BASIC IMAGE PROCESSING USING VERILOG

A Course Based Project Report

B.E. III Semester

submitted By

G. Sai Sri	**1602-24-735-106**
D. Siri Chandana Reddy	**1602-24-735-113**
M. Varshini Satya	**1602-24-735-124**

Subject Name: Digital Logic Design Laboratory

Department of Electronics and Communication Engineering

Vasavi College of Engineering (Autonomous) *ACCREDITED BY NAAC WITH 'A++' GRADE* IBRAHIMBAGH, HYDERABAD-500031

2025-2026

Abstract

Digital image processing is a crucial component of modern embedded systems, particularly in applications involving surveillance, robotics, autonomous machines, and biomedical instrumentation. While software-based solutions such as Python and MATLAB are widely used for image manipulation, hardware-based implementations have become increasingly important due to the need for high-speed, low-latency, and real-time processing.

This project explores the implementation of fundamental image processing operations using **Verilog Hardware Description Language (HDL)**. The primary objective is to design modular, hardware-friendly versions of widely used image-processing techniques—grayscale conversion, brightness control, thresholding, down sampling, contrast stretching, and smoothing—using combinational and sequential logic blocks. Each module processes pixel data in real time, enabling an FPGA or digital logic device to perform image enhancement at hardware speeds.

A pipelined architecture is adopted to ensure smooth flow of pixel data across each processing stage. A 3-bit select line allows dynamic switching between different modules, making the system versatile for demonstrations, testing, debugging, and future extensions. Simulation waveforms generated through Verilog testbenches validate the behaviour of each module, showing precise transformations for multiple pixel values.

The work successfully demonstrates how foundational image preprocessing techniques can be implemented in hardware. It highlights the relevance of digital logic design in real-time embedded vision systems and provides a strong foundation for future FPGA-accelerated image processing tasks.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Project Description

Image processing plays a vital role in various fields of engineering and technology—medical diagnostics, biometric authentication, autonomous navigation, drones, satellite imaging, and even entertainment media. Typically, such operations are performed using software executed on CPUs or GPUs; however, many real-time applications demand faster and more deterministic responses than what general-purpose processors can provide.

Digital logic design enables the realization of computationally efficient image-processing operations at the hardware level. When implemented using Verilog HDL, fundamental image-processing tasks can run in real time, making them highly suitable for FPGA-based or ASIC-based systems.

This project investigates how basic image preprocessing operations can be implemented entirely using hardware logic. The following operations were chosen due to their simplicity and importance:

- Grayscale Conversion
- Brightness Adjustment
- Thresholding
- Down sampling
- Contrast Stretching
- Smoothing (Noise Reduction)

Each operation is designed as an independent module that transforms an incoming pixel based on a predefined logic function. These modules are then integrated into a final top-level design using a pipelined architecture and a multiplexer-based output selection mechanism.

This report elaborates on the theoretical concepts behind each module, design methodology, implementation strategy, and simulation results validating the project's correctness.

1.2 Motivation

Real-time image processing has become a key requirement in:

- Object detection
- Robotics navigation
- Vision-guided systems
- Video analytics
- Surveillance
- Medical imaging devices
- FPGA-based accelerators

Software-based processing introduces latency due to sequential execution and limited parallelism. Using HDL-based logic circumvents such limitations by enabling:

- True hardware-level parallelism
- Low-latency processing
- Deterministic timing regardless of image size
- Customizability for specialized tasks
- Reduced power consumption

This project provides students with a foundational understanding of how image processing can be mapped to hardware logic. It exposes them to the design challenges of creating pixel-level logic circuits and helps bridge the gap between theoretical image processing and practical digital logic implementation.

1.3 Objective of the Project:

The primary objectives guiding this project include:

1. To design hardware modules that perform core image processing tasks using Verilog HDL.
2. To understand pixel-level arithmetic and logic operations.
3. To create a modular, extensible architecture allowing multiple processing techniques in a pipeline.
4. To validate module performance through simulation waveforms and testbench output.
5. To explore how real-time image processing can be executed through digital logic circuits.
6. To build a practical foundation for more advanced FPGA-based image processing systems.

2. LITERATURE SURVEY

Image processing has been extensively studied in both software and hardware domains. Traditional implementations rely on CPUs or GPUs using libraries like OpenCV, MATLAB, and Python's NumPy. These tools provide versatility and ease of development, but fail to meet strict real-time requirements for embedded and industrial applications.

