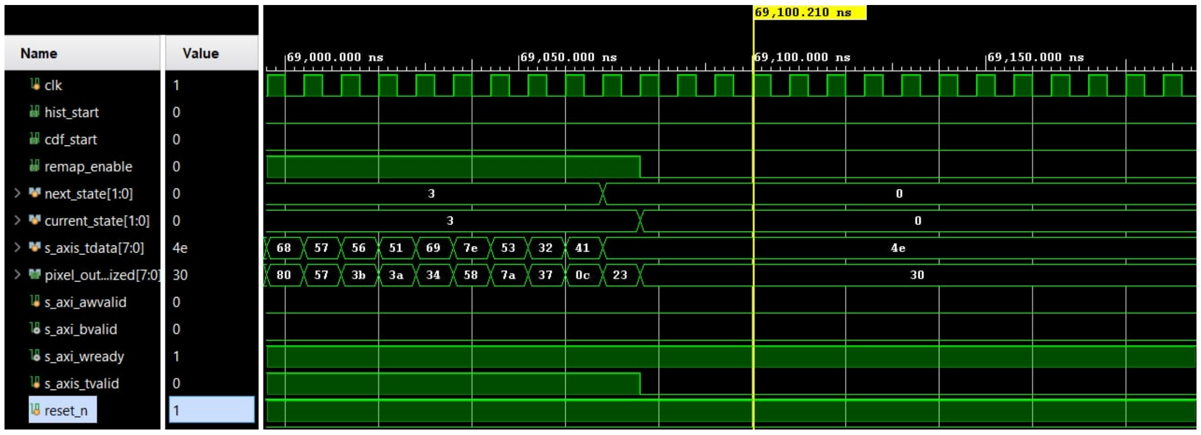


Figure 15: Waveform at the end of a frame.

4.8 Simulation Results for other image resolutions

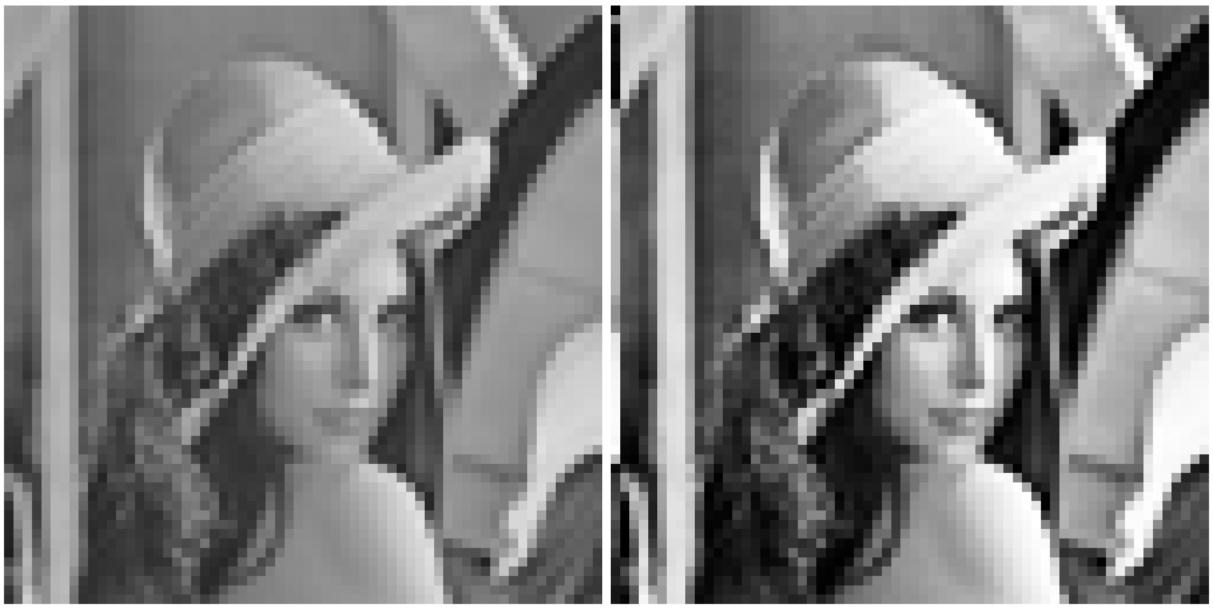


Figure 16: Simulation result for image with 64x64 pixels.

