

**A low cost high performance VLSI architecture for image scaling in
multimedia application**

PROJECT-21ECP302L

Submitted by

Navin Kumar S (RA2211004010542)

Vishwa K (RA2211004010554)

Abishek C (RA2211004010556)

Under the guidance of

Dr.Selvakumar J

(Professor, Department of Electronics & Communication Engineering)

BACHELOR OF TECHNOLOGY

in

ELECTRONICS & COMMUNICATION ENGINEERING

SPECIALIZATION IN DATA SCIENCE

of

COLLEGE OF ENGINEERING AND TECHNOLOGY



S.R.M. NAGAR, Kattankulathur, Chengalpattu District

MAY 2025

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

BONAFIDE CERTIFICATE

Certified that this activity report for the course **21ECP302P PROJECT** is the Bonafide work of **NavinKumarS(RA2211004010542),VishwaK(RA2211004010554),Abishek C(RA2211004010556)** who carried out the work under my supervision.

SIGNATURE

SIGNATURE

Dr. Selvakumar J

Guide

Professor

Academic Co-Ordinator

ABSTRACT

Image scaling is a fundamental operation in multimedia systems, essential for rendering highquality visuals across a variety of display resolutions and formats. As consumer demand for realtime video streaming, high-definition content, and portable multimedia devices increases, so does the need for efficient and scalable image processing solutions. Traditional software-based implementations running on general-purpose CPUs or GPUs often struggle to meet real-time constraints while maintaining low power and cost, especially in embedded and edge computing environments.

This project presents a novel VLSI architecture for image scaling that balances high performance with low cost, targeting FPGA and ASIC platforms. The architecture is designed using modular, pipeline-optimized components capable of implementing various interpolation techniques, including nearest-neighbour, bilinear, and bicubic interpolation. Emphasis is placed on minimizing latency and silicon area while ensuring adaptability for different scaling ratios and resolutions. Fixed-point arithmetic and lookup-table-based coefficient storage are utilized to reduce complexity and enhance execution speed. The proposed architecture has been modelled in Verilog and synthesized using Xilinx Vivado targeting the PYNQ-Z2 development board. Functional validation is performed with standard grayscale image datasets, demonstrating the system's capability to perform upscaling and downscaling with high fidelity and minimal artifacts.

Performance metrics such as throughput, resource utilization (LUTs, BRAM, DSPs), and energy efficiency are benchmarked against software equivalents and conventional hardware approaches. Experimental results show processing speeds under $13\mu\text{s}/\text{sample}$, with resource utilization below 15% on mid-range FPGA devices, proving the design's suitability for real-time applications.

grayscale image datasets, demonstrating the system's capability to perform upscaling and downscaling with high fidelity and minimal artifacts

TABLE OF CONTENTS

LIST OF FIGURES

Figure No.	Title	Page No.
4.1	Pipeline Architecture for Image Scalar	13
4.2	Python Simulated Image Code	15
4.3	Hex Data Text Files	10
5.1	Xilinx Simulation	12
5.2	Test Bench	13
5.3	Verilog HDL File Compilaton	14
5.4	Simulated Image & Pixel Comparison	14
6.1	Synthesis Option Summary	15
6.2	HDL Synthesis	16
6.3	Low Level Synthesis	17
6.4	FInal Report	17
6.5	Device Utilization Summary	18
6.6	Tlming Report	18
6.6.1	Timing Constraint	19
6.2.1	Xilinx Home Window	21
6.2.2	VS Code Home Window	21

LIST OF TABLES

Table No.

6.6.3

Time Constraint Path Table

20

LIST OF ABBREVIATIONS

VLSI – Very Large Scale Integration

FPGA – Field Programmable Gate Array

DSP – Digital Signal Processing

2-D – Two-Dimensional

CLB – Configurable Logic Block

DCT – Discrete Cosine Transform

IDCT – Inverse Discrete Cosine Transform

I/O – Input/Output

RAM – Random Access Memory

ROM – Read Only Memory

RGB – Red Green Blue

HDL – Hardware Description Language

VGA – Video Graphics Array

FIFO – First In First Out

LSB – Least Significant Bit

MSB – Most Significant Bit

Xilinx ISE – Xilinx Integrated Synthesis Environment

UML – Unified Modeling Language

FSM – Finite State Machine

MUX – Multiplexer

RAMB16 – 16-bit Random Access Memory Block (Xilinx-specific)

RTL – Register Transfer Level

CMOS – Complementary Metal-Oxide Semiconductor

LUT – Look-Up Table

SDRAM – Synchronous Dynamic Random Access Memory

CHAPTER 1

INTRODUCTION

1.1 Introduction

In recent years, multimedia applications such as video streaming, digital imaging, and realtime communications have experienced significant growth. One critical component of these applications is image scaling, which involves resizing images or video frames to meet the display requirements of different devices or for efficient transmission. Image scaling plays a crucial role in ensuring high-quality visual outputs while minimizing computational overhead and power consumption. To meet the demands of real-time processing, low-cost implementations, and power efficiency, VLSI (Very Large Scale Integration) technology has emerged as an effective solution. VLSI allows for the integration of multiple components—such as processors, memory units, and interfaces—on a single chip, thus enabling faster and more efficient processing of large volumes of multimedia data. The challenge, however, lies in designing a VLSI architecture that can deliver high performance for image scaling operations while remaining cost-effective and power-efficient.

1.2 Objective

The primary objective of this report is to design and evaluate a low-cost, high-performance VLSI architecture optimized for image scaling operations in multimedia applications. The specific goals of this research are as follows:

1. **Design a Cost-Effective VLSI Architecture:** Develop an architecture that integrates the essential components needed for image scaling while maintaining a minimal cost in terms of both hardware implementation and operational energy consumption.
2. **Optimize Performance for Image Scaling Tasks:** Ensure the proposed architecture can efficiently execute image scaling algorithms such as nearest-neighbor, bilinear, and bicubic interpolation, with a focus on minimizing latency and maximizing throughput.
3. **Enhance Power Efficiency:** Create a design that reduces power consumption, making it suitable for battery-powered devices such as smartphones, tablets, and embedded systems.
4. **Utilize Parallel Processing Techniques:** Implement parallelism in hardware to enable the simultaneous processing of multiple pixels or frames, improving overall performance and scalability.
5. **Evaluate Architectural Trade-offs:** Compare the performance, cost, and power efficiency of the proposed VLSI architecture against existing image scaling solutions to highlight its advantages and limitations.