Hardware acceleration in image processing is not new—FPGAs and ASICs have been used since the 1990s. Research literature highlights:

- **Parallelism:** FPGAs allow simultaneous processing of multiple pixels.
- **Low Latency:** Hardware pipelines provide immediate results.
- **Customization:** Circuits can be tailored to specific image-processing tasks.
- **Reliability:** Hardware execution is deterministic and unaffected by system load.
- **Scalability:** Designs can support high-speed video processing.

Many academic papers demonstrate the use of FPGAs in implementing filters, convolution kernels, feature extractors, and even complete vision pipelines. Industry applications include:

- Automotive ADAS systems
- Surveillance cameras
- Industrial automation systems
- FPGA-based biomedical imaging devices
- Drones and robotics vision modules

This project specifically reviews simple pixel-level transformations, but the underlying techniques reflect principles applicable in larger systems.

3. THEORETICAL ANALYSIS

To implement image-processing functions in hardware, it is essential first to understand the mathematical theory behind each operation.

3.1 Pixel Representation and Image Models:

A pixel (picture element) is the smallest unit of a digital image. Pixel representations include:

- **Binary (1-bit)** – Black/White
- **Grayscale (8-bit)** – intensity range 0–255
- **RGB (24-bit)** – Red, Green, Blue components each 8-bit

Binary Representation

All pixel values are interpreted as unsigned binary numbers in digital logic.

Range Management

Operations must ensure values remain within the legal range (0–255). Overflow and underflow must be handled via logical saturation.

3.2 Grayscale Conversion

Grayscale conversion compresses RGB data into a single 8-bit intensity value, using a weighted average representing human perception:

Gray = (0.3 × R) + (0.59 × G) + (0.11 × B)

Hardware-friendly approximation used in this project:

Gray = (30 × R + 59 × G + 11 × B) / 100

This avoids floating-point computation.

3.3 Brightness Adjustment

Brightness control shifts pixel intensity:

Output = Input + Offset

If output < 0 → clamp to 0

If output > 255 → clamp to 255

This is directly achievable using signed arithmetic in Verilog.

3.4 Thresholding

Thresholding converts grayscale images into binary images:

Pixel ≥ Threshold → 255

Pixel < Threshold → 0

This method is widely used in OCR, edge marking, segmentation, etc.

3.5 Down sampling

To reduce resolution or pixel frequency, a counter-based mechanism is used:

- Keep every Nth pixel
- Skip intermediate ones
- Reduces computational load

Verilog uses rising-edge counters for this purpose.

3.6 Contrast Stretching

Contrast is enhanced by mapping a narrow intensity range (MIN–MAX) to full range (0–255):

if pixel <= MIN → 0

else if pixel >= MAX → 255

else → ((pixel - MIN) × 255) / (MAX - MIN)

This improves visibility in poor-lighting images.

3.7 Smoothing Filters

Smoothing reduces noise by averaging neighbouring pixels:

Output = (p0 + p1 + p2) / 3

This simple filter eliminates minor fluctuations and enhances image quality.

Feature	Software	Hardware
Speed	Moderate	Very high
Real-time performance	May Vary	Guaranteed
Parallelism	Limited	High
Power Consumption	High	Low
Customizability	High	Medium
Ideal for	Research, Flexibility	Embedded, Deterministic Systems

Table 1.1: Hardware vs. software image processing

This project deals exclusively with hardware-based logic implementation.

4. TOOLS USED

4.1 Verilog HDL

Verilog HDL is the core tool used in this project and forms the foundation for describing all hardware modules involved in pixel manipulation and image processing. Verilog is a Hardware Description Language that enables the modeling, simulation, and synthesis of digital circuits. Unlike traditional programming languages that describe instructions for a CPU to execute, Verilog describes *hardware structures*, such as logic gates, arithmetic units, registers, and sequential circuits. Because of this hardware-centric approach, Verilog is well-suited for implementing image-processing operations that require deterministic execution and parallel data flow.

In this project, Verilog was used to design individual modules for grayscale conversion, brightness control, thresholding, down sampling, contrast stretching, and smoothing. Each module reflects a real hardware block that could be implemented on an FPGA or ASIC. The language allows both **behavioural modelling** (describing what a circuit should do mathematically) and **structural modelling** (connecting multiple modules like components in a hardware schematic).