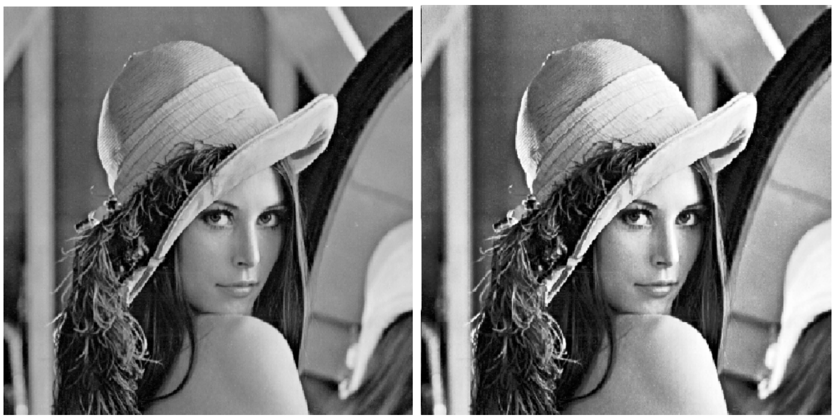


Figure 17: Simulation result for image with 512x512 pixels.

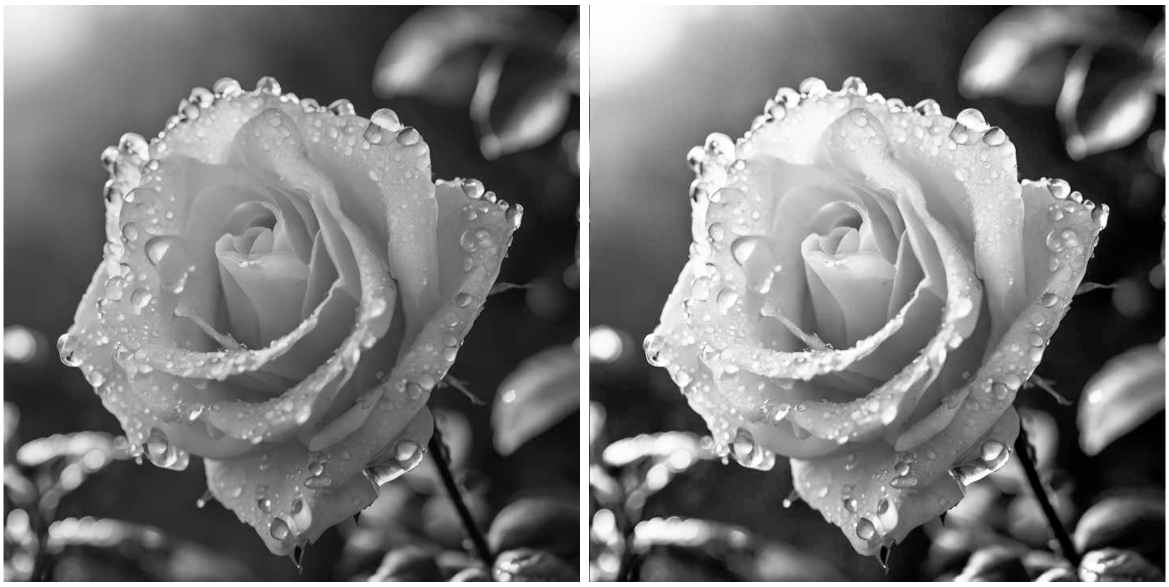


Figure 18: Simulation result for image with 720x720 pixels.

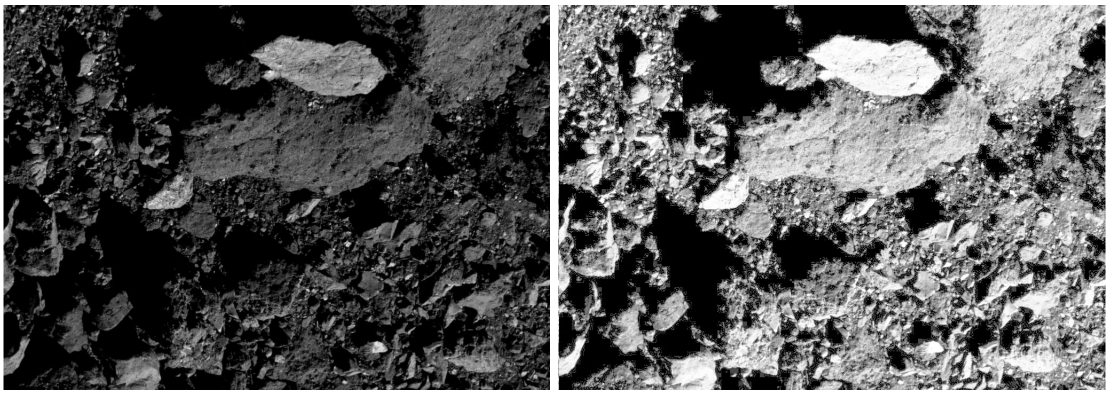


Figure 19: Simulation result for image with 1024x720 pixels (HD image).