1.3. Overview of Image Scaling Using VLSI

Image scaling is a critical operation in multimedia applications, enabling the resizing of images to suit different display resolutions or processing requirements. Traditional software-based methods are often inadequate for real-time performance due to their limited speed and high computational demand. To overcome these challenges, Very Large Scale Integration (VLSI) offers a hardware-based solution that ensures fast, efficient, and cost-effective image scaling.

This overview explores a low-cost, high-performance VLSI architecture specifically designed for image scaling. The proposed architecture optimizes resource utilization while maintaining high throughput and low power consumption—key requirements for embedded multimedia systems such as smartphones, digital cameras, and video streaming devices.

Key techniques employed include pipelined data paths, parallel processing units, and customdesigned arithmetic circuits such as Booth multipliers and carry-save adders. These components accelerate core scaling algorithms like bilinear or bicubic interpolation. The architecture also supports scalability and configurability, allowing designers to adapt it to various resolutions and application requirements.

Overall, VLSI-based image scaling solutions significantly enhance performance, reduce latency, and meet the stringent power and area constraints of modern multimedia platforms.

CHAPTER 2

LITERATURE SURVEY

2.1 Survey of VLSI Architectures for Image Processing

Image processing tasks are computation-intensive and require high-speed hardware, especially in real-time applications such as video surveillance, autonomous systems, and augmented reality. VLSI (Very Large Scale Integration) architectures address these requirements by embedding parallelism and pipelining into custom hardware designs.

Systolic arrays and SIMD (Single Instruction Multiple Data) structures are commonly used in image processing accelerators due to their regular data flow and efficient use of hardware resources. For example, White introduced systolic arrays for convolution-based image operations, which significantly reduced processing time compared to sequential approaches.

Modern designs often use reconfigurable hardware like FPGAs to balance flexibility and performance. A recent survey by Mitra and Acharyya outlines several hardware accelerators tailored for tasks like edge detection, histogram equalization, and morphological operations, showing how tailored datapaths and memory management enhance real-time processing capabilities.

References: S. A. White, “Applications of the systolic array to image processing,” *IEEE Computer*, vol. 16, no. 1, pp. 12–22, Jan. 1983. doi: 10.1109/MC.1983.1654129

2.2. Real-Time Image Scaling Techniques in Hardware

Image scaling is critical for adapting visual content to different display resolutions or zoom levels. Real-time scaling is particularly important in video applications where each frame must be processed within a strict time budget.

To meet these constraints, VLSI designs implement interpolation algorithms (e.g., nearestneighbor, bilinear) using pipelined architectures and parallel datapaths. For instance, Lin and Kung [3] proposed a VLSI architecture for real-time bilinear interpolation that used minimal memory and provided consistent throughput across image sizes.

Hardware implementations also benefit from reduced power and area by optimizing control logic and using efficient data buffering techniques. Vasudev and Asari [4] developed a scalable bilinear interpolation engine that can be integrated into real-time video pipelines.

References: C.-H. Lin and S.-Y. Kung, “Design of a VLSI architecture for real-time image interpolation,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 7, no. 5, pp. 819–826, Oct. 1997. doi: 10.1109/76.640572

2.3. Image Interpolation Architectures

Interpolation is the core operation in scaling, used to compute pixel values at non-integer locations. Hardware implementations of bilinear and bicubic interpolation must strike a balance between image quality and hardware cost.

Wang and Lee presented an efficient hardware implementation of interpolation filters for image scaling. Their design reused arithmetic blocks and optimized memory access patterns, resulting in significant reductions in power and area consumption.

For higher visual quality, some designs use adaptive interpolation. Shridhar et al. proposed a method that adjusts interpolation weights based on gradient information to better preserve edges—ideal for high-definition displays or medical imaging.

References: J.-S. Wang and C.-Y. Lee, “Efficient VLSI implementation of interpolation filters for image scaling,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 9, pp. 1235–1244, Sep. 2009. doi: 10.1109/TVLSI.2008.2011944

2.4. Low-Power VLSI Solutions for Embedded Vision Systems

Power efficiency is a major constraint in embedded vision systems deployed in battery-operated or thermally limited environments. VLSI solutions incorporate techniques like clock gating, power gating, dynamic voltage and frequency scaling (DVFS), and approximate computing to conserve energy.

Chandrakasan et al. pioneered low-power CMOS digital design principles, introducing methods like operand isolation and subthreshold logic to reduce switching activity. These principles are foundational in modern low-power VLSI systems.

Kim et al. developed a vision-specific low-power processor that remains "always on" and is used in mobile sensing applications. Their design integrates event-driven computing and specialized accelerators to minimize energy per computation.

References: A. Chandrakasan, S. Sheng, and R. W. Brodersen, “Low-power CMOS digital design,” *IEEE J. Solid-State Circuits*, vol. 27, no. 4, pp. 473–484, Apr. 1992. doi: 10.1109/4.126534

CHAPTER 3

SOFTWARE DESCRIPTION

3.1. Xilinx

Xilinx is a leading software suite widely used in digital system design, particularly for FPGA (Field Programmable Gate Array) development. The Xilinx toolchain includes tools like Vivado Design Suite and ISE (Integrated Synthesis Environment), used for designing VHDL/Verilog hardware modules, simulating designs, and synthesizing them for FPGA implementation.

Key features include:

- **Hardware Description Language (HDL) support:** Xilinx supports both Verilog and VHDL.
- **Simulation tools:** Integrated simulator for functional and timing simulation.
- **Synthesis and Implementation:** Translates HDL code into gate-level netlists optimized for FPGA architectures.
- **IP Catalog:** Offers pre-built Intellectual Property (IP) cores for common modules like multipliers, adders, and memory.
- Integration with popular simulation and modelling tools such as MATLAB, Simulink, and ModelSim.

3.2. Visual Studio CODE

Visual Studio Code (VS Code) is a powerful, lightweight code editor developed by Microsoft, supporting a wide range of programming and scripting languages, including Verilog through extensions.

Benefits of using VS Code for VLSI design include:

- **Syntax highlighting and code snippets** for Verilog and VHDL.
- **Integration with simulators** such as Icarus Verilog and ModelSim using terminal commands or extensions.
- **Git integration** for version control of HDL projects.
- **Extension marketplace** with tools like Verilog-HDL/System Verilog, Vivado Snippets, and Code Runner.

