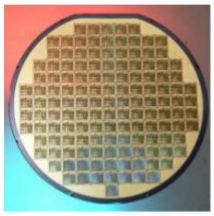
THE ANDREW & ERNA VITERBI
FACULTY OF
ELECTRICAL
ENGINEERING

TECHNION – ISRAEL INSTITUTE OF TECHNOLOGY



VLSI SYSTEMS RESEARCH CENTER

VLSI LABORATORY

DEPARTMENT OF ELECTRICAL ENGINEERING | DEPARTMENT OF COMPUTER SCIENCE

Hardware accelerator for adaptive edge enhancement of video stream Project

Final Report

Shiraz Hill

Supervisor: Pavel Gushpan

Technion- Israel Institute of Technology
Project B (044169) | Special project (044170)
Spring 2023/2024

Contents

Figures

Abstract

This project presents the design, synthesis, and layout of an efficient hardware implementation for adaptive Sobel edge detection in video processing, leveraging Cadence tools for development. The core objective is to create a VLSI solution that effectively balances algorithmic accuracy and computational efficiency, meeting stringent requirements for latency, frame rate, and power consumption. By employing SystemVerilog (SV) blocks for the edge detection pipeline, this work addresses the challenges of real-time image processing, ensuring both high throughput and minimal delay. The project explores the integration of adaptive scaling for edge magnitude control, as well as the use of AXI Stream interfaces for seamless data transfer between processing units. Extensive testing and analysis validate the performance of the design, highlighting its ability to deliver accurate edge detection for serval resolution video streams (e.g. HD) while maintaining high frame rates. The results demonstrate the hardware's suitability for applications in real-time video enhancement, with potential for integration into various fields requiring high-quality image processing, such as computer vision, autonomous systems, and multimedia applications.

Introduction

Chapter 1: Edge Detection Algorithms

Edge detection, especially step edge detectors, is a fundamental component of many computer vision systems. This process simplifies image analysis by significantly reducing the amount of data that needs to be processed, while still preserving critical structural information about object boundaries. By identifying points in an image where the intensity changes sharply, edge detection enhances the recognition and interpretation of shapes and objects within the image, making it a crucial step in various image processing applications. [3]

This explanation of an image is easy to incorporate into a large amount of object recognition algorithms used in computer vision along with other image processing applications. [1]

Edge detection techniques excel in accurately identifying edge lines with proper orientation, but there is no universal standard for evaluating their performance, which is usually judged subjectively based on specific applications.
Edge detection is a fundamental tool for image segmentation. Edge detection methods transform original images into edge images benefits from the changes of grey tones in

the image. In image processing especially in computer vision, the edge detection treats the localization of important variations of a gray level image and the detection of the physical and geometrical properties of objects of the scene. It is a fundamental process detects and outlines of an object and boundaries among objects and the background in the image. Edge detection is the most familiar approach for detecting significant discontinuities in intensity values. [1]

Edges are local changes in the image intensity. Edges typically occur on the boundary between two regions. The main features can be extracted from the edges of an image. Edge detection has major feature for image analysis. These features are used by advanced computer vision algorithms. Edge detection is used for object detection which serves various applications like medical image processing, biometrics etc. Edge detection is an active area of research as it facilitates higher level image analysis. There are three different types of discontinuities in the grey level like point, line and edges. Spatial masks can be used to detect all the three types of discontinuities in an image.

Recently, adaptive edge detection methods have gained attention due to their ability to dynamically adjust parameters based on image content, thereby significantly enhancing detection accuracy across varying image conditions. Adaptive approaches employ mechanisms to automatically scale edge magnitude thresholds according to local image features or global image statistics. This adaptive scaling allows the algorithm to effectively handle images with diverse intensity ranges and varying levels of noise, improving edge clarity and consistency. By continuously optimizing threshold parameters, adaptive edge detection techniques deliver superior performance and robustness compared to traditional static-threshold methods, making them particularly suited for real-time and dynamic image processing applications.

This chapter reviews the most used discontinuity-based edge detection methods, including Roberts, Sobel, Prewitt and Canny edge detection techniques.

1.1 Roberts

Roberts' algorithm, developed by Lawrence Roberts in 1965, is a fundamental technique for edge detection in digital image processing. It operates by computing the gradient of the image intensity at each pixel, which effectively measures the rate of change of the image's brightness. This is achieved through the application of two 2x2 convolution kernels that approximate the first derivative of the image in both the horizontal and vertical directions. Specifically, these kernels highlight regions where there is a significant change in intensity, corresponding to edges in the image. The resulting gradient magnitude at each pixel is then used to determine the presence and strength of edges.

-1	0
0	+1

G_x

0	-1
+1	0

G_y

figure 1 masks in both x and y axis

1.1.1 MATLAB Code- Roberts

Figure 2 present the MATLAB code for Roberts algorithm.

```
% MATLAB Code for Roberts

% Read Input Image
input_image = imread('super-car-outline-black-and-white-design-free-vector.jpeg');

% Convert the truecolor RGB image to the grayscale image
input_image = rgb2gray(input_image);

% Convert the image to double
input_image = double(input_image);

% Pre-allocate the filtered_image matrix with zeros
filtered_image = zeros(size(input_image));

% Robert Operator Mask
Mx = [1 0; 0 -1];
My = [0 1; -1 0];

% Edge Detection Process
% When i = 1 and j = 1, then filtered_image pixel
% position will be filtered_image(1, 1)
% The mask is of 2x2, so we need to traverse
% to filtered_image(size(input_image, 1) - 1
% size(input_image, 2) - 1)
for i = 1:size(input_image, 1) - 1
    for j = 1:size(input_image, 2) - 1

        % Gradient approximations
        Gx = sum(sum(Mx.*input_image(i:i+1, j:j+1)));
        Gy = sum(sum(My.*input_image(i:i+1, j:j+1)));

        % Calculate magnitude of vector
        filtered_image(i, j) = sqrt(Gx.^2 + Gy.^2);

    end
end

% Define a threshold value
thresholdValue = 100; % varies between [0 255]
output_image = max(filtered_image, thresholdValue);
output_image(output_image == round(thresholdValue)) = 0;

% Displaying Output Image
output_image = im2bw(output_image);
figure, imshow(output_image); title('Edge Detected Image');
```

figure 2 MATLAB code for Roberts edge detection

The running output is presented in figure 3.

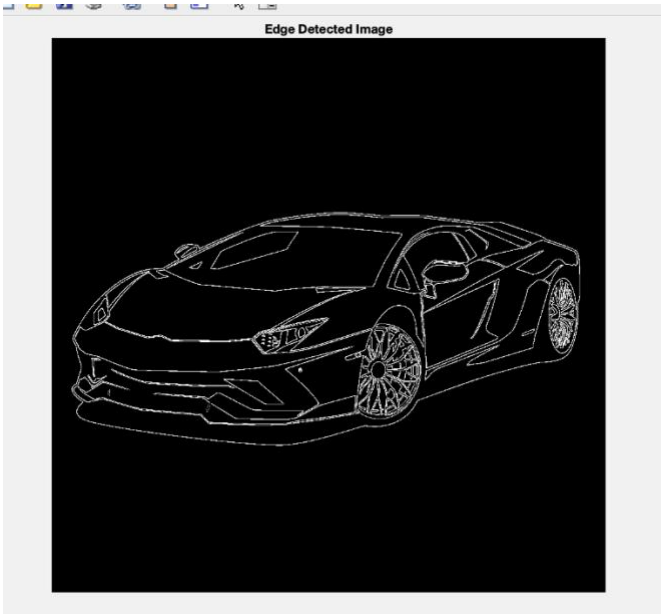


figure 3 Roberts-the full edge detection

1.2 Prewitt

The Prewitt edge detection method, introduced by Judith M. S. Prewitt in 1970, is an effective technique for estimating the magnitude and orientation of edges in an image. The Prewitt operator is particularly useful for detecting horizontal and vertical edges, corresponding to changes in pixel intensities along the x-axis and y-axis, respectively.

This operator uses a first-order derivative mask to identify edges, which appear as local maxima or minima in the resulting graph representation.

Despite the computational complexity associated with different gradient edge detection methods, which require time-consuming calculations to estimate edge directions from magnitudes in the x and y directions, the compass edge detection approach offers a more efficient alternative. This method directly derives edge directions from the kernel with the highest response. Although it is limited to eight possible directions, empirical evidence suggests that this constraint does not significantly compromise accuracy.

The Prewitt edge detector evaluates gradients within a 3x3 neighborhood for these eight directions. It calculates all eight convolution masks, ultimately selecting the one with the largest module to determine the edge direction.

Prewitt operator provides us two masks one for detecting edges in the horizontal direction and another for detecting edges in a vertical direction.

1.2.1 MATLAB Code- Prewitt

Figure 4 present the MATLAB code for Prewitt algorithm. The code based on the following steps of the algorithm [2]:

1. Read the image.
2. Convert into grayscale if it is colored.
3. Convert into the double format.
4. Define the mask or filter.
5. Detect the edges along X-axis.
6. Detect the edges along Y-axis.
7. Combine the edges detected along the X and Y axes.
8. Display all the images.

```
% MATLAB code for prewitt
k=imread('super-car-outline-black-and-white-design-free-vector.jpeg');
k=rgb2gray(k);
k1=double(k);
p_msk=[-1 0 1; -1 0 1; -1 0 1];
kx=conv2(k1, p_msk, 'same');
ky=conv2(k1, p_msk', 'same');
ked=sqrt(kx.^2 + ky.^2);

% display the images
imageViewer(k)

% display the edge detection along x-axis
imageViewer(abs(kx))

% display the edge detection along y-axis
imageViewer(abs(ky))

% display the full edge detection
imageViewer(abs(ked))
```

figure 4 MATLAB code for Prewitt edge detection

The running output is presented in figures 5-8.

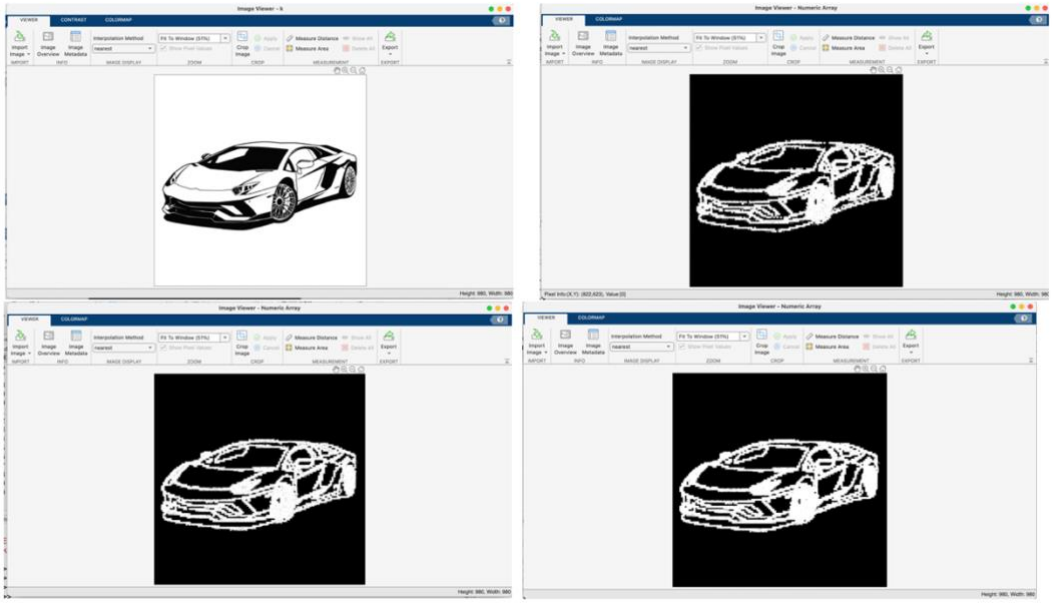


figure 5 The Top left image is the orginal image, the top right is the output of edge detection along X-axis, the bottom left is the output along y-axis ad the right bottom is the full edge detection

1.3 Sobel

The Sobel algorithm, introduced by Irwin Sobel and Gary Feldman in the 1970s, is a widely used method for edge detection in digital image processing. It enhances the traditional gradient method by combining smoothing and differentiation to reduce noise sensitivity while highlighting edges. The algorithm typically uses two 3x3 convolution kernels, one for detecting changes in the horizontal direction (x-axis) and the other for the vertical direction (y-axis). These kernels approximate the first derivative of the image intensity, producing gradient components that indicate the direction and magnitude of the greatest rate of change.