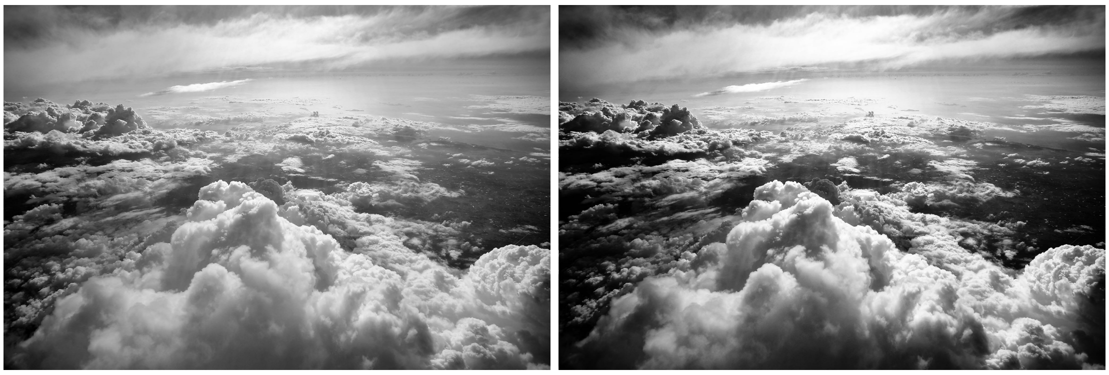


Figure 20: Simulation result for image with 2736x1824 pixels (HD image).

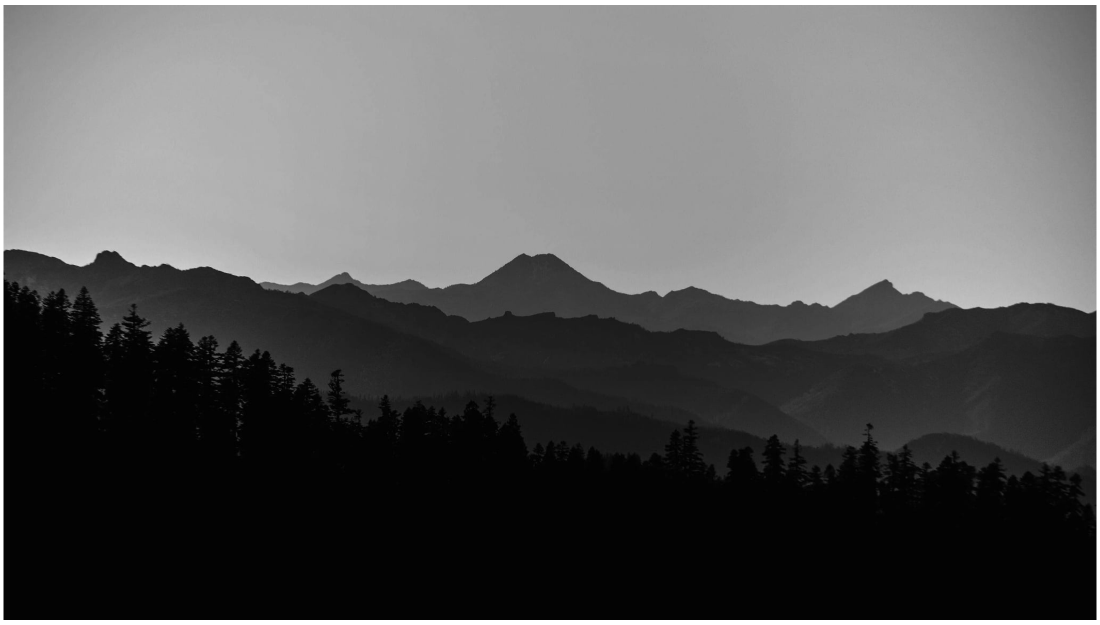


Figure 21: Simulation result for image with 3840x2160 pixels (4K image).