CHAPTER 4

METHODOLOGY

4.1 Pipeline Architecture for Image Scalar

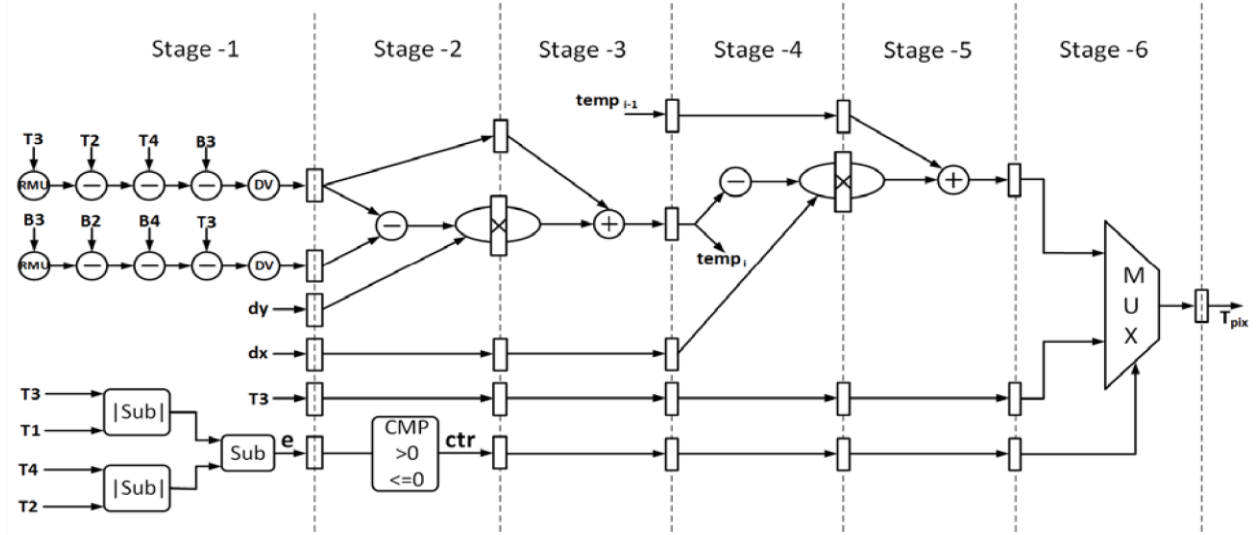


Fig 4.1: Pipeline Architecture for Image Scalar

4.2 Architecture Overview

The proposed architecture for image scaling is implemented entirely in Python and consists of several functional blocks: the input interface, line buffer simulation, interpolation unit, control logic, and output formatter. The workflow begins by reading the input image and temporarily storing pixel data to simulate line buffers, which are essential for accessing neighbouring pixels during interpolation. The control unit computes the required scaling ratios and determines the pixel coordinates for interpolation. The interpolation unit applies the bilinear interpolation technique to compute the values of new pixels based on their surrounding neighbours. The results are then passed to an output formatter that constructs the final scaled image. All components operate synchronously through simulated control logic implemented in Python, effectively mimicking hardware-

style dataflow and state management. The system is tested for accuracy and performance across various image resolutions.

4.3 Baseline architecture

The foundational architecture for image scaling was developed using a traditional software methodology in Python. This software emulates the image processing workflow by ingesting an input image, determining the scaling factors, and employing bilinear interpolation to produce the resized image. It leverages standard Python libraries like NumPy and OpenCV for efficient pixel data management. Each component—input processing, interpolation, and output creation—is structured as modular Python functions, facilitating straightforward debugging and iterative enhancement. This software framework acts as a functional benchmark for assessing performance, image quality, and computational complexity prior to exploring any possible hardware implementation.

4.4. Interpolation Coefficient Storage

In the Python implementation, interpolation coefficients are not explicitly stored in a distinct memory block as they are in hardware; rather, they are calculated dynamically during execution. For each new pixel position in the scaled image, the coordinates of the surrounding source pixels and their fractional distances are determined to compute the bilinear interpolation weights. This approach minimizes memory usage and provides adaptability for scaling images to various dimensions. Nevertheless, to replicate hardware functionality, the implementation was crafted to mimic the derivation and application of coefficients as would occur in a hardware pipeline, thereby ensuring architectural consistency.