Verilog is particularly advantageous for image processing because it supports:

- **True parallelism:** Multiple operations can run at the same time, unlike software that executes instructions sequentially.
- **Clock-driven execution:** Ensures predictable timing needed for real-time video and camera pipelines.
- **Bit-level manipulation:** Essential for operations such as thresholding, intensity comparison, and pixel masking.
- **Synthesizability:** The same Verilog code used for simulation can later be mapped onto FPGA hardware without modification.
- **Modularity:** Each image-processing block is independent and reusable, enabling easy system expansion.

Overall, Verilog HDL provides a flexible, efficient, and hardware-accurate environment to design the core processing logic used in this project.

4.2 Simulation Environment

Before any hardware implementation is attempted, it is essential to verify the correctness of digital logic using a simulation environment. Tools such as **Xilinx Vivado**, **ModelSim**, or **Xilinx ISE** were used for simulation in this project. These software tools provide a complete workflow for compiling Verilog code, generating testbenches, monitoring signals, and analysing waveform outputs.

The simulation environment plays a vital role by allowing the designer to inspect the behaviour of each module under different pixel values, threshold levels, brightness offsets, and clock cycles. Using a waveform viewer, every signal transition—including intermediate grayscale values, brightness-adjusted outputs, threshold results, and smoothed pixel values—can be observed in detail. This helps identify logical errors, timing issues, overflow/underflow conditions, and unwanted glitches.

Key features of the simulation environment include:

- **Verilog compiler** to translate HDL into a testable simulation model.
- **Waveform viewer** to visualize pixel transformations over time.
- **Testbench support** to apply custom inputs such as RGB values, threshold settings, offsets, and clock pulses.
- **Debugging utilities** to step through clock cycles, inspect intermediate registers, and isolate errors.
- **Clock and reset simulation** for testing sequential circuits like counters used in down sampling.

Using simulation ensures that the design is functionally correct and stable before being deployed on high-cost hardware platforms. This step significantly reduces debugging time and guarantees correct pixel operations at the hardware level.