While the 3x3 kernel is the most used configuration, the Sobel algorithm can also be implemented with larger kernels, such as 5x5. The use of a 5x5 kernel provides a more robust smoothing effect, which can further reduce noise and improve the detection of edges in more complex images. This larger kernel considers a wider neighborhood of pixels, resulting in a more refined gradient estimation and potentially more accurate edge detection, especially in images with fine details or significant noise. However, the increased kernel size affects the localization of edges. One kernel is simply the other rotated by 90°. (This is very alike to the Roberts Cross operator)

-1	-2	-1
0	0	0
+1	+2	+1

G_x

-1	0	-1
-2	0	+2
-1	0	+1

G_y

figure 6 2D- spatial gradient, in both x and y axis [1]

1.3.1 MATLAB Code- Prewitt

In figure 10, presented the MATLAB code for Sobel algorithm with 3x3 kernel.

```
% Sobel Edge Detection
sobel_img=rgb2gray(imread('super-car-outline-black-and-white-design-free-vector.jpeg'));
% MATLAB code for Sobel operator
% edge detection
k1=double(sobel_img);
s_msk_x=[-1 0 1; -2 0 2; -1 0 1];
s_msk_y=[-1 -2 -1; 0 0 0; 1 2 1];
kx=conv2(k1, s_msk_x, 'same');
ky=conv2(k1, s_msk_y', 'same');
ked=sqrt(kx.^2 + ky.^2);


% display the edge detection along x-axis
imageViewer(abs(kx))

% display the edge detection along y-axis
imageViewer(abs(ky))

% display the full edge detection
 imageViewer(abs(ked))
```

figure 7 MATLAB code for Sobel

The running output is presented in figures 10-12.

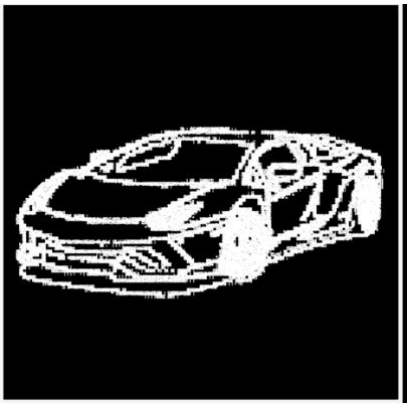


figure 8 edge detection along x-axix

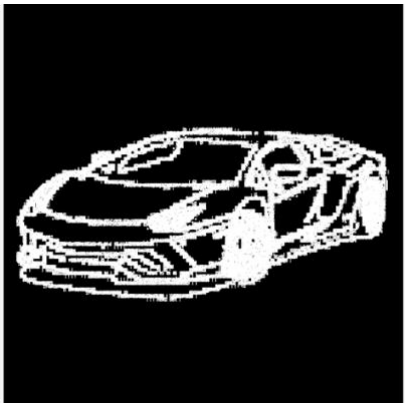


figure 9 edge detection along y-axix

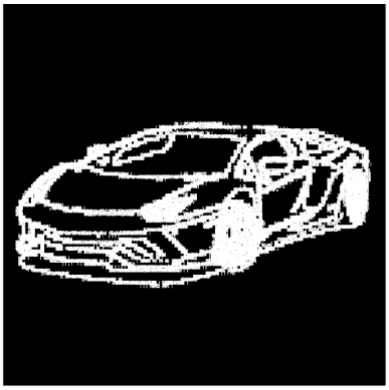


figure 10 Total edge output- Sobel

1.4 Canny

In industry, the Canny edge detection technique is one of the standard edge detection techniques. It was first created by John Canny for his master's thesis at MIT in 1983, and still outperforms many of the newer algorithms that have been developed. [1]

Canny method is a better method without disturbing the features of the edges in the image afterwards it is applying the tendency to find the edges and the serious value for threshold.

The algorithmic steps are as follows, as shown more mathematically in Pseudocode in figure 13:

1. Step one, **Gaussian Smoothing**- meaning, convolve the input with a Gaussian filter to produce a smoothed image.
2. Step two, **Gradient Calculation**- Apply a first difference gradient operator to the smoothed image to compute the gradient vector at each pixel. Calculate the edge strength (gradient magnitude) and direction.
3. Step three, **Non-Maxima Suppression**-Thin the edges by applying non-maxima suppression to the gradient magnitude image. This step retains only the local maxima in the direction of the gradient, ensuring edges are one pixel thick.
4. Step four, **Double Thresholding**- Apply a high and low threshold to the non-maxima suppressed image to detect strong and weak edges. Pixels with gradient magnitude greater than the high threshold are marked as strong edges. Pixels with gradient magnitude between the low and high thresholds are marked as weak edges.
5. Step five, **Edge Tracking by Hysteresis**- Start with the strong edges and recursively include the weak edges that are connected to strong edges. This ensures that weak edges connected to strong edges are also considered as edges.

Algorithm 1 Standalone Canny Edge Detection

Input: Image I, thresholds τ_l, τ_h
Output: Canny edge image E, gradient information M, ϕ

procedure CANNY-STANDALONE
Convolve image with Gaussian filter G
  $I \leftarrow I \circledast G$
Compute gradients
  for each pixel (x, y) **do**
    $g_x(x,y) = \frac{1}{2}I(x-1,y) - I(x,y) + \frac{1}{2}I(x+1,y)$
    $g_y(x,y) = \frac{1}{2}I(x,y-1) - I(x,y) + \frac{1}{2}I(x,y+1)$
    $M(x,y) = \sqrt{g_x(x,y)^2 + g_y(x,y)^2}$ ▷ magnitude
    $\phi(x,y) = \tan^{-1}(g_y(x,y)/g_x(x,y))$ ▷ orientation
  end for
Non-maxima suppression
  for each pixel (x, y) **do**
    Find 2 neighbours in direction of gradient $\phi(x, y)$
    if $M(x, y) \leq M$(any neighbour) **then**
      $M(x, y) \leftarrow 0$
    end if
  end for
Hysteresis thresholding
  for each pixel (x, y) **do**
    if $M(x, y) \geq \tau_h$ **then**
      $E(x, y) \leftarrow 1$ ▷ edge pixel
    else
      $E(x, y) \leftarrow 0$ ▷ non-edge pixel
    end if
  end for
  Recursively find non-edge pixels with $M \geq \tau_l$ and with an edge pixel as a neighbour. Mark them as edge pixels as well.
end procedure

figure 11 Canny Algorithm Pseudocode [4]

1.4.1 MATLAB Code- Canny

Matlab offers the edge function. This function looks for places in the image where the intensity changes rapidly, using one of these two criteria:

- Places where the first derivative of the intensity is larger in magnitude than some threshold
- Places where the second derivative of the intensity has a zero crossing edge provides several derivative estimators, each of which implements one of these definitions. For some of these estimators, you can specify whether the operation should be sensitive to horizontal edges, vertical edges, or both. edge returns a binary image containing 1's where edges are found and 0's elsewhere.

The most powerful edge-detection method that edge provides is the Canny method. The Canny method differs from the other edge-detection methods in that it uses two different thresholds (to detect strong and weak edges), and includes the weak edges in the output only if they are connected to strong edges.

The running output is presented in figure 14.

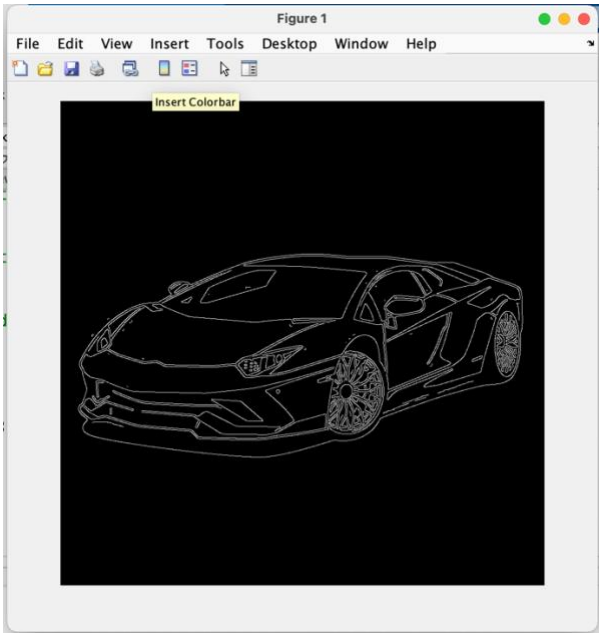


figure 12 Total edge output- Sobel

1.5 Comparison table

In Table 1, we compare the different algorithms based on the parameters presented in the table for convenience and future selection for further work with the desired algorithm.

Algorithm/Parameter	Simplicity	Noise	In use today	Year created	localization	accuracy
Roberts	Quite simple to implement computationally	Noise-sensitive	Commonly used in certain niches and for educational purposes.	1965	good	Not accurate
Sobel	Quite simple to implement computationally Easy to implement in hardware	Noise-sensitive	Commonly used in It's used in basic image processing pipelines, such as: -Preprocessing for OCR (optical character recognition) -Simple object boundary detection -Motion and contour analysis - It's also used as a **baseline algorithm** to compare more	1970	Depends on the kernel size (quite good for 3x3)	Accurate with High-contrast, clean images

			advanced methods.			
Prewitt	Quite simple to implement computationally	Nosier than the other ones	Commonly used in certain niches and for educational purposes.	1970	Quite good	Not accurate
Canny	Very complicated	Not very susceptible to noise	Commonly used in industry	1983	Quite good	accurate

table 1 Edge Detection Algorithms Comparison table

Chapter 2: Selected Algorithm with Python-Based PoC

Due to its **simplicity in hardware implementation**, the **Sobel edge detection algorithm** was chosen for this project. However, the version used here is **not the classic Sobel** — it has been extended with an element of **adaptivity**, meaning the algorithm can dynamically adjust its sensitivity to edges. To make the Sobel operator adaptive, a tunable **parameter** was introduced to control the contribution of weak gradients. When the parameter is set to **1**, the algorithm behaves like the classic Sobel. When the parameter is **very low**, weaker edges are heavily suppressed, resulting in a sparser edge map. When the parameter is **very high**, even weak gradients contribute strongly, which can increase noise or edge fuzziness. The proof-of-concept will demonstrate the impact of this adaptive control.

2.1 Python-Based PoC

This project presents a proof-of-concept (PoC) implementation of an adaptive Sobel edge detection algorithm using Python. The purpose of this PoC is to evaluate the feasibility of applying adaptive gradient scaling within the classic Sobel framework.

The code is divided into two main parts: the first part converts the input image from RGB to grayscale using a weighted sum of the color channels, and the second part performs the gradient magnitude calculation using Sobel operators in both the x and y directions. In this second stage, edge strength is adaptively scaled based on a tunable parameter, allowing for fine control over how weak and strong edges are treated.

In the grayscale conversion stage of the code, a slight modification was made to the standard RGB-to-grayscale weights to better suit future hardware implementation in SystemVerilog. The commonly used floating-point coefficients—r_const = 0.2126, g_const = 0.7152, and b_const = 0.0722—are accurate for perceptual luminance but not ideal for fixed-point arithmetic in hardware due to their high precision and non-integer nature.

To simplify hardware realization, these values were approximated using 8-bit fixed-point equivalents by multiplying each coefficient by 256 and rounding to the nearest integer. This yields r_const = 54, g_const = 183, and b_const = 18. When divided back by 256 to normalize in software, the effective weights become r_const ≈ 0.2109, g_const ≈ 0.7148, and b_const ≈ 0.0703. These approximated values maintain close accuracy to the original while being significantly more hardware-friendly, as they allow the grayscale computation to be implemented using integer multipliers and a simple bit-shift operation (>> 8) instead of floating-point division.

figure 13 Python code- Part 1 - converting from RGB to gray-scale

```
# PART II - Applying the Sobel operator
"""
my Gx and Gy kernels are:
      _               _                 _               _
     |                 |               |                 |
     | 1.0   0.0  -1.0 |               |  1.0   2.0   1.0 |
Gx = | 2.0   0.0  -2.0 |    and    Gy = |  0.0   0.0   0.0 |
     | 1.0   0.0  -1.0 |               | -1.0  -2.0  -1.0 |
     |_               _|               |_                _|
"""

# Here we define the matrices associated with the Sobel filter
Gx = np.array([[1.0, 0.0, -1.0], [2.0, 0.0, -2.0], [1.0, 0.0, -1.0]])
Gy = np.array([[1.0, 2.0, 1.0], [0.0, 0.0, 0.0], [-1.0, -2.0, -1.0]])
[rows, columns] = np.shape(grayscale_image)  # we need to know the shape of the input grayscale image
sobel_filtered_image = np.zeros(shape=(rows, columns))  # initialization of the output image array (all elements are 0)

# Now we "sweep" the image in both x and y directions and compute the output
for i in range(rows - 2):
    for j in range(columns - 2):
        gx = np.sum(np.multiply(Gx, grayscale_image[i:i + 3, j:j + 3]))  # x direction
        gy = np.sum(np.multiply(Gy, grayscale_image[i:i + 3, j:j + 3]))  # y direction
       # sobel_filtered_image[i + 1, j + 1] = np.sqrt(gx ** 2 + gy ** 2)
        gradient = np.sqrt(gx ** 2 + gy ** 2)
        scale_param=1  #adjust this value as needed

        sobel_filtered_image[i + 1, j + 1] = scale_param*gradient


# Display the original image and the Sobel filtered image
plt.figure(2)
plt.subplot(121)
#plt.figure(figsize=(12, 8), dpi=300)
plt.imshow(input_image)
plt.subplot(122)
plt.imshow(sobel_filtered_image, cmap='gray', vmin=0, vmax=255)
output_path = os.path.join(output_dir, 'sobel_adaptive_scale=1.png')
plt.savefig(output_path, bbox_inches='tight')
plt.show()
```

figure 14 Part 2- Appling the Sobel Operator

2.2 Results

The **first input image** used for evaluation is a structured image of a single car, representing a scene with one dominant object. The results presented include a comparison between the **original RGB image**, its **grayscale conversion**, and the output of the **adaptive Sobel algorithm** using various scale_param values. Specifically, the algorithm was tested with scale_param = 0.01, which—as expected—produced a very weak gradient and a darker image. This serves as a valuable **sanity check** for future hardware validation. Further results are shown for scale_param = 0.5, scale_param = 1 (which corresponds to the classic Sobel operator), and scale_param = 2. These experiments demonstrate that the parameter can be tuned according to the application's specific requirements, allowing flexible edge enhancement based on context.