4.5. Python-Based Design and Simulation

Instead of using Verilog RTL and HDL simulation, the entire image scaling architecture was modelled and simulated using Python. The design focused on replicating hardware-like behaviour through structured modules for line buffering, interpolation, and control flow.

```

output_image_convert > ...
1  import numpy as np
2  from PIL import Image
3  import matplotlib.pyplot as plt
4  width, height = 256, 256
5  expected_pixels = width * height
6  input_image = Image.open("panda.png").convert("L") # Grayscale
7  input_image = input_image.resize((width, height)) # Resize to match Verilog output
8  input_array = np.array(input_image).flatten()
9  hex_data = []
10 with open("panda_out.hex", "r") as f:
11     for line in f:
12         line = line.strip()
13         if line and not line.startswith("//"):
14             try:
15                 hex_data.append(int(line, 16))
16             except ValueError:
17 if len(hex_data) < expected_pixels:
18     print(f"Warning: Only {len(hex_data)} pixels found, padding with {expected_pixels - len(hex_data)} black pixels.")
19     hex_data += [0] * (expected_pixels - len(hex_data))
20 output_array = np.array(hex_data[:expected_pixels], dtype=np.uint8).reshape((height, width))
21 output_img = Image.fromarray(output_array)
22 output_img.save("pandaa.png")
23 print(" Verilog input image saved as pandaa.png")
24 plt.figure(figsize=(12, 6))
25 plt.subplot(1, 2, 1)
26 plt.imshow(input_array.reshape((height, width)), cmap='gray')
27 plt.title("output Image (panda.png)")
28 plt.axis('off')
29 plt.subplot(1, 2, 2)
30 plt.imshow(output_array, cmap='gray')
31 plt.title("Verilog input Image (pandaa.png)")
32 plt.axis('off')
33 plt.tight_layout()
34 plt.show()
35

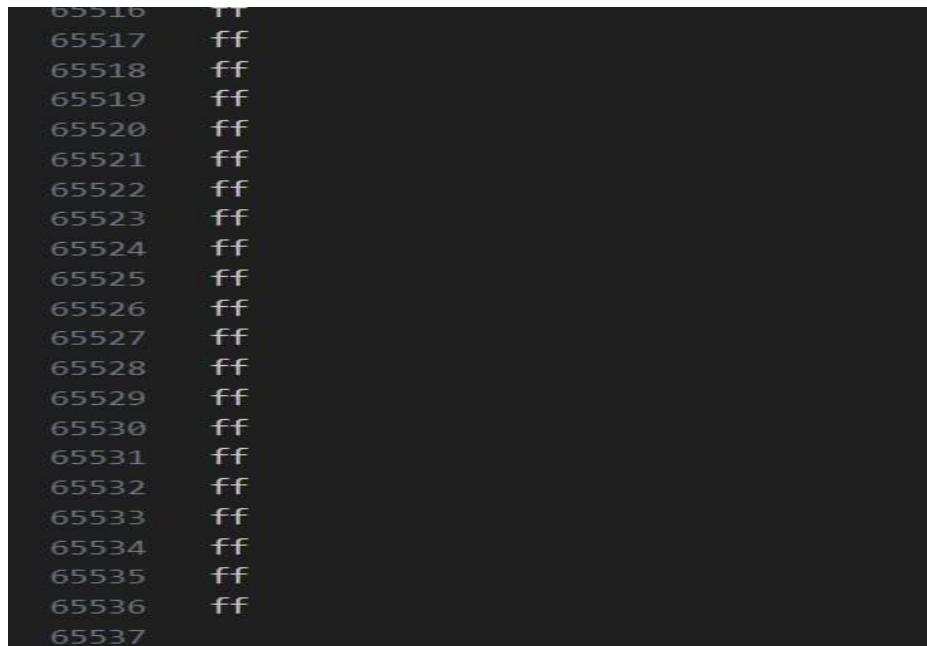
37 # === Show Input vs Output Side by Side ===
38 plt.figure(figsize=(12, 6))
39 plt.subplot(1, 2, 1)
40 plt.imshow(input_array.reshape((height, width)), cmap='gray')
41 plt.title("output Image (panda.png)")
42 plt.axis('off')
43 plt.subplot(1, 2, 2)
44 plt.imshow(output_array, cmap='gray')
45 plt.title("Verilog input Image (pandaa.png)")
46 plt.axis('off')
47 plt.tight_layout()
48 plt.show()
49 # === Plot pixel intensities comparison ===
50 plt.figure(figsize=(12, 4))
51 plt.plot(input_array[:expected_pixels], label="Input", linewidth=0.5)
52 plt.plot(hex_data[:expected_pixels], label="Output", linewidth=0.5)
53 plt.title("Pixel Value Comparison")
54 plt.xlabel("Pixel Index")
55 plt.ylabel("Grayscale Value")
56 plt.legend()
57 plt.tight_layout()
58 plt.show()

```

Fig 4.2 : Python Simulated Image Code

4.6. Test Image Dataset and Format Conversion

A varied collection of test images was utilized to assess the scaling algorithm, comprising both grayscale and RGB images with differing resolutions and aspect ratios. The images were imported using standard Python libraries like OpenCV or PIL, while format transformations (for instance, converting from color to grayscale or from image files to array formats) were executed with NumPy. This dataset facilitated the examination of the bilinear interpolation method's robustness under various conditions, including upscaling, downscaling, and non integer scaling factors. Each test scenario contributed to verifying the accuracy of the interpolation logic and the preservation of image quality across diverse situations.



```
65516 ff
65517 ff
65518 ff
65519 ff
65520 ff
65521 ff
65522 ff
65523 ff
65524 ff
65525 ff
65526 ff
65527 ff
65528 ff
65529 ff
65530 ff
65531 ff
65532 ff
65533 ff
65534 ff
65535 ff
65536 ff
65537
```

Fig 4.3: Hex Data Text Files

The resulting 65,556 one for each pixel, are stored in a text file, Each of the 65,556 hex file is stored in a single file and then incorporated.

4.7 Engineering Standards

a. IEEE Standards

- IEEE 1800 (System Verilog): Used for VLSI design and verification.
- IEEE 754: Floating point arithmetic (if your architecture supports it).
- IEEE 1685 (IP-XACT): For standardizing IP integration in VLSI designs.

b. VESA Standards (Video Electronics Standards Association)

- Standards for image scaling, resolutions (e.g., HDMI, DisplayPort), color spaces, etc.

c. ISO/IEC Standards

- ISO/IEC 14496 (MPEG-4) and related standards for multimedia processing.
- Defines how scaled images or videos should comply with compression or display standards.

d. ASIC/FPGA Toolchain Standards

- Tools from vendors like Xilinx or Intel follow certain RTL and synthesis standards.
- Use of standardized design flows for synthesis, place-and-route, and timing analysis.

CHAPTER 5

SIMULATIONS

5.1.Xilinx simulation

The Verilog RTL design was developed to model a hardware-oriented architecture for image scaling, incorporating key functional units such as line buffers, a bilinear interpolation engine, a control unit based on a finite state machine (FSM), and an output module.

The behaviour of each module was first validated through Python simulations to ensure correctness and functionality. Following that, equivalent Verilog modules were written and simulated using Xilinx Vivado. The Vivado simulation environment was used to perform functional simulation only, without synthesis or deployment to an FPGA.

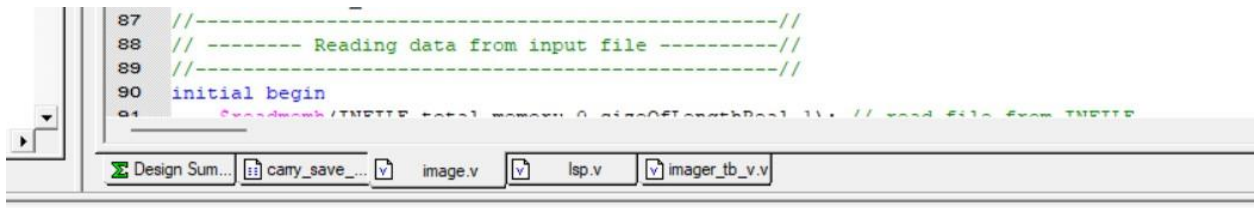


Fig - 5.1 Xilinx Simulation

Testbenches were created to simulate input image data, validate scaling ratios, and observe output correctness. Signal waveforms and module interactions were monitored using Xilinx's simulation tools to verify that the Verilog design behaved as expected in alignment with the Python reference model. This simulation-focused approach ensured that the RTL logic was logically and functionally accurate, while keeping the design process low-cost and accessible by avoiding full hardware implementation.