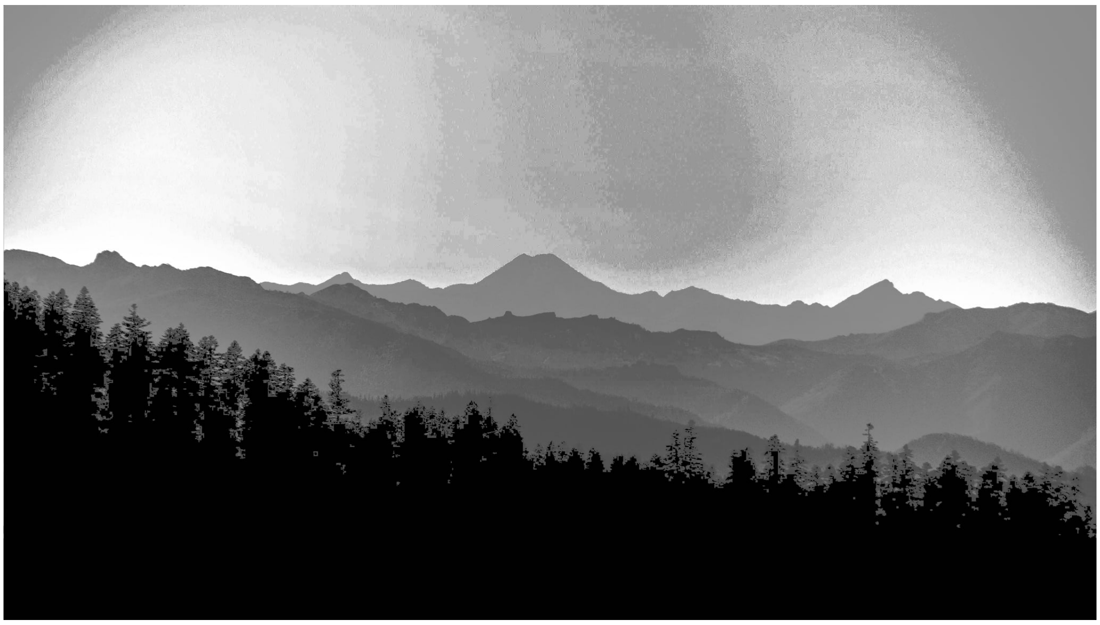


Figure 22: Simulation result for image with 3840x2160 pixels (4K image).

5 Difficulties

During the project implementation, the team encountered several significant environmental and technical hurdles:

1. **Development Environment Constraints:** The lack of native Vivado support on macOS created a workflow bottleneck, as one team member relies on this operating system. This limitation restricted local simulation and debugging capabilities, significantly hampering progress synchronization among team members.

2. **Pixel Misalignment Artifact:** The output image currently exhibits a one-column spatial shift compared to the original. This is an "off-by-one" timing mismatch caused by imprecise latency compensation in the address generation logic regarding the LUT pipelining. While the pixel value transformation is correct, this positional artifact remains unresolved.

3. **System Integration Conflicts:** Implementing advanced features introduced critical conflicts regarding interface timing and memory arbitration. These issues caused instability during high-speed data transfer, forcing the team to limit the scope of the implementation to ensure the reliability of the core processing unit.

4. **Hardware Division Optimization:** The direct use of the Verilog division operator (/) resulted in excessive resource utilization and high latency during synthesis, despite functioning correctly in behavioral simulation. A more optimal hardware solution would require implementing a sequential divider or utilizing a dedicated IP Core.

6 Conclusion

The project successfully designed and implemented a hardware-based Histogram Equalization system using Verilog HDL. Utilizing a modular two-pass architecture coordinated by a central FSM, the system effectively automated the workflow from Histogram computation to Lookup Table (LUT) generation. Simulation results on Vivado across various resolutions, from 64Ư64 to 2736Ư1824, confirmed the functional correctness and the scalability of the pixel counters within the design.

However, the implementation process revealed specific technical challenges. A one-pixel column shift in the output image reflects the complexity of synchronizing pipelined data paths with memory addressing logic. Furthermore, the integration of high-speed data loading features was constrained by interface timing conflicts and development environment limitations . These issues provided valuable insights into the critical distinction between behavioral simulation and synthesizeable hardware design, particularly regarding the resource cost of arithmetic operations .

Future development will prioritize rectifying the alignment artifact by refining the latency compensation within the FSM. To optimize the design for physical deployment, the division logic used for LUT generation requires replacement with sequential dividers or dedicated IP cores to conserve resources . Finally, the system will be completed with a fully compliant AXI-Stream interface supporting DMA to enable real-time video processing on the Arty Z7 FPGA platform .

7 Reference

About FPGA Image Processing:

- R. Gonzalez and R. Woods, Digital Image Processing, Pearson, 4th Ed., 2018.
- R. Woods, J. McAllister, G. Lightbody, and Y. Yi, FPGA-based Implementation of Signal Processing Systems, Wiley, 2017.
- OpenCV Documentation
- AXI DMA_251025_220033
- IHI0051B_amba_axi_stream_protocol_spec

8 Link Github

https://github.com/k4yd133/Logic_Design_Project