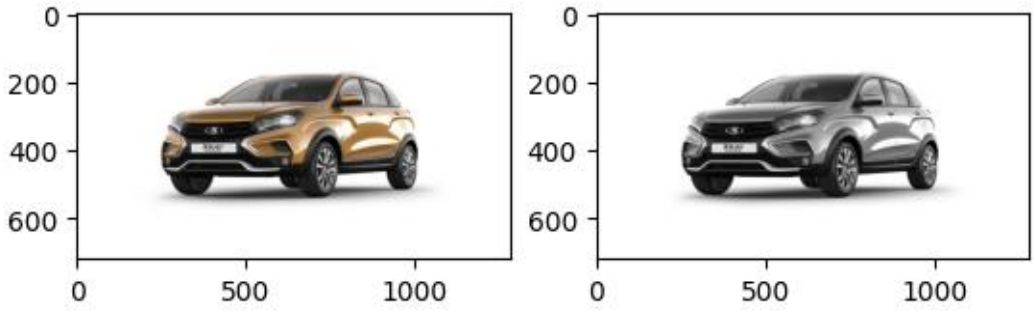


figure 15 The original RGB image (left) and its grayscale conversion (right)

Figure 16 presents the results of the Adaptive Sobel edge detection algorithm using four different scale_param values: 0.1, 0.5, 1 (classic Sobel), and 2.

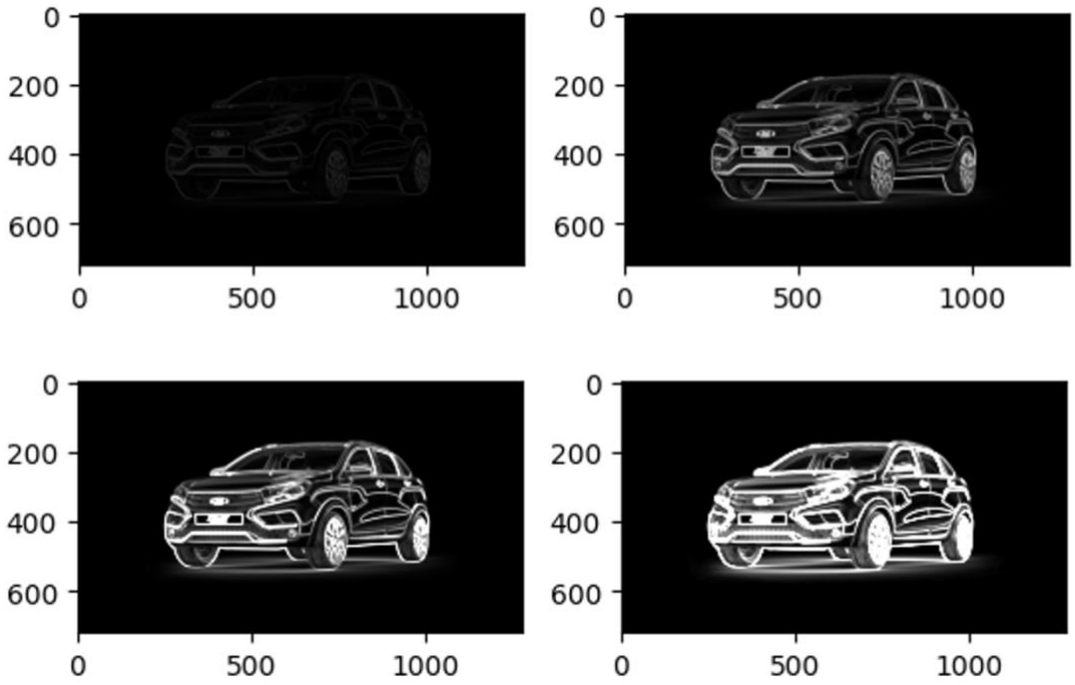


figure 16 Adaptive Sobel edge detection results with varying scale_param values. Top-left: 0.1, Top-right: 0.5 ,Bottom-left: 1, Bottom-right: 2.

The **second input image** is a more complex scene, depicting a highway with multiple vehicles and structural elements. Analyzing such a multi-object image is important to evaluate the behavior of the adaptive Sobel filter in more realistic and cluttered scenarios. As with the first image, the results were examined for scale_param = 0, 0.5, 1, and 2 to observe how the algorithm responds to varying levels of gradient sensitivity in more intricate scenes.

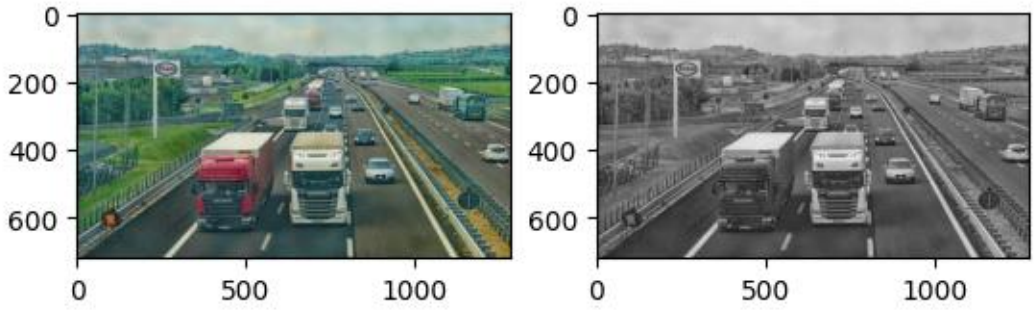


figure 17 The original RGB image (left) and its grayscale conversion (right)

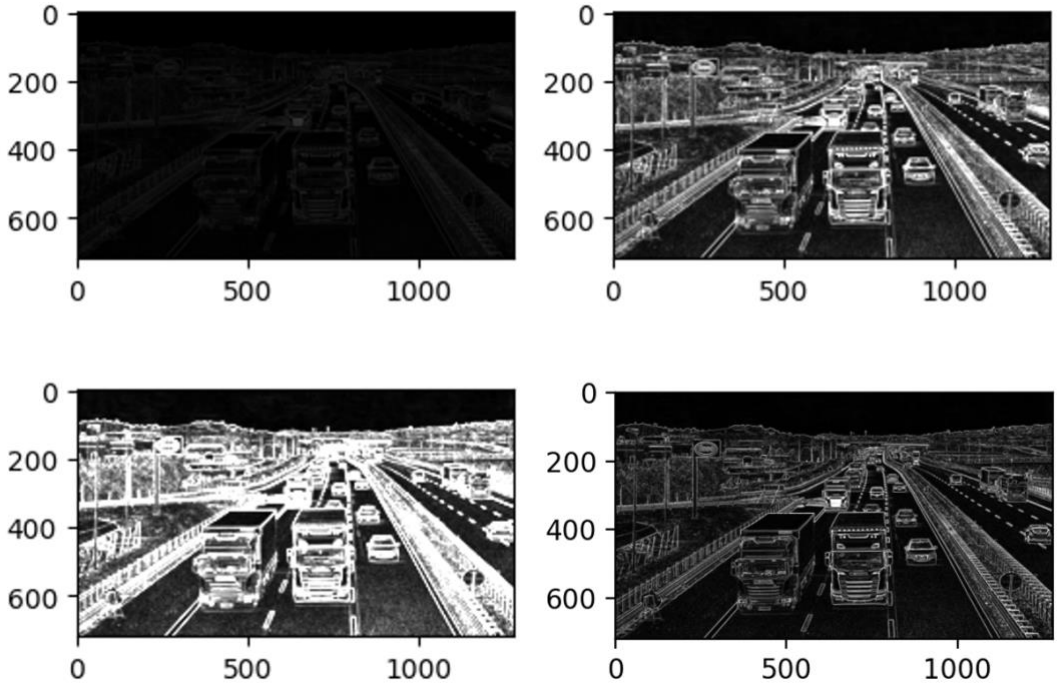


figure 18 Adaptive Sobel edge detection results with varying scale_param values. Top-left: 0.1, Top-right: 1 ,Bottom-left: 2, Bottom-right: 0.5.

Chapter 3: Hardware Design and Implementation

3.1 Interface and Functional Requirements

3.1.1 Global signals

The system shall receive Clock input signal with an appropriate frequency. The Clock frequency calculation presented in equation 1.

(1) $Clk = Image\ width \cdot\ Image\ height \cdot\ Frame\ rate = 1280 \cdot 720 \cdot 30 = 27.648\ [MHz]$

- The system shall receive Reset input signal. The reset signal should be **high active**, meaning that when it is high, the output will be '0'.

  In RGB, a '0' output means that all three-color components (red, green, and blue) are at their maximum values, which is 0. This results in the color **black**. So, if the reset signal is high, the entire image will be **black**.

3.1.2 Video processing

The Sobel Detection Algorithm will be used for the edge detection prosses. The edge detection block shall receive input video with a maximum frame rate of 30fps and a maximum resolution of 1280x720 pixels (HD), using AXI-Stream protocol.

Input/Output Color space: RGB is used for video encoding.

3.1.3 Memory

For efficient real-time edge detection processing using the Sobel algorithm, managing latency and memory utilization is critical. Given a typical video application running at 30 frames per second (fps), the maximum permissible latency between video input and output is approximately 33 milliseconds per frame. To achieve this latency constraint, an optimized buffering strategy is necessary.

Instead of buffering entire frames, which would require substantial on-chip memory (particularly for high-definition resolutions and above), this project implements a compact buffer system that stores only three lines of input pixels at a time. This cyclic buffer approach significantly reduces memory demands compared to full-frame storage solutions. Incoming pixels are stored sequentially in a vector-like manner—filling buffer rows from left to right before moving down to the next line. This method allows continuous and efficient processing as new pixel rows overwrite previously processed rows in a cyclic manner.

Additionally, by avoiding full-frame buffering, the design eliminates the need for external memory (such as off-chip storage), which would otherwise require utilizing AXI memory protocols. Fetching and writing entire frames through external memory interfaces would introduce substantial latency and complexity to the data path, negatively impacting real-time performance. The implemented three-line cyclic buffer efficiently addresses these constraints, ensuring high-speed processing, reduced on-chip memory usage, and compliance with stringent latency requirements.

Therefore, two separate buffers are utilized—one for input pixels and one for output pixels. The Sobel algorithm relies on convolution operations, which require accessing neighboring pixels around the current pixel. To preserve the integrity of neighboring pixels during processing, an additional output buffer is necessary. This ensures that input pixel data remains intact and unaffected while processed data is safely stored and prepared for subsequent stages, thus maintaining accurate edge detection results and real-time performance.

3.1.4 Configuration

The register configuration interface for parameterization of video settings and edge detection is AXI-Lite[1] (AXI4-Lite) protocol.

- Resolution default value set to 1280x720. (Also supports options to change to other video resolutions)

- Frame rate default value 30 fps.

- video input/output via AXI STREAM[2] protocol

3.2 Circuit Design

According to the design requirements, the architecture must include all necessary inputs and outputs to implement the AXI Stream protocol (TVALID, TDATA, TREADY) for both receiving and transmitting pixel data. Additionally, all inputs and outputs required for implementing the AXI Lite protocol must be provided to receive user-defined parameters, such as the frame size.