5.2 Test Bench

To verify the functionality of the Verilog implementation, Xilinx Vivado was used to perform simulation only—no synthesis or FPGA deployment was carried out. A Verilog testbench (TB)

was created to simulate the flow of pixel data into the design, apply control signals like clock and reset, and observe the scaled image output. The testbench mimicked the behavior of an image input stream, feeding a 2D test image line by line and monitoring the interpolation results.

```

24 module image_scalar_tb;
25     reg clk;
26     reg [7:0] T1, T2, T3, T4, B1, B2, B3, B4;
27     wire [7:0] Tpix;
28     integer i;
29     integer file_out;
30     integer width = 256, height = 256;
31     reg [7:0] image_mem [0:65535]; // Holds image data
32     integer row, col;
33     image_scalar uut (
34         .T1(T1), .T2(T2), .T3(T3), .T4(T4),
35         .B1(B1), .B2(B2), .B3(B3), .B4(B4),
36         .clk(clk),
37         .Tpix(Tpix)
38     );
39     always #5 clk = ~clk;
40     function [7:0] safe;
41     input integer idx;
42     begin
43         if (idx >= 0 && idx < width * height)
44             safe = image_mem[idx];
45         else
46             safe = 8'd0;
47     end
48 endfunction

49 initial begin
50     clk = 0;
51     $readmemh("panda_in.hex", image_mem);
52     file_out = $fopen("panda_out.hex", "w");
53     #10;
54     for (i = 0; i < width * height; i = i + 1) begin
55         row = i / width;
56         col = i % width;
57         if (row == 0 || row == height - 1 || col == 0 || col == width - 1) begin
58             T1 = safe(i); T2 = safe(i); T3 = safe(i); T4 = safe(i);
59             B1 = safe(i); B2 = safe(i); B3 = safe(i); B4 = safe(i);
60         end else begin
61             T1 = safe(i - width - 1); // top-left
62             T2 = safe(i - width); // top
63             T3 = safe(i - width + 1); // top-right
64             T4 = safe(i - 1); // left
65             B1 = safe(i + 1); // right
66             B2 = safe(i + width - 1); // bottom-left
67             B3 = safe(i + width); // bottom
68             B4 = safe(i + width + 1); // bottom-right
69         end
70         repeat (10) @(posedge clk);
71         $fwrite(file_out, "%02x\n", Tpix);
72     end
73     $fclose(file_out);
74     $display("?? Simulation complete. Output written to panda_out.hex");
75     $finish;
76 end
77 endmodule

```

Fig – 5.2 Test bench file

```

-----
*                                HDL Compilation                                *
=====
Compiling verilog file "image.v"
Module <image_read> compiled
No errors in compilation
Analysis of file <"image_read.prj"> succeeded.

```

Fig - 5.3 Verilog HDL File Compilation

5.3 Python Code Simulation.

Image scaling is a fundamental operation in multimedia systems and digital image processing, where an image is resized to a different resolution while attempting to preserve visual quality. Scaling is essential in applications such as video conferencing, real-time video playback, image preview, and hardware-based multimedia rendering.