5.BLOCK DIAGRAM AND WORKING

5.1 System Architecture:

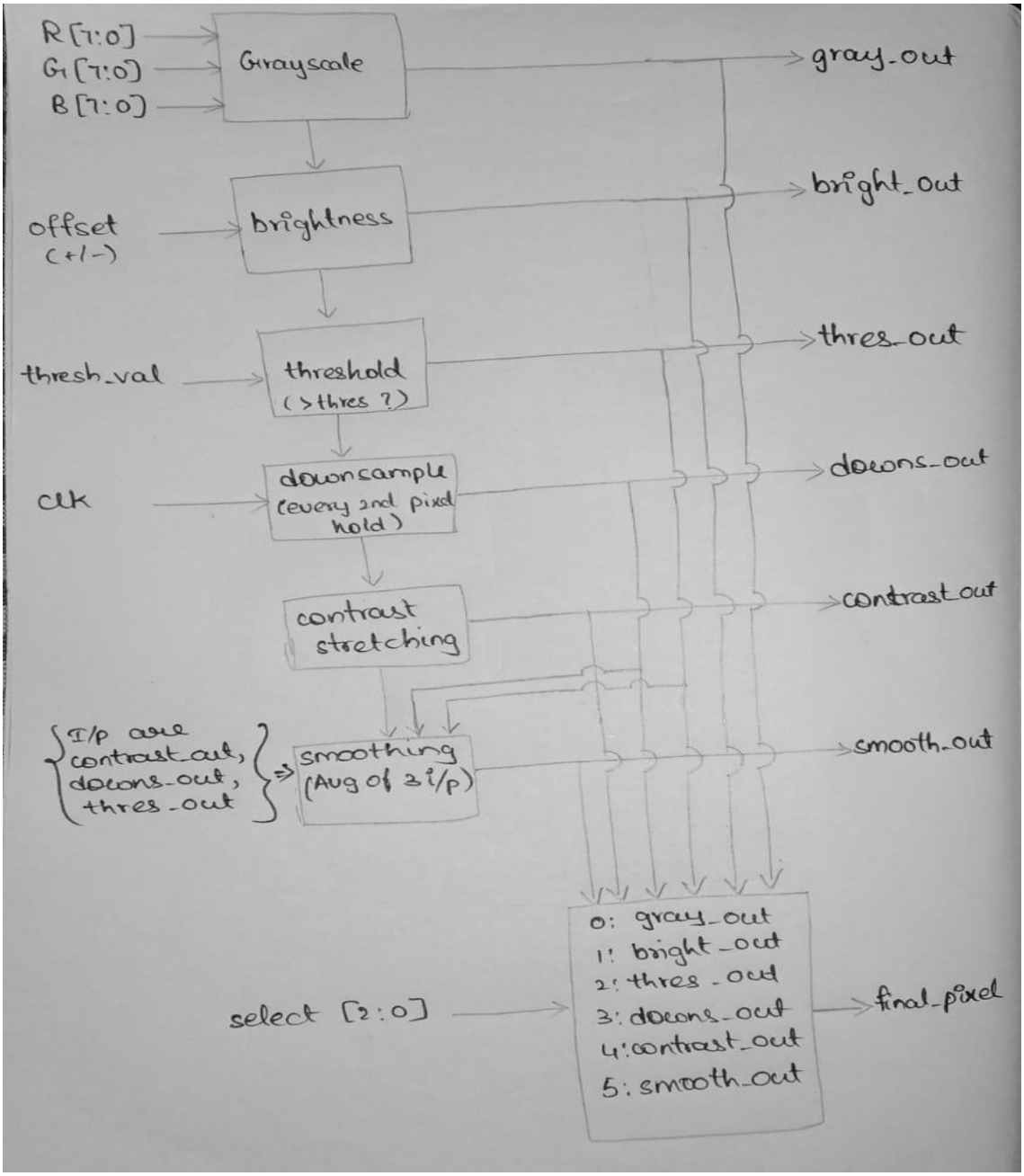

Figure 1.1: Block Diagram of Basic Image Processing

5.2 Module-Level Working:

Operation	Function
Grayscale	Converts RGB (3-channel color) into a single 8-bit intensity value using weighted average.
Brightness Adjustment	Adds a positive/negative offset to each pixel and clamps it between 0–255.
Thresholding	Compares each pixel with a threshold value and outputs 0 or 255 (binary image).
Down-sampling	Reduces the pixel rate by storing one pixel every few cycles using a counter.
Contrast Stretching	Expands pixel values from a limited range (MIN–MAX) to full range 0–255.
Smoothing(3-Tap Filter)	Averages 3 consecutive pixel values to reduce random noise.

Figure 1.2: Module-level Working of Basic Image Processing

We designed the image processor as a sequence of **independent Verilog modules**, each performing a specific enhancement step. Incoming pixel data flows through each module. Each stage produces an intermediate output, which can be selected via a multiplexer. This modular approach allows easy debugging, testing, and extensibility.

6.Verilog Implementation

6.1 Grayscale Module:

```
 module grayscale( gray , R , G , B );
    input  [7:0] R, G, B;
    output [7:0] gray;
    assign gray = ((R*30) + (G*59) + (B*11)) / 100;
endmodule
```

6.2 Brightness Module:

```
module brightness( pixel_out , pixel_in , offset );
    input  [7:0] pixel_in;
    input signed  [7:0] offset;
    output reg [7:0] pixel_out;
    reg signed [9:0] temp;
    always @(*)
    begin
        temp = pixel_in + offset;
        if (temp < 0)
            pixel_out = 0;
       else if (temp > 255)
            pixel_out = 255;
       else
            pixel_out = temp[7:0];
  end
endmodule
```

6.3 Thresholding Module:

```
module thresholding( pixel_out , pixel_in , threshold );
    input [7:0] pixel_in,threshold;
    output [7:0] pixel_out;
    assign pixel_out = (pixel_in > threshold) ? 8'd255 : 8'd0;
endmodule
```

6.4 Downsampling Module:

```
module downsample( pixel_out , pixel_in , clk );
    input  [7:0] pixel_in;
    input        clk;
    output [7:0] pixel_out;
    reg [7:0] counter = 0,last_pixel = 0;
    always @(posedge clk)
    begin
        counter <= counter + 1;
        if (counter == 2)
        begin
            last_pixel <= pixel_in;
            counter <= 0;
        end
    end
    assign pixel_out = last_pixel;
endmodule
```

6.5 Contrast Stretching Module:

```
module contrast_stretch( pixel_out , pixel_in );
    input  [7:0] pixel_in;
    output [7:0] pixel_out;
    parameter MIN = 50;
    parameter MAX = 200;
    reg [15:0] temp_reg;
    always @(*)
    begin
      if (pixel_in <= MIN)
          temp_reg = 0;
      else if (pixel_in >= MAX)
          temp_reg = 255;
      else
          temp_reg = ((pixel_in - MIN) * 255) / (MAX - MIN);
    end
    assign pixel_out = temp_reg[7:0];
endmodule
```