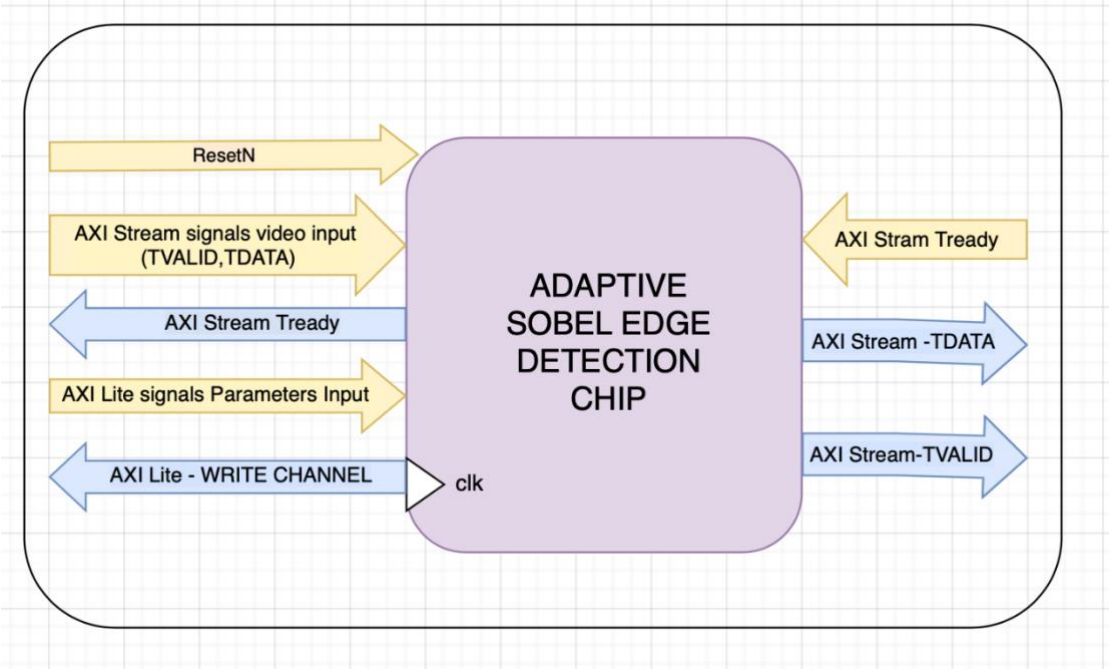


figure 19 Top hierarchy diagram

For convenience purposes some of the inputs in the code slightly renamed.

[1] AXI-Lite protocol (AXI4-Lite), 29.09.2023, https://developer.arm.com/documentation/ihi0022/latest//
[2] AXI-Stream protocol, 09.04.2021, https://developer.arm.com/documentation/ihi0051/b/?lang=en

```
module AdaptiveSobelTop (
    input  logic        clk,
    input  logic        rst,

    // AXI Stream Slave
    input  logic [31:0] s_axis_tdata,
    input  logic        s_axis_tvalid,
    output logic        s_axis_tready,
    input  logic        s_axis_tlast,

    // AXI Stream Master
    output logic [31:0] m_axis_tdata,
    output logic        m_axis_tvalid,
    input  logic        m_axis_tready,
    output logic        m_axis_tlast
);

    // Internal signals
    logic [7:0] r, g, b;
    logic [7:0] gray;
    logic [7:0] pi_out;
    logic [23:0] rgb_out;
    logic write_enable_gray;
    logic write_enable_rgb;
    logic internal_ready;

    // Unpack RGB input
    assign r = s_axis_tdata[23:16];
    assign g = s_axis_tdata[15:8];
    assign b = s_axis_tdata[7:0];

    // s_axis_tready now reflects internal ready signal based on master backpressure
    assign s_axis_tready = internal_ready;

    // RGB to GRAY
    RGB_to_GRAY u_rgb2gray (
        .resetN(~rst),
        .clk(clk),
        .r(r),
        .g(g),
        .b(b),
        .write_enable(s_axis_tvalid & internal_ready),
        .gray(gray),
        .write_enable_o(write_enable_gray)
    );

    // GRAY to RGB
    GRAY_to_RGB u_gray2rgb (
        .clk(clk),
        .resetN(~rst),
        .gray(pi_out),
        .write_enable(m_axis_tvalid),
        .rgb_out(rgb_out),
        .write_enable_o(write_enable_rgb)
    );

  // Valid-ready handshake block (within the Sobel edge detection)
    Sobel ready_valid (
        .clk(clk),
        .rst_n(~rst),
        .valid_in(write_enable_rgb),
        .ready_in(m_axis_tready),
        .valid_out(m_axis_tvalid),
        .ready_out(internal_ready),
         .pixel_in(gray),
         .pixel_out(pi_out)
    );
```

Figure 20 Top hierarchy block inputs/outputs

3.3 Algorithm Implementation

To perform adaptive Sobel edge detection on each frame while receiving it pixel by pixel, a hardware pipeline is implemented that processes each incoming pixel in real time. The pipeline converts the pixel to grayscale, buffers the necessary lines for neighborhood access, and dynamically applies the Sobel filter before outputting the edge-detected result.

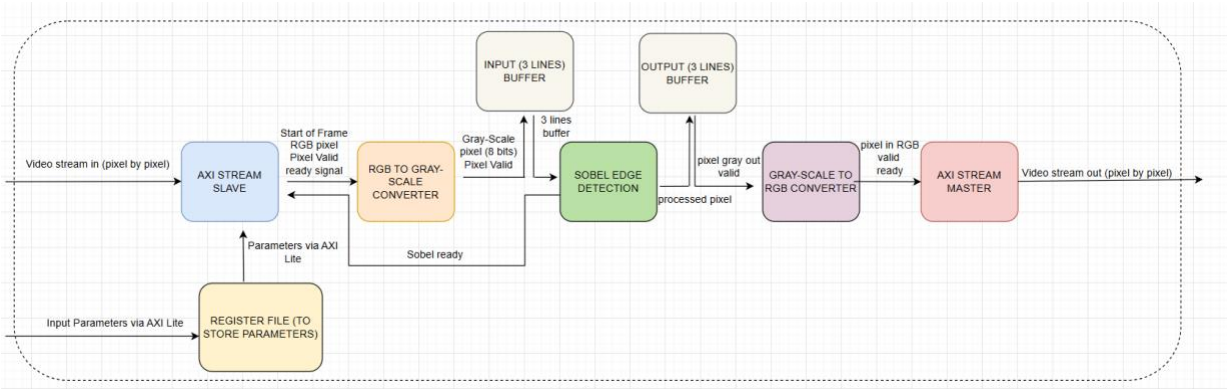


Figure 21 Block Diagram of Adaptive Sobel Edge Detector

REGISTER FILE:

Before describing the main pipeline, the Register File block is introduced. This block functions as a small internal memory used by the chip to store user-defined parameters. It operates as an AXI-Lite slave, allowing a user to write the size of the upcoming frame into its memory via an AXI-Lite master. The block includes all necessary inputs and outputs to support the AXI-Lite protocol. Upon receiving valid data and an address, it stores the data in the corresponding register.
The Register File also exposes its contents so that other blocks within the pipeline can access the stored parameters directly. In this implementation, only the AXI-Stream Slave at the start of the pipeline reads from it to determine the size of the next incoming frame.

AXI STREAM SLAVE:

The main pipeline starts with an instance of an Axi Stream Slave. This block will receive in a safe way the pixels transferred by the user (via an instance of an Axi Stream Master). It will also count how many pixels it has received for each frame. Every time the counter reaches the size of the current frame (according to the parameter given by the user), the counter will be set back to 0 and the next pixel will be treated as the start of the frame (a signal start_of_frame will be outputted). At the same time, this block will read the parameter currently stored at address 0 of the Register File and it will interpret it as the size of the next frame.

Buffers Block:

To perform Sobel edge detection, each pixel must be evaluated based on its 3×3 neighborhood. Since the video stream is received sequentially, a set of line buffers is used to temporarily store incoming pixels and construct the sliding window required by the filter. Specifically, three-line buffers are used to hold three consecutive rows of the frame, enabling access to the full 3×3 window for each pixel as the frame is streamed in.

This buffer structure allows real-time processing resolution frames by ensuring that every new pixel can be evaluated without waiting for the full frame to be received. As each new line is processed, the oldest line is discarded, and the new one is shifted into the buffer, maintaining continuous streaming and edge detection.

RGB to Grayscale Converter:

After receiving an RGB pixel, it is necessary to convert it to grayscale before applying Sobel edge detection. This block performs the conversion using a weighted sum of the red, green, and blue components based on the perceived luminance formula. The resulting grayscale intensity value is then passed to the buffer block for further processing.

Grayscale to RGB Converter:
After Sobel edge detection is performed on grayscale pixels, the resulting intensity values need to be converted back to RGB format for output display or further processing. This block replicates the grayscale value across the red, green, and blue channels to form a valid RGB pixel. The uniform replication preserves the edge information while ensuring compatibility with standard RGB output interfaces, such as the AXI Stream Master.

Sobel Block:

The Sobel block processes grayscale pixels using a 3×3 convolution window to compute edge strength. It relies on a rolling 3-line buffer for both input and output, which allows the module to operate line-by-line without requiring full-frame memory. After initially filling the first three lines of the frame, the module enters a cyclic mode where it continuously replaces the oldest line with a new incoming line, maintaining a sliding window of three rows. For each valid window, the Sobel operator is applied in both horizontal and vertical directions to compute gradient magnitudes. These magnitudes are scaled and clipped, then stored in the output buffer for the corresponding center line. The entire process is tightly synchronized with the upstream AXI Stream Slave interface using a handshake mechanism, ensuring that pixels are received only when the Sobel block is ready to process them.

AXI Stream Master:

Once the Sobel edge detection is completed for a pixel, the result is ready to be transmitted outside the chip. To accomplish this, the AXI Stream protocol is used. An

AXI Stream Master block is responsible for formatting the pixel data and transmitting it through the appropriate AXI-Stream interface. This ensures compatibility with external systems and enables efficient streaming of edge-detected video data.

3.1 Architecture of Each Block

AXI STREAM SLAVE:

```
module AxiStreamSlave
#(
    parameter MAX_SIZE_OF_FRAME = 400,
    localparam COUNTER_WIDTH = $clog2(MAX_SIZE_OF_FRAME)+1
)
(
    input logic clk,
    input logic resetN,
    input logic is_valid,
    input logic enable,
    input logic [31:0] pixel_in,
    input logic [COUNTER_WIDTH-1:0] size_of_frame,

    output logic is_ready,
    output logic pixel_valid,
    output logic start_of_frame,
    output logic [31:0] pixel_out,
    output logic [COUNTER_WIDTH-1:0] current_size_of_frame

);
//-------------------------------------------------------------------------------------------

reg [COUNTER_WIDTH-1:0] pixel_counter = 18'b0;  //pixel counter register


assign is_ready = enable;


always_ff@(posedge clk or negedge resetN) begin

    if (!resetN) begin
        pixel_counter <= 18'b0;
        start_of_frame <= 0;
        pixel_out <= 0;
        pixel_valid <= 1'b0;
        current_size_of_frame <= MAX_SIZE_OF_FRAME;
    end
    else if (is_ready && is_valid) begin

        pixel_out <= pixel_in;
        pixel_valid <= 1'b1;

        if(pixel_counter == 0) begin
            start_of_frame <= 1'b1;
            current_size_of_frame <= size_of_frame;
        end
        else begin
            start_of_frame <= 1'b0;
        end

        if (pixel_counter == current_size_of_frame-1) begin
            pixel_counter <= 0;
        end
        else begin
            pixel_counter <= pixel_counter+1;
```

Figure 22 AXI STREAM SLAVE SystemVerilog module code

Inputs:

- is_valid: Binary signal indicating that the input pixel provided by the user is valid.
- enable: Binary signal indicating whether the pipeline is allowed to progress.
- pixel_in: The RGB pixel currently provided by the external source.
- size_of_frame: The expected number of pixels in the current frame, read from the Register File.

Outputs:

- is_ready: Binary signal indicating that the slave is ready to receive the next pixel.
- pixel_valid: Binary signal forwarded to the next pipeline stage to indicate that the received pixel is valid.
- start_of_frame: Signal indicating that the current pixel marks the beginning of a new frame.
- pixel_out: Pixel received by the slave and forwarded to the next block in the pipeline.
- current_size_of_frame: The size of the current frame being processed.

The AXI Stream Slave block manages the reception of pixels from the external source using the AXI Stream protocol. The slave is considered ready to receive a new pixel whenever the pipeline is allowed to continue processing, as indicated by the enable signal from the final stage of the pipeline.