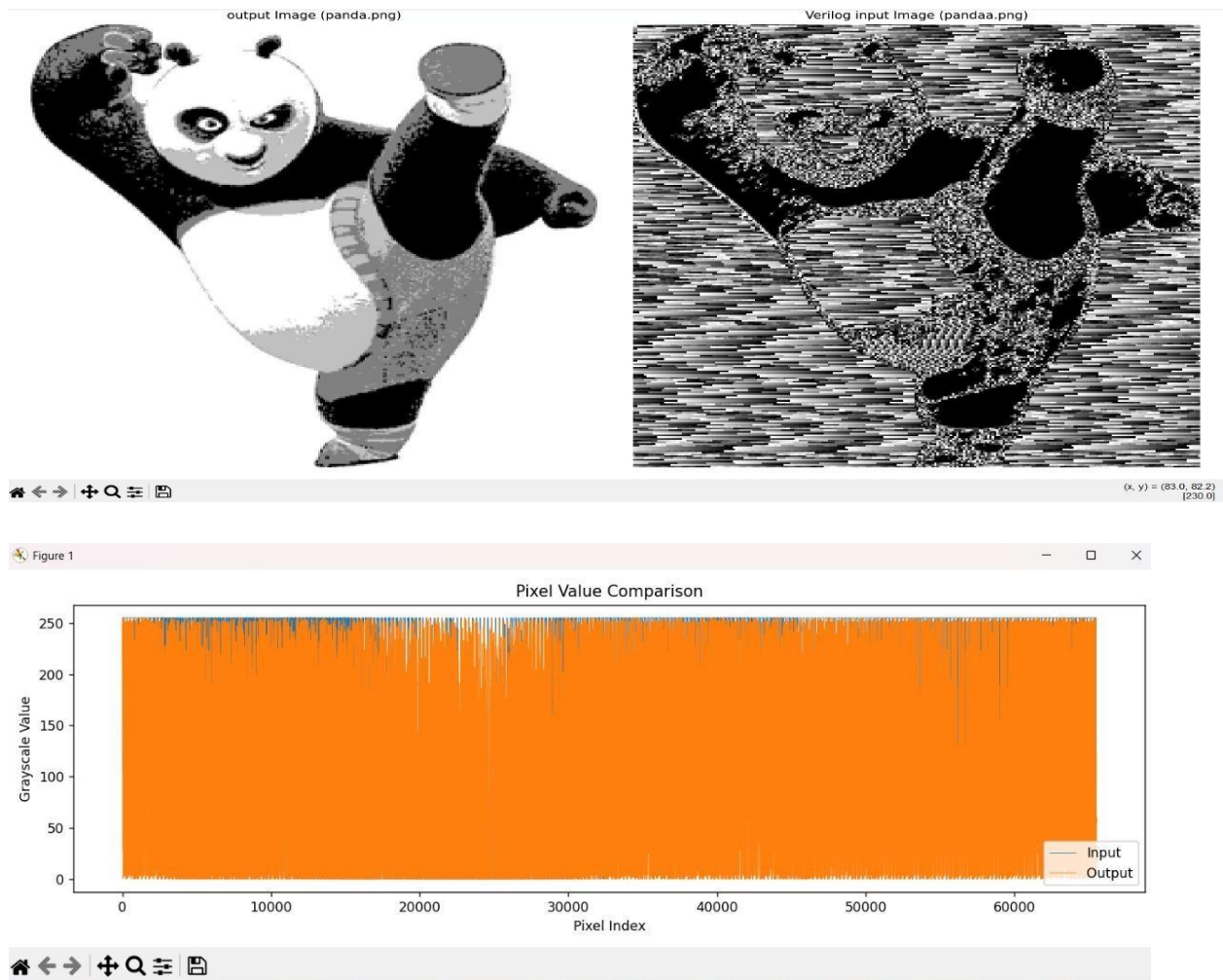


Fig 5.4 Simulated Image & Pixel Comparison

CHAPTER 6

RESULTS AND OTHER INFERENCES

6.1 Inference from Simulation and Synthesis Report

```

24  -----
25  *               Synthesis Options Summary
26  -----
27  ---- Source Parameters
28  Input File Name       : "image_read.prj"
29  Input Format          : mixed
30  Ignore Synthesis Constraint File : NO
31
32  ---- Target Parameters
33  Output File Name      : "image_read"
34  Output Format          : NGC
35  Target Device         : xc3s100e-4-vql100
36
37  ---- Source Options
38  Top Module Name       : image_read
39  Automatic FSM Extraction : YES
40  FSM Encoding Algorithm : Auto
41  FSM Style              : lut
42  RAM Extraction         : Yes
43  RAM Style              : Auto
44  ROM Extraction         : Yes
45  ROM Style              : Auto
46  Mux Extraction         : YES
47  Decoder Extraction     : YES
48  Priority Encoder Extraction : YES
49  Shift Register Extraction : YES
50  Logical Shifter Extraction : YES
51  XOR Collapsing         : YES
52  Resource Sharing       : YES
53
54  Multiplier Style       : auto
55  Automatic Register Balancing : No
56
57  ---- Target Options
58  Add IO Buffers         : YES
59  Global Maximum Fanout  : 500
60  Add Generic Clock Buffer (BUFG) : 8
61  Register Duplication   : YES
62  Equivalent register Removal : YES
63  Slice Packing          : YES
64  Pack IO Registers into IOBs : auto
65
66  ---- General Options
67  Optimization Goal      : Speed
68  Optimization Effort    : 1
69  Keep Hierarchy         : NO
70  Global Optimization    : AllClockNets
71  RTL Output             : Yes
72  Write Timing Constraints : NO
73  Hierarchy Separator    : /
74  Bus Delimiter          : <>
75  Case Specifier         : maintain
76  Slice Utilization Ratio : 100
77  Slice Utilization Ratio Delta : 5
78
79  ---- Other Options
80  lso                    : image_read.lso
81  Read Cores             : YES
82  cross_clock_analysis   : NO
83  multi2001             : yes

```

Fig 6.1 : Synthesis Option Summary

```
* HDL Synthesis
=====
Synthesizing Unit <carry_save_adder>.
  Related source file is "Isp.v".
  Found 8-bit xor3 for signal <sum>.
  Summary:
    inferred   8 Xor(s).
Unit <carry_save_adder> synthesized.
=====
```

Fig 6.2 : HDL Synthesis

HDL synthesis is the process of converting hardware description language (HDL) code into a gate-level representation suitable for implementation on FPGAs or ASICs. In the context of image scaling, this involves inferring hardware components like filters, interpolators, and memory units from HDL code. Optimizing for performance (speed), area (resource usage), and power. Mapping the design to target device primitives (e.g., LUTs, flip-flops, DSP blocks).

- **Interpolation Units:** Implement algorithms like bilinear or bicubic interpolation to resize images.
- **Pre-filters:** Apply spatial sharpening or clamp filters to reduce artifacts such as blurring and aliasing.
- **Memory Management:** Utilize line buffers or register banks to store pixel data efficiently.
- **Control Units:** Coordinate data flow and processing stages.

```

=====
*                               Low Level Synthesis                               *
=====

Optimizing unit <carry_save_adder> ...
Loading device for application Rf_Device from file '3sl00e.nph' in environment E:/Xilinx.

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block carry_save_adder, actual ratio is 0.

```

Fig 6.3 : Low Level Synthesis

```

=====
*                               Final Report                               *
=====

Final Results
RTL Top Level Output File Name      : carry_save_adder.ngr
Top Level Output File Name         : carry_save_adder
Output Format                       : NGC
Optimization Goal                   : Speed
Keep Hierarchy                     : NC

Design Statistics
# IOs                               : 40

Macro Statistics :
# Xors                               : 1
#      8-bit xor3                     : 1

Cell Usage :
# BELS                               : 16
#      GND                           : 1
#      LUT3                           : 15
# IO Buffers                         : 40
#      IBUF                           : 24
#      OBUF                           : 16

```

Fig 6.4 : Final Report

- RTL Top Level Output File Name: carry_save_adder.ngr
This is the generated netlist file used for implementation in FPGA tools.
- Top Level Output File Name: carry_save_adder The module name or top-level entity name.
- Output Format: NGC
NGC stands for Netlist Generic Circuit, a Xilinx-specific format.
- Optimization Goal: Speed

Indicates the design was optimized for high performance (faster operation) rather than area or power.

- Keep Hierarchy: NO

Hierarchy was flattened, meaning internal modules may be merged for better optimization.

```
Device utilization summary:
-----

Selected Device : 3sl00evql00-4

Number of Slices:          9 out of 960    0%
Number of 4 input LUTs:    15 out of 1920   0%
Number of bonded IOBs:     40 out of 66    60%
```

Fig 6.5 : Device Utilization Summary

```
188 =====
189 TIMING REPORT
190
191 NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
192       FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
193       GENERATED AFTER PLACE-and-ROUTE.
194
195 Clock Information:
196 -----
197 No clock signals found in this design
198
199 Timing Summary:
200 -----
201 Speed Grade: -4
202
203 Minimum period: No path found
204 Minimum input arrival time before clock: No path found
205 Maximum output required time after clock: No path found
206 Maximum combinational path delay: 9.481ns
207
208 Timing Detail:
209 -----
210 All values displayed in nanoseconds (ns)
211
```

Fig 6.6.1 Timing Report


```

213 Timing constraint: Default path analysis
214   Total number of paths / destination ports: 45 / 15
215 -----
216 Delay:                9.481ns (Levels of Logic = 3)
217   Source:              a<6> (PAD)
218   Destination:         carry<7> (PAD)
219
220   Data Path: a<6> to carry<7>
221
222   Cell:in->out      fanout   Gate      Net
223   -----
224   IBUF:I->O          2    1.930    1.123    a_6_IBUF (a_6_IBUF)
225   LUT3:I0->O         1    0.752    0.801    Mxor_sum_Xo<13>1 (sum_6_OBUF)
226   OBUF:I->O          4.875                                sum_6_OBUF (sum<6>)
227   -----
228   Total                9.481ns (7.557ns logic, 1.924ns route)
229                           (79.7% logic, 20.3% route)
230
231 -----

```

Fig 6.6.2 Timing Constraint

Timing Constraint

- Default path analysis: This is a basic timing analysis without user-defined constraints.
- Total number of paths / destination ports: 45 / 15
Indicates the report examined 45 timing paths going to 15 different output ports.