6.6 Smoothing Module:

```
module smoothing( pixel_out , pixel0 , pixel1 , pixel2 );
    input  [7:0] pixel0;
    input  [7:0] pixel1;
    input  [7:0] pixel2;
    output [7:0] pixel_out;
   assign pixel_out = (pixel0 + pixel1 + pixel2) / 3;
endmodule
```

6.7 Top Integration Module:(Design Code)

```
module image_processor_top  ( final_pixel , gray_out , bright_out , thres_out , downs_out , contrast_out , smooth_out , R , G , B , threshold_val , brightness_offset , select , clk );

output [7:0] final_pixel, gray_out,bright_out, thres_out, downs_out,contrast_out,smooth_out;

input  [7:0] R, G, B,threshold_val;

input  signed [7:0] brightness_offset;

input  [2:0] select;

input clk;

wire [7:0] gray, bright, thres, downs, contrast, smooth;

reg [7:0] out_reg;


   // 1. Grayscale

   grayscale u1(gray, R, G, B);

   assign gray_out = gray;


   // 2. Brightness Adjustment

   brightness u2(bright, gray, brightness_offset);

   assign bright_out = bright;


   // 3. Thresholding

   thresholding u3(thres, bright, threshold_val);

   assign thres_out = thres;


   // 4. Downsampling

   downsample u4(downs, thres, clk);

   assign downs_out = downs;
```

```
    // 5. Contrast Stretching
    contrast_stretch u5(contrast, downs);
    assign contrast_out = contrast;

    // 6. Smoothing filter (final stage)
    smoothing u6(smooth, downs, contrast, bright);
    assign smooth_out = smooth;



    always @(*) begin
        case(select)
            3'd0: out_reg = gray;
            3'd1: out_reg = bright;
            3'd2: out_reg = thres;
            3'd3: out_reg = downs;
            3'd4: out_reg = contrast;
            3'd5: out_reg = smooth;
            default: out_reg = 8'd0;
        endcase
    end

    assign final_pixel = out_reg;

endmodule
```

6.8 Test Bench:

```
module tb;

reg [7:0] R, G, B, threshold_val;

reg signed [7:0] brightness_offset;

reg clk;

reg [2:0] select;

wire [7:0] final_pixel,gray_out, bright_out, thres_out, downs_out, contrast_out, smooth_out;

image_processor_top uut  ( final_pixel , gray_out , bright_out , thres_out , downs_out ,
contrast_out , smooth_out , R , G , B , threshold_val , brightness_offset , select , clk );

always #5 clk = ~clk;

initial

   begin

      clk = 0; brightness_offset = 20;threshold_val = 100;

      R=100; G=150; B=200;

      select = 0; #10;  // grayscale

      select = 1; #10;  // brightness

      select = 2; #10;  // threshold

      select = 3; #10;  // downsample

      select = 4; #10;  // contrast stretch

      select = 5; #10;  // smoothing

      R=250; G=120; B=60;

      select = 0; #10;    select = 1; #10;    select = 2; #10;  select = 3; #10;

      select = 4; #10;   select = 5; #10;     $finish;

   end

endmodule
```

7.Results & Discussions:

7.1Waveform:

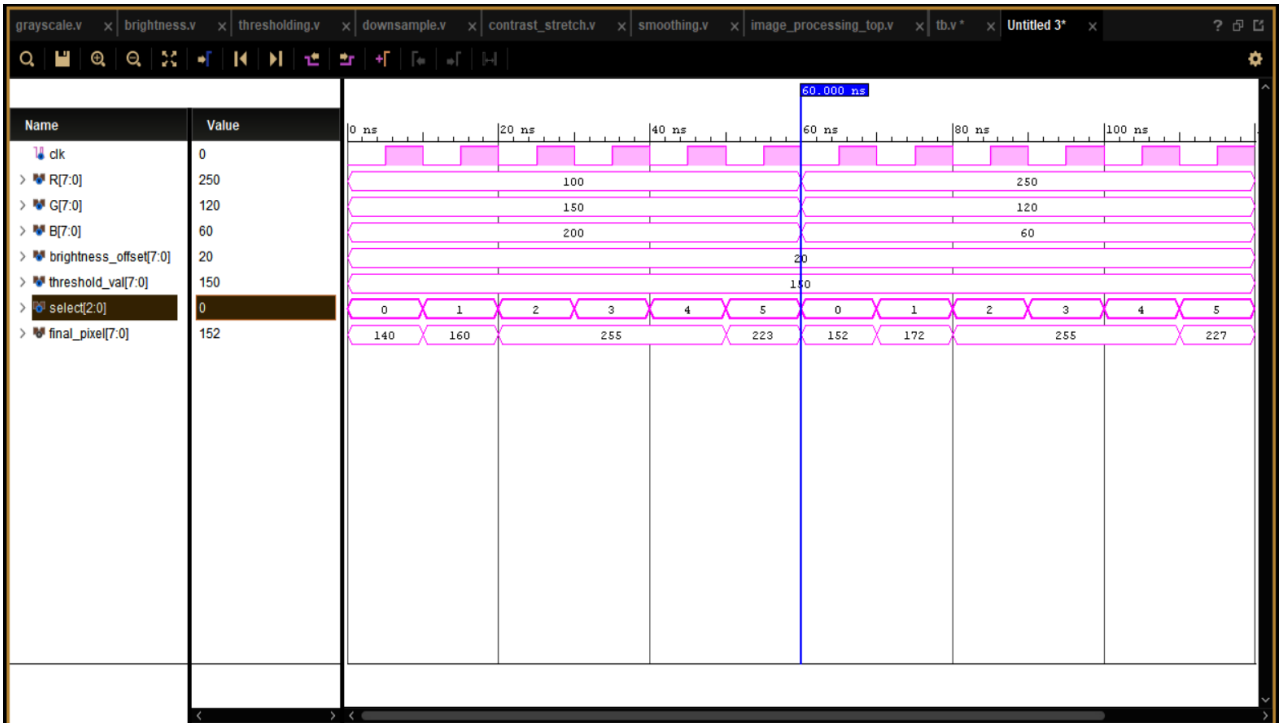

Figure 1.3: Obtained Waveform of Basic Image Processing

7.2 Pixel Transformations:

The pixel values observed during simulation clearly demonstrate that each processing stage behaves as expected. Grayscale conversion consistently reduces RGB inputs to appropriate intensity levels. Brightness adjustment shows predictable upward or downward shifts based on the applied offset. Thresholding produces clean binary outputs where pixels above the threshold are converted to 255 and others to 0. Contrast stretching noticeably expands mid-range values across the full 0–255 range, improving pixel visibility. Smoothing operations reduce sudden changes between adjacent pixels, resulting in visibly lower noise. Overall, before-and-after comparisons confirm that the pipeline alters pixel values exactly according to the intended algorithms.

7.3 Observations:

All image-processing modules operate independently and generate stable outputs during simulation. Switching between module outputs using the select signal functions correctly, with no unexpected behavior. Minor transitions during switching are expected due to testbench timing and do not affect overall system accuracy. Value clamping ensures that all outputs remain within the valid pixel range of 0–255, preventing overflow or underflow. The design remains consistent across multiple test inputs, confirming the robustness of the implementation.

7.4 Limitations:

The current design focuses only on single-pixel processing and does not include frame-level memory or buffering, which are required for full image or video operations. The smoothing filter is limited to a simple 3-pixel average and does not account for larger window filters or spatial kernels. More complex operations such as convolution, edge detection, histogram equalization, or multi-pixel neighborhood processing are not yet implemented. The project demonstrates the core concepts but does not address large-scale image processing workloads within the current scope.

7.5 Conclusions:

This project demonstrates that fundamental image-processing operations can be effectively implemented using Verilog HDL, with each module—grayscale, brightness adjustment, thresholding, downsampling, contrast stretching, and smoothing—showing correct and stable behavior during simulation. The modular design ensured easy testing and reliable pixel transformations within the valid 0–255 range. While limited to single-pixel processing, the implementation provides a clear foundation for scaling toward frame-level and real-time hardware image-processing systems on FPGA platforms.

8.Refernces:

1. R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th Edition, Pearson Education, 2018.
Provides the theoretical background for grayscale conversion, smoothing, thresholding, and other pixel-level transformations used in this project.

2. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, McGraw-Hill, 2014.
Helped in understanding Verilog HDL fundamentals, combinational logic, sequential circuits, and simulation methodologies.

3. Xilinx Inc., *Vivado Design Suite User Guide*, 2020.
Official documentation used for simulation flow, waveform analysis, and tool-based verification of Verilog modules.

4. IEEE Research Papers on Hardware Image Processing, Google Scholar (2015–2024).
Provided insights on FPGA-based image-processing architectures, hardware pipelines, and real-time system design.

5. Online FPGA and Verilog tutorials (fpga4student.com, asic-world.com, Xilinx Community).
Offered practical examples, reference Verilog snippets, and explanations of arithmetic and logical operations used in pixel-processing modules.