When both is_valid and is_ready are high, the block accepts the pixel through pixel_in and increments an internal pixel counter. Each time a new pixel is received, a pixel_valid signal is asserted to notify downstream modules that the pixel should be processed.

Once the internal counter reaches the specified size_of_frame, a start_of_frame signal is asserted during the next clock cycle to mark the beginning of a new frame. The expected frame size is continuously updated by reading from the Register File, allowing dynamic adaptation to varying frame dimensions.

RGB to Grayscale Converter:

```
module RGB_to_GRAY(
    input logic resetN,
    input logic clk,
    input logic [7:0] r,
    input logic [7:0] g,
    input logic [7:0] b,
    input logic write_enable,

    output logic [7:0] gray,
    output logic write_enable_o
);

logic [15:0] gray_calc;

always_comb begin
            if (write_enable) begin
                // Approximate:  gray ≈ 0.375*R + 0.5625*G + 0.03125*B\
              //instead of: gray = 0.299*R + 0.587*G + 0.114*B
    gray_calc = ((r << 2) + (r << 1) + g + (g << 3) + (b >> 1));
        gray = gray_calc >> 4;
            end
     write_enable_o = write_enable;  // just passes through
end



endmodule
```

Figure 23 RGB to Grayscale Converter SystemVerilog module code

Inputs:

- r: Red component of the input pixel.
- g: Green component of the input pixel.
- b: Blue component of the input pixel.
- input_pixel_valid: Binary signal indicating that the current input pixel is valid.
- write_enable: Binary signal indicating whether the pipeline is allowed to progress.
- input_start_of_frame: Binary signal indicating that the pixel marks the beginning of a new frame.
- size_of_frame_in: Size of the frame containing the current pixel.

Outputs:

• gray: Grayscale intensity computed from the RGB components using a weighted approximation.
• write_enable_o: Binary signal indicating that the output pixel is valid, directly passed through from the input write_enable. This block takes RGB values as input and computes the corresponding grayscale intensity using a weighted sum approximation.

This block takes RGB input values and computes the corresponding grayscale intensity using an integer-based approximation of the standard luminance formula. The approximation used is:

$$gray \approx 0.375 \cdot R + 0.5625 \cdot G + 0.03125$$

This is implemented efficiently in hardware as:

$$gray = \frac{(r \ll 2) + (r \ll 1) + g + (g \ll 3) + (b \gg 1)}{16}$$

The output write_enable_o is used to indicate when the grayscale output is valid and is directly passed through from the input write_enable.

Grayscale to RGB Converter:

```
module GRAY_to_RGB (
    input  logic clk,
    input  logic resetN,
    input  logic [7:0] gray,
    input  logic write_enable,

    output logic [31:0] rgb_out,
    output logic write_enable_o
);

always_comb begin
    if (write_enable) begin
        // RGB: {alpha, R, G, B} = {8'b0, gray, gray, gray}
        rgb_out = {8'b0, gray, gray, gray};
        write_enable_o = 1;
    end else begin
        rgb_out = 32'd255;
        write_enable_o = 0;
    end
end

endmodule
```

Figure 24 Grayscale to RGB Converter SystemVerilog module code

Inputs:

• gray: The 8-bit grayscale intensity value to be converted into an RGB format.
• write_enable: Binary signal indicating whether the current grayscale pixel is valid and should be processed.
• clk: Clock signal (included for consistency, though the logic is combinational).
• resetN: Active-low reset signal (present for interface uniformity, not used internally in this block).

Outputs:

• rgb_out: 32-bit RGB pixel output, formatted as {8'b0, gray, gray, gray} to match AXI Stream output (with zero-padded alpha channel).
• write_enable_o: Binary signal indicating that the RGB output pixel is valid, asserted only when write_enable is high.

This block takes a grayscale intensity input and replicates the value across the red, green, and blue channels to create a uniform RGB pixel. The resulting 24-bit RGB value is padded with 8 leading zeros to form a 32-bit word:

$$rgb\_out = \{8'b0, gray, gray, gray\}$$

This format is compatible with the AXI Stream protocol, which typically expects 32-bit wide data. When write_enable is low, the output is set to 32'd255 as a default value and write_enable_o is de-asserted, indicating that the output is not valid.

SOBEL:

```
module Sobel #(
    parameter threshold_scale = 1,
    parameter DATA_WIDTH      = 8,
    parameter FRAME_WIDTH     = 1280
)(
    input  logic clk,
    input  logic rst_n,

    input  logic [7:0] pixel_in,
    input  logic valid_in,
    output logic ready_out,

    output logic [7:0] pixel_out,
    output logic valid_out,
    input  logic ready_in
);

    logic [7:0] input_buffer [0:1279][0:2];
    logic [7:0] output_buffer [0:1279][0:2];

    logic iii = 1;
    integer jjj = 0;
    logic first_3_lines_full = 0;
    logic index_full_3_first =0 ;

    logic [10:0] col_counter;
    logic [1:0] row_counter;
    logic [1:0] row_counter_cyclic;

    //flags
    logic buffer_fill_done;
    logic output_start;
    logic [10:0] out_col_counter;
    logic [1:0] output_row_idx;   //used to iterate 0-1-2 during initial output

    logic [1:0] top_row_idx, mid_row_idx, bot_row_idx;

    logic [15:0] sum_sq_arr     [1:FRAME_WIDTH-2];
    logic [7:0]  sqrt_result_arr[1:FRAME_WIDTH-2];

    genvar i;
    generate
        for (i = 1; i < FRAME_WIDTH - 1; i++) begin : sqrt_blocks
            DW_sqrt #(.width(16)) u_sqrt (
                .a(sum_sq_arr[i]),
                .root(sqrt_result_arr[i])
            );
        end
    endgenerate
```

Figure 25 Sobel SystemVerilog module code

Inputs:
• clk: Clock signal for synchronizing sequential logic operations.
• rst_n: Active-low reset signal used to initialize internal state.
• pixel_in: 8-bit grayscale input pixel to be processed by the Sobel operator.
• valid_in: Indicates when the input pixel is valid and should be accepted by the block.
• ready_in: Handshake signal from the next stage in the pipeline, indicating it is ready to receive data.

Outputs:
• ready_out: Asserted when the Sobel block is ready to receive a new input pixel. Used as a handshake with valid_in.
• pixel_out: 8-bit output pixel representing the Sobel edge-detected value.

• valid_out: Indicates when the output pixel is valid and ready to be consumed by the next block.

```
        pixel_out          <= 8'd0;
    end else begin
        if (!buffer_fill_done) begin
            if (ready_out) begin
                if (col_counter == 1279) begin
                    col_counter <= 0;

                if (row_counter == 2 && first_3_lines_full == 0) begin
                    input_buffer[col_counter][row_counter]  <= pixel_in;
                    output_buffer[col_counter][row_counter] <= pixel_in;
                    row_counter_cyclic  <= 0;
                    first_3_lines_full  <= 1;
                    row_counter         <= 0;
                    buffer_fill_done    <= 1;
                    output_start        <= 1;
                    ready_out           <= 0;
                    valid_out           <= 1;

                    for (int col = 1; col < FRAME_WIDTH - 1; col++) begin
                        logic signed [10:0] gx, gy;
                        logic [21:0] gx_sq, gy_sq, sum_sq;
                        logic [15:0] scaled_magnitude;
                        logic [7:0]  sobel_out;

                        gx = -input_buffer[col-1][0] + input_buffer[col+1][0]
                           - 2 * input_buffer[col-1][1] + 2 * input_buffer[col+1][1]
                           - input_buffer[col-1][2] + input_buffer[col+1][2];

                        gy = -input_buffer[col-1][0] - 2 * input_buffer[col][0] - input_buffer[col+1][0]
                           + input_buffer[col-1][2] + 2 * input_buffer[col][2] + input_buffer[col+1][2];

                        gx_sq = gx * gx;
                        gy_sq = gy * gy;
                        sum_sq = gx_sq + gy_sq;

                        sum_sq_arr[col] = sum_sq[15:0];
                        scaled_magnitude = sqrt_result_arr[col] * threshold_scale;
                        sobel_out = (scaled_magnitude > 255) ? 8'd255 : scaled_magnitude[7:0];

                        output_buffer[col][1] = sobel_out;

                    end

                end else if (first_3_lines_full == 1) begin
                    top_row_idx <= mid_row_idx;
                    mid_row_idx <= bot_row_idx;
                    bot_row_idx <= (bot_row_idx == 2) ? 0 : bot_row_idx + 1;

                    row_counter_cyclic <= (row_counter_cyclic == 2) ? 0 : row_counter_cyclic + 1;

                    for (int i = 0; i < FRAME_WIDTH; i++) begin
                        input_buffer[i][bot_row_idx] <= pixel_in;
                    end

                    buffer_fill_done <= 1;
                    output_start     <= 1;
                    ready_out        <= 0;
                    valid_out        <= 1;
                    row_counter      <= 0;
```

Figure 26 Sobel SystemVerilog module code

This module performs Sobel edge detection on a grayscale video stream using a 3×3 convolution window. It receives one pixel per clock cycle using the AXI Stream-like valid/ready handshake, buffering the incoming data in a rolling 3-line buffer of size FRAME_WIDTH × 3.

Once the first three lines of the frame are buffered, the Sobel logic computes horizontal (Gx) and vertical (Gy) gradients using the classic Sobel masks, and then calculates the gradient magnitude:

$$magnitude = \sqrt{Gx^2 + Gy^2} * threshold\_scale$$

The square root is calculated using a DW_sqrt Synopsys IP core instantiated for each pixel column. If the scaled magnitude exceeds 255, it is saturated to 8'd255; otherwise, the least significant 8 bits are forwarded as pixel_out.

The block uses internal row and column counters to manage buffering and output sequencing. Initially, it processes and outputs the three middle lines of the first buffered frame. Then it continues processing one new row at a time, updating the buffers cyclically and streaming results for the corresponding middle row. The

ready_out signal is used to stall incoming pixels when the block is not ready to buffer more data, ensuring proper synchronization with the rest of the pipeline.

DW_sqrt (including the sqrt_function.inc):

```
module DW_sqrt (a, root);

  parameter integer width   = 8;
  parameter integer tc_mode = 0;

  input  [width-1 : 0]       a;
  output [(width+1)/2-1 : 0] root;

  // include modeling functions
`include "C:/Users/shira/OneDrive/desktop/VLSI/DW_sqrt_function.inc"

  initial begin : parameter_check
    integer param_err_flg;

    param_err_flg = 0;


    if (width < 2) begin
      param_err_flg = 1;
      $display(
    "ERROR: %m :\n  Invalid value (%d) for parameter width (lower bound: 2)",
    width );
    end

    if ( (tc_mode < 0) || (tc_mode > 1) ) begin
      param_err_flg = 1;
      $display(
    "ERROR: %m :\n  Invalid value (%d) for parameter tc_mode (legal range: 0 to 1)",
    tc_mode );
    end

    if ( param_err_flg == 1) begin
      $display(
        "%m :\n  Simulation aborted due to invalid parameter value(s)");
      $finish;
    end

  end // parameter_check



  assign root = (tc_mode == 0)?
          DWF_sqrt_uns (a)
              :
          DWF_sqrt_tc (a);

endmodule

//-------------------------------------------------------------------------------
```

Figure 27 DW_sqrt IP verilog code

Inputs:

• a: Unsigned input operand of bit width defined by the width parameter. Represents the number for which the square root will be computed.

Outputs:

• root: The computed square root of the input a, with a width of $\frac{(width+1)}{2}$ bits.

The DW_sqrt module used in this project is a pre-designed intellectual property (IP) core provided by Synopsys DesignWare. It is not a custom module developed as part of this project, but rather a trusted and optimized simulation model for computing the integer square root of a given input value. The module supports both unsigned and

two's complement (signed) input modes and includes parameter checks to ensure proper configuration. Internally, it relies on utility functions defined in the accompanying DW_sqrt_function.inc file. By leveraging this industry-standard IP block, we ensured correctness and efficiency in the square root calculation stage of the Sobel gradient magnitude computation.