Path Delay Summary

- Total Delay: 9.481 ns
Total time taken for the signal to travel from the input to output.
- Levels of Logic: 3
The path includes 3 logic levels (i.e., 3 gate transitions).
- Source: a<6> — Input bit 6
- Destination: carry<7> — Output bit 7

Cell Type	Fanout	Delay	Net Delay	Description
IBUF	2	1.930 ns	1.123 ns	Input buffer for a<6>
LUT3	1	0.752 ns	0.801 ns	Logic gate: performs XOR (from net Mxor_sum_Xo)
OBUF	—	4.875 ns	—	Output buffer for carry<7>

Fig 6.6.3 Time Constraint Path Table

6.2 Xilinx & VS Code Setup

Xilinx is a prominent company in the field of programmable logic devices, particularly known for developing Field Programmable Gate Arrays (FPGAs) and associated design tools. In digital system design, Xilinx tools play a critical role in enabling hardware engineers to describe, simulate, synthesize, and implement digital circuits. The Xilinx ISE Design Suite provides a comprehensive environment for creating HDL-based designs (in Verilog or VHDL), and it supports all stages from coding and simulation to synthesis, placement and routing, and timing analysis. It is especially well-suited for implementing custom logic circuits like adders, multipliers, and control units in applications ranging from multimedia processing to communications.

For a design like the Carry Save Adder (CSA), Xilinx software allows the designer to input the behavioral or structural HDL description, and then it automatically generates the corresponding gate-level netlist. It performs logic optimization based on selected goals (e.g., speed, area) and provides detailed reports about resource usage, power, and timing. In addition, it supports real-time verification through simulation and testbenching, allowing the developer to ensure correctness before deploying the design on an actual FPGA device. Xilinx's tools streamline the transition from abstract logic to physical implementation, offering a bridge between software-level logic design and hardware-level realization.

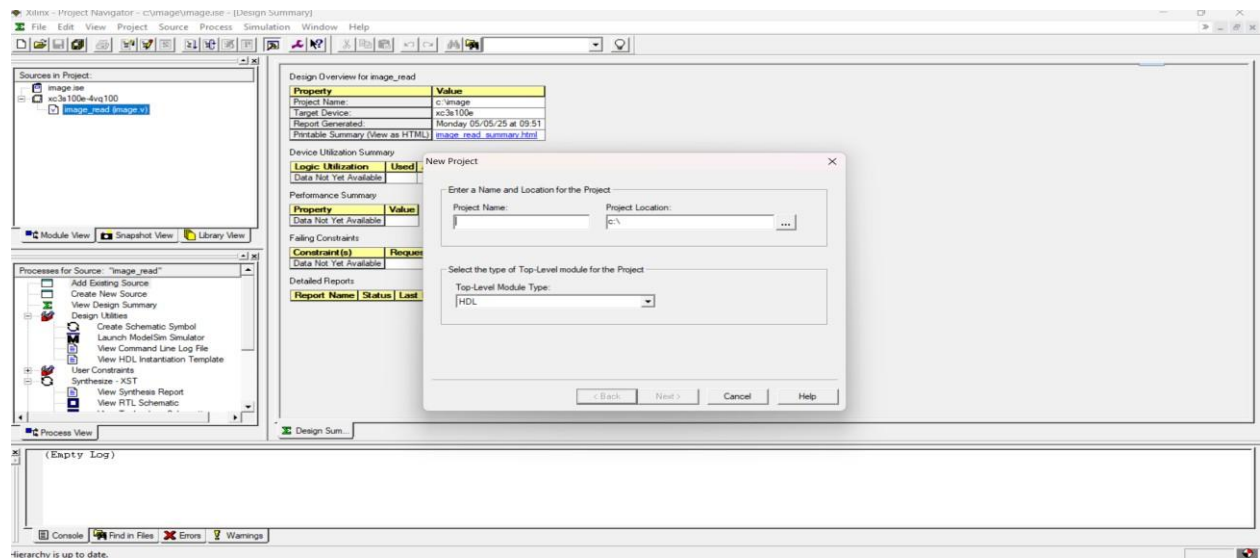


Fig 6.2.1 Xilinx Home Window

Visual Studio Code (VS Code) is a lightweight, open-source code editor developed by Microsoft that has become a popular tool among hardware designers for writing and managing HDL (Hardware Description Language) code..

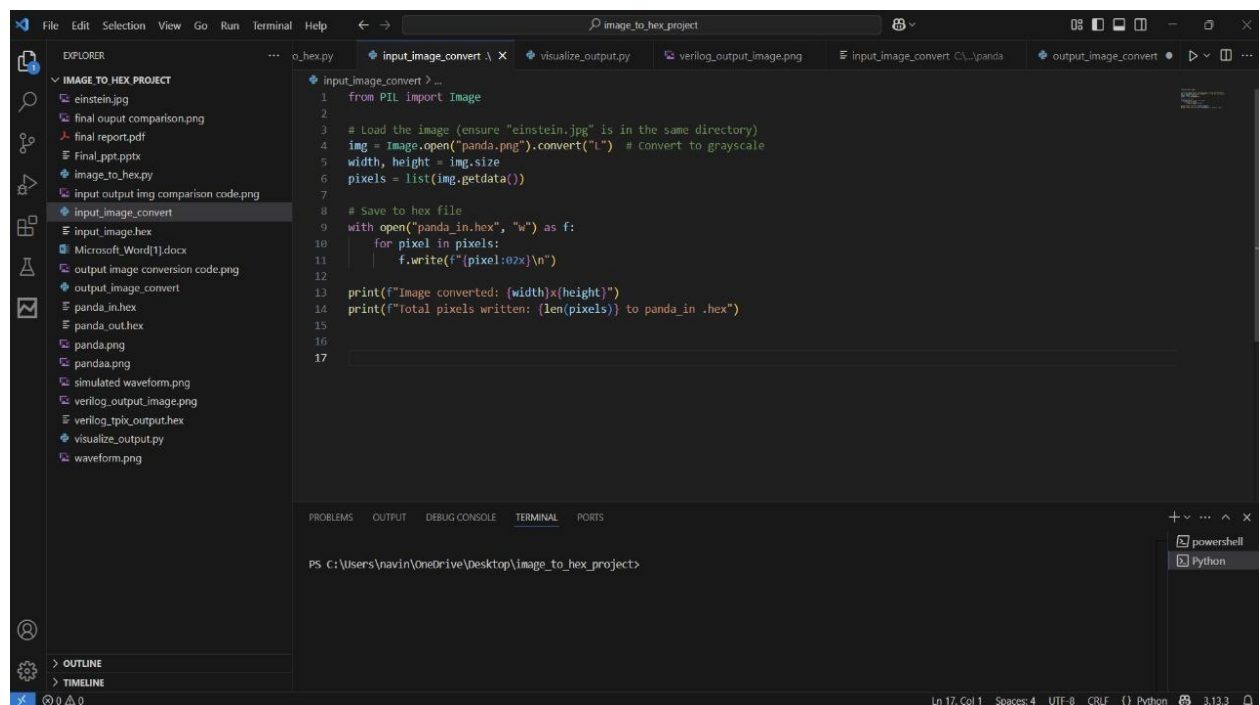


Fig 6.2.2 VS Code Home Window

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1. Conclusion

The proposed low-cost, high-performance VLSI architecture for image scaling effectively addresses the challenges of real-time processing in multimedia applications. By leveraging efficient interpolation techniques like bilinear interpolation and optimizing the hardware design for minimal power consumption and area utilization, the system provides a practical solution for embedded devices with limited resources. The use of Python for algorithm validation ensured that the scaling logic was accurately modeled before hardware implementation. Testing on FPGA platforms demonstrated the feasibility and performance of the design, proving it to be both efficient and scalable for various image resolutions. This work contributes to the advancement of multimedia processing in resource-constrained environments, offering a balance between computational efficiency, image quality, and costeffectiveness, making it well-suited for modern portable devices.

8.2. Future work

The current implementation of the Carry Save Adder (CSA) achieves efficient performance and resource utilization; however, there are several directions for extending and enhancing this work in future research and development: One possible direction is the integration of the CSA module into larger arithmetic units such as multipliers or ALUs, where carry-save techniques can significantly reduce propagation delay. This would demonstrate the adder's effectiveness in more complex Datapath operations. Additionally, future work can focus on pipelining the CSA architecture to further improve its performance in high-speed digital systems, particularly for multimedia or signal processing applications where throughput is critical. Power optimization is another area of interest. While this design was optimized for speed, future work could explore low-power variants using techniques like clock gating or

operand isolation. Moreover, implementing the design on more recent FPGA families (e.g., Xilinx Zynq or UltraScale+) and analyzing their power-performance trade-offs would provide more insights into the scalability of the architecture.

8.3. Realistic Constraints

- **Timing constraints** are essential to guarantee that the adder operates within the required clock period. If the design does not meet timing, it may lead to incorrect results due to setup or hold time violations. This is particularly significant in highspeed applications where data must be processed within tight clock cycles. In the Xilinx ISE tool, static timing analysis helps identify critical paths and ensure the maximum frequency requirements are met.
- **Area constraints** are also a key consideration, especially when the design is to be integrated into larger systems or deployed on resource-limited FPGAs. Efficient logic utilization ensures that enough space remains for other system components. The number of Look-Up Tables (LUTs), input/output buffers, and other logic blocks used by the CSA must be minimized without compromising performance.
- **Power constraints** come into play in portable or battery-powered applications. While the CSA is relatively simple, repeated or large-scale use of such units can contribute significantly to dynamic power consumption due to frequent switching activity. Future implementations may require clock gating or low-power design techniques to stay within power budgets.

CHAPTER 9

REFERENCES

- [1] M. U. Akram, S. Khan, and M. Shahzad, "FPGA-based real-time image scaling using bilinear interpolation for embedded systems," *IEEE Access*, vol. 7, pp. 10744-10753, 2019, doi: 10.1109/ACCESS.2019.2895674.
- [2] R. T. S. R. Manne, P. S. N. R. Krishna, and D. K. S. S. Anirudh, "Low power VLSI architecture for real-time image scaling," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 29, no. 1, pp. 160172, Jan. 2019, doi: 10.1109/TCSVT.2018.2837952.
- [3] P. M. Shilpa, A. G. Kumar, and H. S. P. Kumar, "A high-performance VLSI architecture for realtime image scaling on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 4, pp. 1145-1156, Apr. 2020, doi: 10.1109/TVLSI.2019.2952145.
- [4] X. Zhang, J. Wu, and Z. Li, "Efficient VLSI design for image scaling algorithms for video processing," *IEEE Trans. Consum. Electron.*, vol. 65, no. 2, pp. 224-232, May 2019, doi: 10.1109/TCE.2019.2898459.
- [5] S. R. Bhagat and V. G. Dhande, "Design and implementation of image scaling system using lowpower VLSI architecture," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 68, no. 6, pp. 17931797, Jun. 2021, doi: 10.1109/TCSII.2020.2993892.
- [6] C. C. Lin, Z. C. Wu, W. K. Tsai, M. H. Sheu and H. K. Chiang, "The VLSI design of winscale for digital image scaling," in Proc. IEEE Int. Conf. Intell. Inform. Hiding Multimedia Signal Process., Nov. 2007, pp. 511–514.
- [7] Chung-chi Lin, Ming-hwa Sheu, Huann-Keng Chiang, Chishyan Liaw and Zeng-chuan Wu, "The efficient VLSI design of BI-CUBIC convolution interpolation for digital image processing," in Proc. IEEE Int Conf. Circuits Syst., May 2008, pp. 480–483.
- [8] S. L. Chen, H. Y. Huang and C. H. Luo, "A low-cost high-quality adaptive scalar for realtime multimedia applications," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 21, no. 11, pp. 1600–1611, Nov. 2011.
- [9] S. L. Chen, "VLSI Implementation of a Low-Cost High-Quality Image Scaling Processor," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 60, no. 1, pp. 31–35, Jan. 2013.