AXI STREAM MASTER:

```
module axi_stream_master
    (
    input logic clk,
    input logic resetN,
    input logic pixel_valid,
    input logic t_ready,
    input logic [31:0]pixel_in,

    output logic t_valid,
    output logic [31:0]pixel_out,
    output logic enable

   );
 //-------------------------------------------------------------------------------------------

 // state machine declaration
    enum logic [2:0] {INIT_ST, VALID_ST, READY_ST, VALID_READY_ST} pres_st, next_st;

//-------------------------------------------------------------------------------------------
//    syncronous code:  executed once every clock to update the current state
always_ff@(posedge clk or negedge resetN)
   begin

   if ( !resetN )  // Asynchronic reset
       pres_st <= INIT_ST;

    else        // Synchronic logic FSM
       pres_st <= next_st;

    end // always sync

//-------------------------------------------------------------------------------------------
//asynchronous code
always_comb
    begin
    // set all default values
        next_st = pres_st;
        t_valid = 1'b0;
        pixel_out = 32'b0;
        enable = 1'b0;

        case (pres_st)

            INIT_ST: begin

                if(pixel_valid && t_ready) begin
                    next_st = VALID_READY_ST;
                end
                else if(pixel_valid) begin
                    next_st = VALID_ST;
                end
                else if(t_ready) begin
                    next_st = READY_ST;
                end
                else begin
                    next_st = INIT_ST;
                end

            end // idle
```

Figure 28 AXI stream out (master) SystemVerilog module code

Inputs:

- clk: Clock signal for synchronizing the FSM and output logic.
- resetN: Active-low reset signal that asynchronously resets the FSM to the initial

state.
• pixel_valid: Indicates that the incoming pixel (pixel_in) is valid and ready for transmission.
• t_ready: AXI Stream handshake signal from the receiver, indicating it is ready to accept data.
• pixel_in: 32-bit RGB pixel to be transmitted over the AXI Stream interface.

Outputs:

• t_valid: AXI Stream handshake signal asserted when the master is transmitting valid data.
• pixel_out: 32-bit pixel forwarded to the AXI interface when t_valid is high.
• enable: Control signal sent backward to allow upstream modules to provide the next valid pixel when the master is ready.

This block serves as an **AXI Stream Master** interface that transmits 32-bit RGB pixel data to a downstream module (e.g., display controller or VGA interface) using the AXI-Stream handshake protocol. The internal logic is governed by a finite state machine (FSM) with four states:

- INIT_ST: Idle state, waiting for either a valid pixel or readiness from the receiver.
- VALID_ST: A valid pixel is present, but the receiver is not yet ready. The module asserts t_valid and holds the pixel.
- READY_ST: The receiver is ready, but no valid pixel is currently available. The module asserts enable to prompt upstream data transmission.
- VALID_READY_ST: Both a valid pixel is present and the receiver is ready. The module asserts t_valid, forwards the pixel via pixel_out, and asserts enable to accept the next pixel.

The block ensures correct synchronization and data integrity by transitioning between these states based on pixel_valid and t_ready. When both signals are high, data is transferred in a single clock cycle. Otherwise, the FSM waits in either the VALID_ST or READY_ST until both conditions are met.
This design allows efficient and fully compliant data transfer over the AXI Stream protocol while coordinating with upstream logic using the enable signal.

3.4 Testing and Validation

A dedicated testbench was developed for each module to verify correct behavior under various use cases. The following section presents the results obtained from multiple simulation scenarios conducted for functional validation.

AXI STREAM SLAVE:

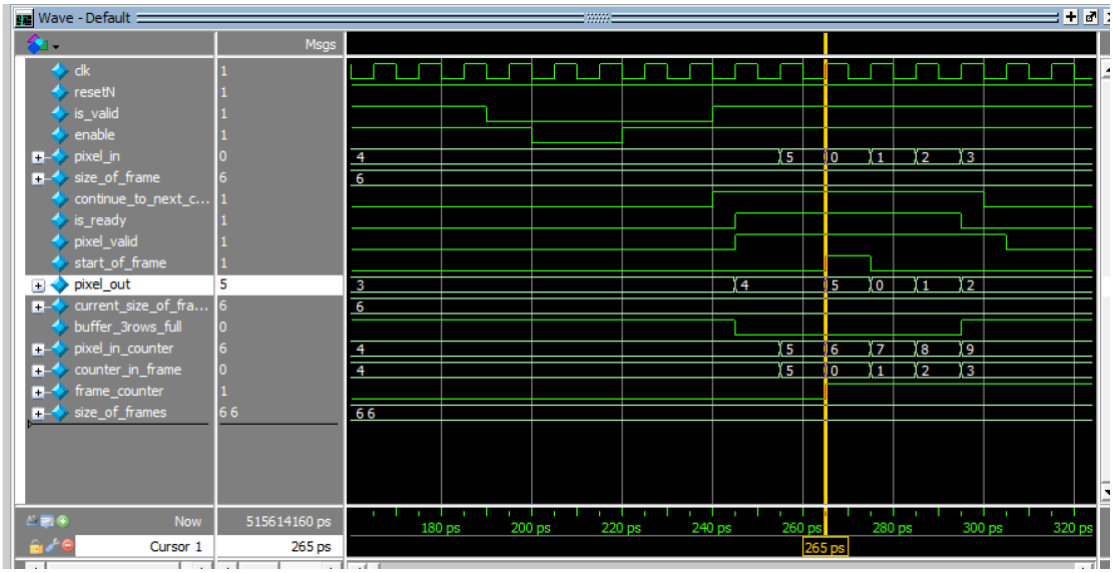


Figure 29 Axi Stream Slave simulation

As we can see in the figure above, when the frame size is set to 6. (two frames are being sent) The width is 1 pixel. After reset when first both is_ready and pixel_valid signal are high, we get the output valid is high after one clock cycle and with that the pixel_out is set to the pixel_in from the previous clock cycle. When the signal buffer_3rows_full is high we stop output the pixel_out and the pixel out valid is also set to low indicating the pixel_out is not valid to use.

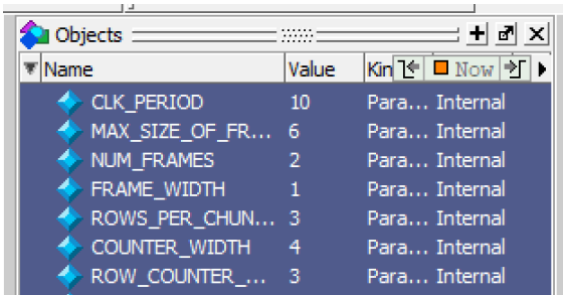


Figure 30 AXI Stream Slave simulation- parameters

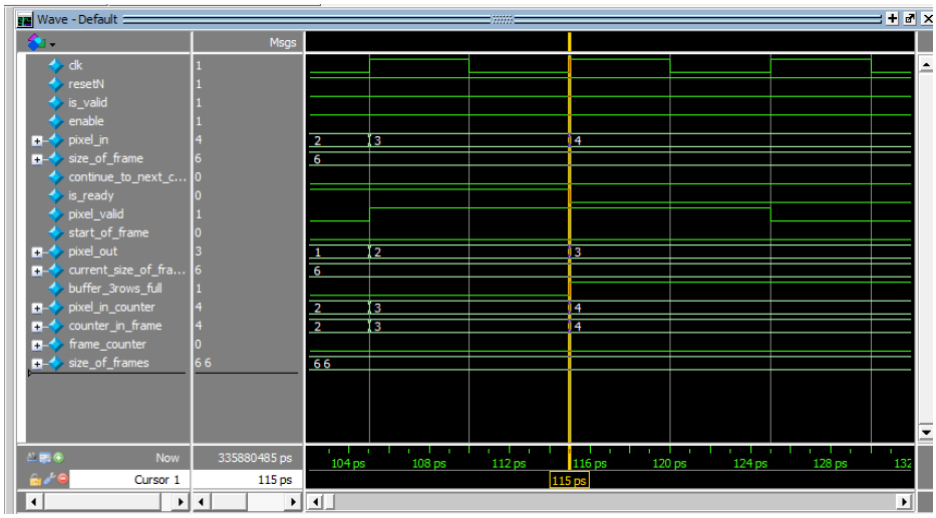


Figure 31 AXI Stream Slave simulation

At the beginning of the simulation, the frame size is set to 6. When both is_ready and is_valid signals are high, the pixel_valid signal asserts on the following clock cycle, and the frame size is updated accordingly. Since this is the first pixel of the frame, the start_of_frame signal is also asserted briefly and then de-asserted. As the simulation progresses, pixel transfers continue under the control of the sobel_ready_for_next_line signal, which indicates when the Sobel block has finished processing the current chunk and is ready to receive the next line. This ensures that the input buffer updates cyclically and maintains synchronization between pixel input and processing. When is_valid is temporarily de-asserted, the pipeline pauses, and no new pixels are transferred until the Sobel block signals readiness again.

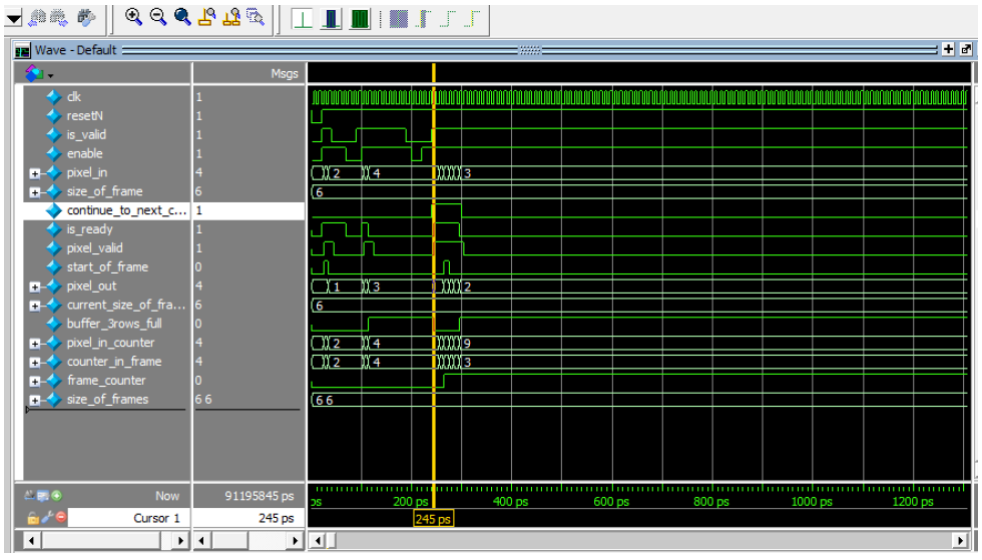


Figure 32 AXI Stream Slave simulation

RGB TO GRAY-SCALE CONVERTER:

The simulation results confirm that the RGB to Grayscale conversion block functions correctly, producing output values that match the expected weighted sum of the input RGB components. The design uses write_enable signals at both the input and output to regulate pixel transmission. The output is asserted only when the input write_enable is high, ensuring that pixel conversion and transfer occur only under proper control conditions. Frame-level synchronization is verified through the alignment of the start_of_frame signal between input and output.

We can observe specific examples in the simulation that further validate the correctness of the grayscale conversion. For instance:

- When in_R = 0, in_G = 0, and in_B = 0, the grayscale output is 0, as expected.
- When in_R = 0, in_G = 255, and in_B = 0, the output is 143. This corresponds
- When in_R = 0, in_G = 0, in_B = 255, the grayscale result is 7.
- When in_R = 255, in_G = 255, in_B = 255, the output is 247, close to the maximum as expected.
- When in_R = 100, in_G = 150, in_B = 200, the grayscale output is 128, which reflects a balanced contribution from all three channels.

These examples collectively demonstrate that the conversion logic performs as intended and that the control signals (write_enable) properly gate the pixel flow through the module.

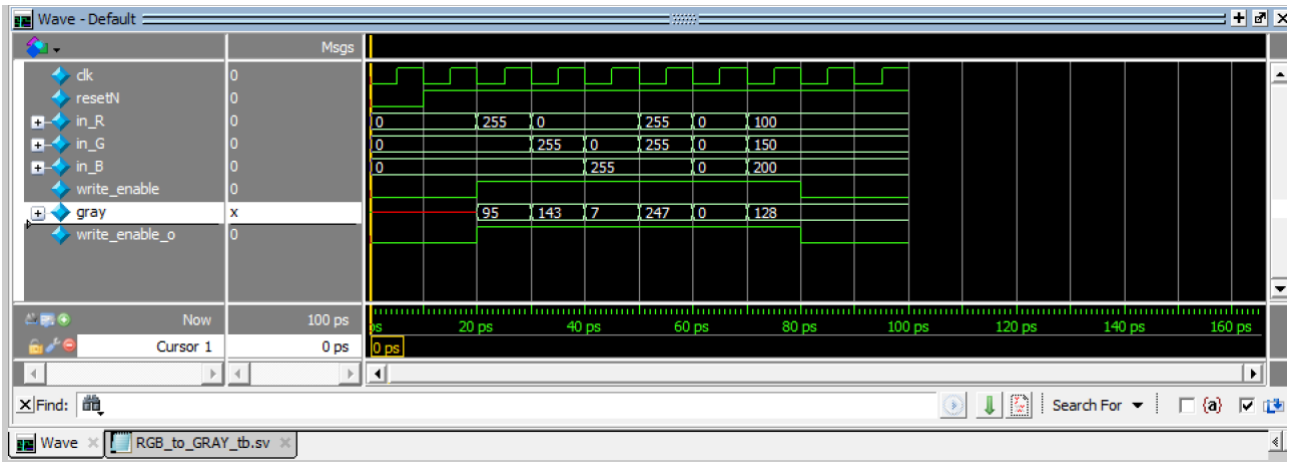


Figure 33 RGB to GRAY-SACLE simulations

GRAY-SCALE TO RGB CONVERTER:

The simulation results validate the correct functionality of the grayscale-to-RGB conversion block. This block replicates the 8-bit grayscale value across all three color channels—red, green, and blue—to produce a uniform RGB output. For example, when the grayscale input is 0, the output is 0x00000000, indicating that all RGB components are zero and the full 32-bit word is padded with zeros to match AXI Stream protocol requirements. Similarly, when the input is 55, the output is 0x00373737, where 0x37 (decimal 55) is duplicated across the red, green, and blue channels. The result is then aligned to 32 bits by masking with zeros in the upper byte, since AXI Stream requires data widths to be powers of two (in this case, 32 bits = 2^5).

This confirms that the grayscale-to-RGB conversion logic not only functions correctly but also adheres to the AXI Stream formatting constraints.

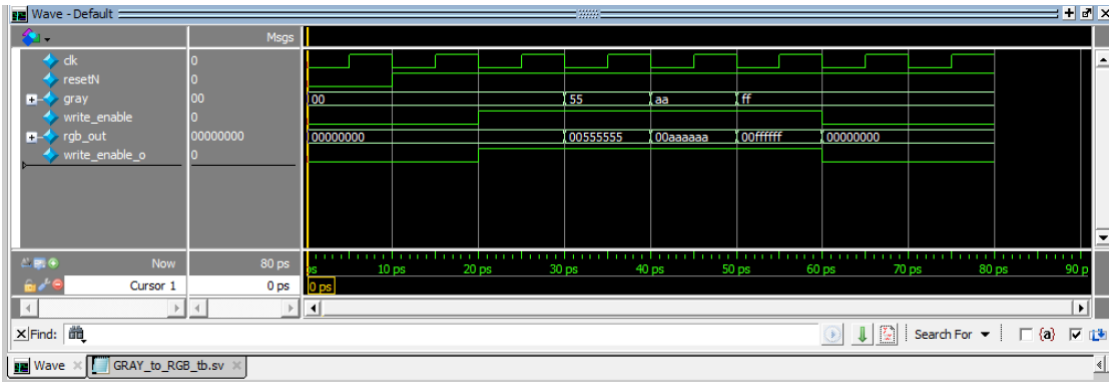


Figure 34 GRAY-SACLE TO RGB simulations

SOBEL BLOCK:

The simulation waveforms validate the correct operation of the Sobel block, particularly in its interaction with the 3-line input buffer. While the full frame-wise processing was primarily verified in the integrated top-level simulation, this block-level test focused on confirming that the Sobel operator correctly computes gradients for the middle line of the buffered data, which is where the 3×3 convolution window is centered during processing.

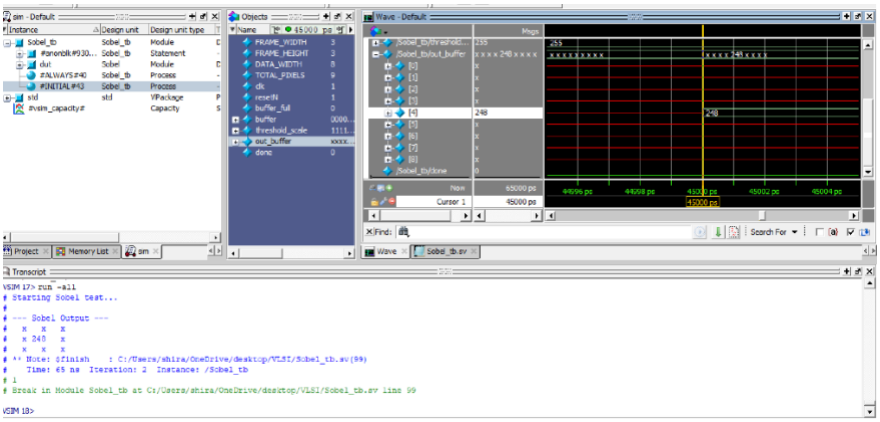


Figure 35 SOBEL 3x3 frame simulation

In the first test case, a small 3×3 image frame was transferred into the buffer. As expected, since the Sobel convolution computes the gradient magnitude using a 3×3 window, only the center pixel (i.e., the pixel at position [1][1]) can be processed. The simulation shows that the input buffer correctly holds all three rows, and the resulting output corresponds to the expected gradient based on the surrounding pixel values. The computed gradient values align with the expected results for both the Gx and Gy filters, and the final scaled magnitude output confirms proper functionality.

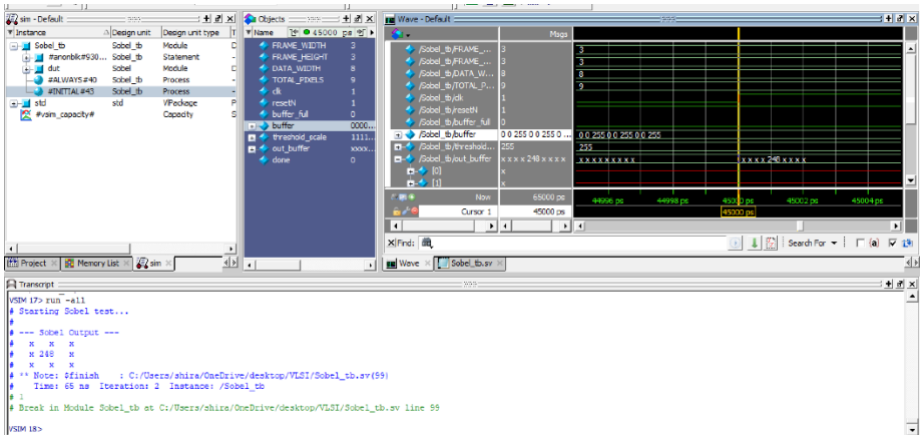


Figure 36 SOBEL 3x3 frame simulation

In the next test case, a larger 20×20 image frame was streamed in. As per the design logic, only three lines are held in the buffer at any given time (making it a 3×20 buffer), which is sufficient for line-by-line convolution. The simulation confirms that the buffer is correctly filled with a rolling window of three lines, and that the middle line is processed to compute the Sobel gradient. The output values on this middle line are consistent with expected edge responses, and the line-by-line advancement is gated correctly by the Sobel readiness signal. This test validates that the core Sobel logic processes pixel windows accurately within the constraints of limited internal buffering, and that the structure is scalable to full-frame processing when integrated with the top-level control and output logic.

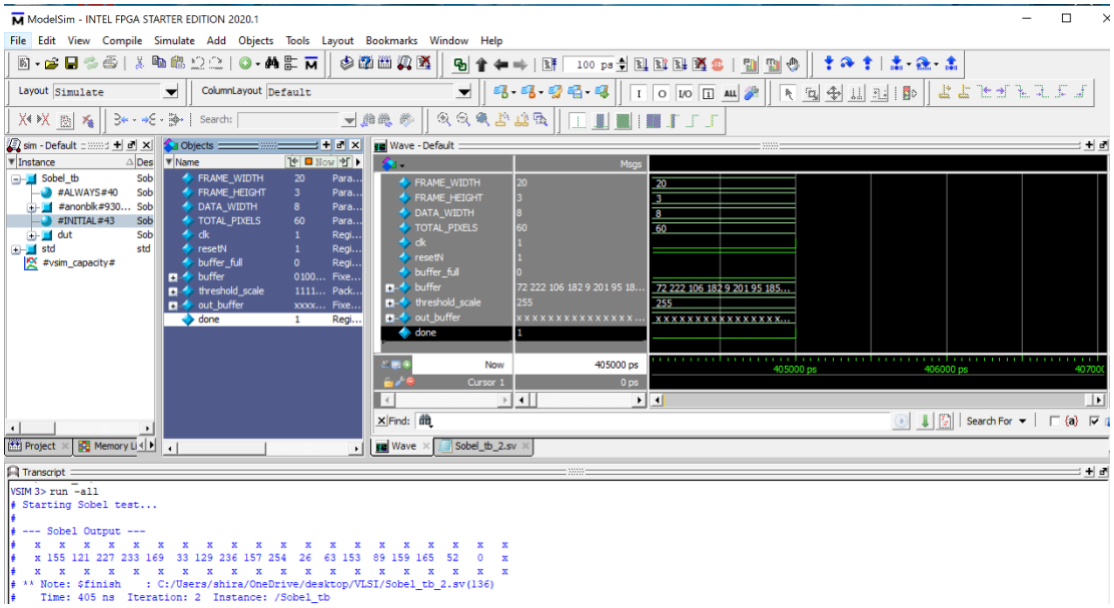


Figure 37 SOBEL 20x20 frame simulation

<u>BUFFERS (INPUT AND OUTPUT) BLOCKS:</u>

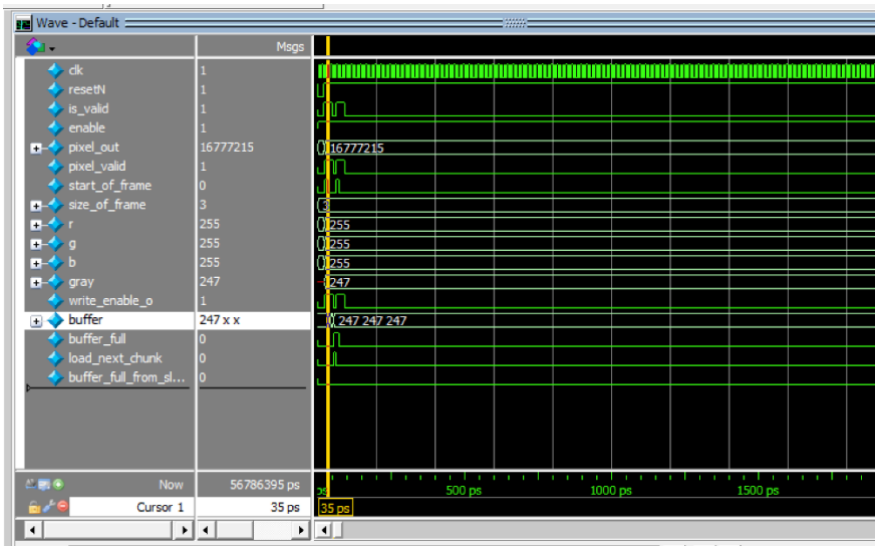


Figure 38 Buffers blocks simulations

The simulation waveforms for the buffer block illustrate its correct behavior in handling line-by-line pixel storage prior to Sobel processing. In this test, the frame width is set to 1 pixel per row, resulting in a buffer organized as a 3×1 structure. This simplified configuration allows for easier observation and verification of the buffer's filling mechanism.

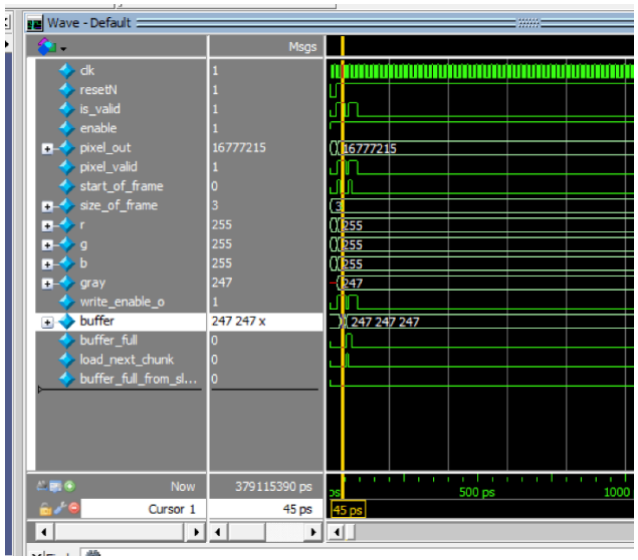


***Figure* 39** Buffers blocks simulations

As shown in **Figure 30**, the first row of the buffer is filled with the incoming pixel data. In **Figure 31**, the second row is subsequently written, and in **Figure 32**, the third and final row is stored. The simulation clearly shows the transfer of a pixel with value **247** through the buffer pipeline, verifying that the pixel data is correctly captured and retained.

Additionally, the buffer_full signal behaves as expected throughout the simulation. It remains low (0) while the buffer is still being filled. Once the third row is written—completing the buffer fill—the buffer_full signal is asserted high. This signal serves as

a control indication to the Sobel block that the buffer now holds sufficient data (for a full 3×3 window) to begin processing. This handshake mechanism ensures proper synchronization between the buffer and the Sobel filter, preventing premature computation and maintaining correct alignment of input data.

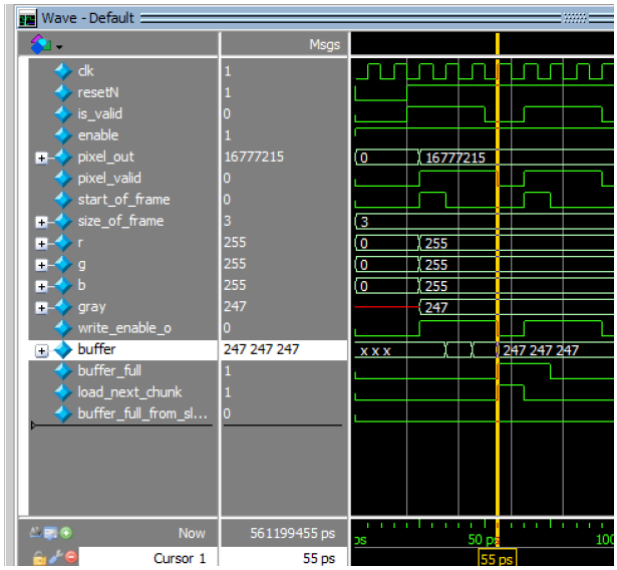


Figure 40 Buffers blocks simulations

AXI STREAM MASTER:

The simulation waveforms for the AXI Stream Master block confirm correct adherence to the AXI-Stream protocol, including the control of pixel transmission via the valid and ready handshake signals. As demonstrated in Figure 34, when the valid signal is low, no pixel is transferred—this is clearly reflected by the pixel_out signal holding steady or remaining undefined, indicating that the module is correctly blocking output when data is not valid.

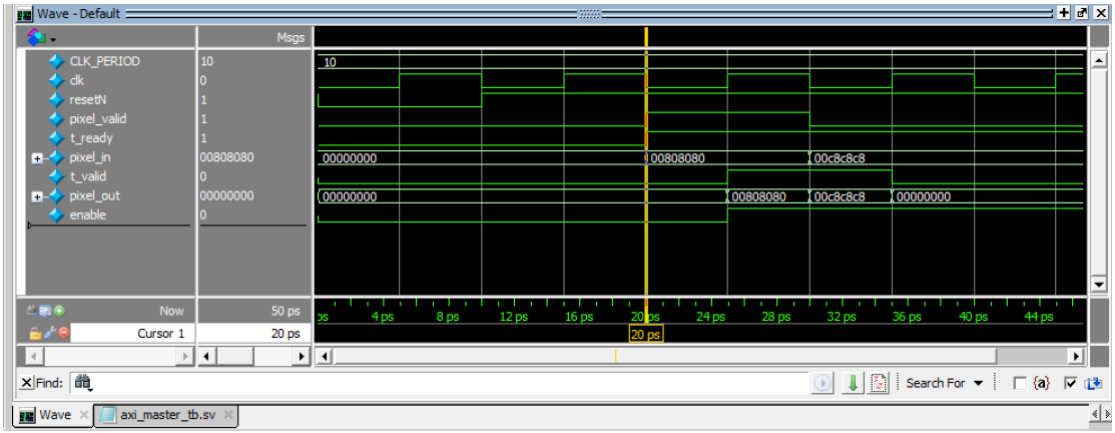


Figure 41 AXI Stream master simulations

In contrast, Figures 35 show cases where the valid signal is asserted 1 and the ready input is also high, completing the handshake. As a result, the pixel_out signal is actively driven, and data is successfully transferred to the connected display or VGA

output module. This confirms proper synchronization between the AXI Master and the downstream interface, enabling seamless pixel flow when both sides are ready.

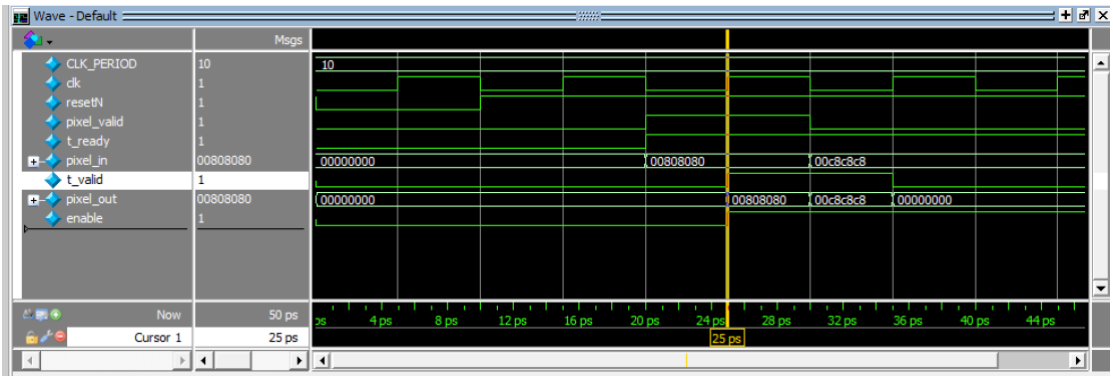


Figure 42 AXI Stream master simulations

Moreover, the simulation also illustrates the correct formatting of pixel data. The grayscale pixel input, replicated across red, green, and blue channels, forms a 24-bit RGB value. This 24-bit value is then aligned to the 32-bit AXI stream width by padding with zeros in the upper byte, as required by the AXI4-Stream protocol. This ensures compatibility with display controllers or external IPs that expect 32-bit wide stream data, while preserving the pixel intensity information.

These results collectively validate the AXI Stream Master block's functionality, including correct output gating, proper handshake control, and data formatting for high-resolution video output.

Chapter 4: Synthesis and Layout of the Design

4.1 Synthesis

After completing functional simulations of the design using ModelSim, I proceeded to perform synthesis using Synopsys tools.

(During this process, I encountered several challenges, primarily related to the complexity and register usage in the full design. While our architecture includes buffering mechanisms, it was carefully optimized to use only two 3-line buffers, which greatly reduced storage requirements compared to full-frame memory approaches.)

Due to the high computational load associated with synthesizing and optimizing the full HD-resolution line buffers, we adapted the design for synthesis and layout purposes by scaling down the frame size to a manageable 20×20 resolution. In this configuration, the internal line buffers are sized as 3×20 instead of 3×1280, significantly reducing memory usage and synthesis complexity. This modification was made to prevent excessive compile times and tool slowdowns during synthesis, which were observed when targeting the full HD frame size. The change preserves the structural behavior and modular hierarchy of the system while allowing for practical verification and resource-efficient backend flow.

To manage synthesis scope and reduce complexity, we focused on synthesizing critical components of the design, such as the RegisterFile block and the core image

processing modules, rather than the entire top-level system. This selective approach ensured better control over timing, area, and logic resource utilization.

For synthesis, we applied a clock constraint derived from the system requirements: 27.648 MHz so the corresponding to a clock period of approximately **36.18 ns**.

The synthesis process successfully generated the design netlist for use in place and route, along with detailed reports on timing, power, and area. Upon review of the timing analysis report, the synthesized blocks met the imposed constraints, validating the design's suitability. Despite initial concerns about timing closure, the chosen modules demonstrated compatibility with the required clock specification, confirming the viability of the hardware architecture.

Following the Timing Analysis Report of the Synthesis:

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : AdaptiveSobelTop
Version: U-2022.12
Date   : Tue May 13 17:11:09 2025
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: tsl18fs120_typ   Library: tsl18fs120_typ
Wire Load Model Mode: enclosed

  Startpoint: ready_valid/sum_sq_arr_reg_15__14_
              (falling edge-triggered flip-flop clocked by clk)
  Endpoint: ready_valid/output_buffer_reg_15__2__0_
            (falling edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  AdaptiveSobelTop   70000                 tsl18fs120_typ
  Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20
                     70000                 tsl18fs120_typ
  Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_3
                     4000                  tsl18fs120_typ

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock clk (fall edge)                                  18.08      18.08
  clock network delay (ideal)                             0.00      18.08
  ready_valid/sum_sq_arr_reg_15__14_/CPN (dfnfb1)         0.00 #    18.08 f
  ready_valid/sum_sq_arr_reg_15__14_/Q (dfnfb1)           0.39      18.48 f
  ready_valid/sqrt_blocks_15__u_sqrt/a[14] (Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_3)
                                                          0.00      18.48 f
  ready_valid/sqrt_blocks_15__u_sqrt/U29/ZN (nr02d1)      0.88      19.36 r
  ready_valid/sqrt_blocks_15__u_sqrt/U33/ZN (nr02d1)      0.21      19.57 f
  ready_valid/sqrt_blocks_15__u_sqrt/u_fa_PartRem_6_2/CO (ad01d1)
                                                          0.33      19.90 f
  ready_valid/sqrt_blocks_15__u_sqrt/U99/ZN (aoi21d1)     0.88      20.78 r
  ready_valid/sqrt_blocks_15__u_sqrt/U16/ZN (inv0d0)      0.87      21.65 f
  ready_valid/sqrt_blocks_15__u_sqrt/U22/ZN (nd12d1)      0.20      21.85 r
  ready_valid/sqrt_blocks_15__u_sqrt/U97/ZN (aon211d1)
                                                          0.17      22.02 f
  ready_valid/sqrt_blocks_15__u_sqrt/u_fa_PartRem_5_3/CO (ad01d1)
                                                          0.36      22.37 f
  ready_valid/sqrt_blocks_15__u_sqrt/U18/ZN (nd12d1)      0.31      22.69 f
  ready_valid/sqrt_blocks_15__u_sqrt/U9/ZN (inv0d1)       0.53      23.21 r
  ready_valid/sqrt_blocks_15__u_sqrt/U64/Z (xr02d1)       0.38      23.59 f
```

figure 43 Timing Analysis Report of the Synthesis

```
ready_valid/sqrt_blocks_15__u_sqrt/U11/ZN (inv0d0)      0.14      33.56 r
ready_valid/sqrt_blocks_15__u_sqrt/U85/ZN (aon211d1)
                                                        0.10      33.66 f
ready_valid/sqrt_blocks_15__u_sqrt/u_fa_PartRem_1_5/CO (ad01d1)
                                                        0.29      33.95 f
ready_valid/sqrt_blocks_15__u_sqrt/u_fa_PartRem_1_6/CO (ad01d1)
                                                        0.27      34.21 f
ready_valid/sqrt_blocks_15__u_sqrt/u_fa_PartRem_1_7/CO (ad01d1)
                                                        0.23      34.45 f
ready_valid/sqrt_blocks_15__u_sqrt/U6/ZN (nd12d1)       0.32      34.77 f
ready_valid/sqrt_blocks_15__u_sqrt/U4/ZN (inv0d0)       0.81      35.58 r
ready_valid/sqrt_blocks_15__u_sqrt/U30/ZN (nr02d1)      0.10      35.68 f
ready_valid/sqrt_blocks_15__u_sqrt/U59/ZN (oan211d1)
                                                        0.18      35.86 r
ready_valid/sqrt_blocks_15__u_sqrt/U31/ZN (inv0d0)      0.11      35.98 f
ready_valid/sqrt_blocks_15__u_sqrt/U57/CO (cg01d0)      0.19      36.17 f
ready_valid/sqrt_blocks_15__u_sqrt/U20/ZN (inv0d0)      0.13      36.30 r
ready_valid/sqrt_blocks_15__u_sqrt/U55/ZN (aon211d1)
                                                        0.12      36.42 f
ready_valid/sqrt_blocks_15__u_sqrt/U7/ZN (inv0d0)       0.16      36.58 r
ready_valid/sqrt_blocks_15__u_sqrt/U53/ZN (aon211d1)
                                                        0.13      36.71 f
ready_valid/sqrt_blocks_15__u_sqrt/U50/ZN (aoim21d1)
                                                        0.17      36.88 f
ready_valid/sqrt_blocks_15__u_sqrt/U49/ZN (aoi21d1)     0.13      37.01 r
ready_valid/sqrt_blocks_15__u_sqrt/U48/CO (cg01d0)      0.22      37.24 r
ready_valid/sqrt_blocks_15__u_sqrt/U12/ZN (nd02d1)      0.10      37.33 f
ready_valid/sqrt_blocks_15__u_sqrt/U46/ZN (aon211d1)
                                                        0.18      37.51 r
ready_valid/sqrt_blocks_15__u_sqrt/U45/ZN (oai211d1)
                                                        0.10      37.61 f
ready_valid/sqrt_blocks_15__u_sqrt/root[0] (Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_3)
                                                        0.00      37.61 f
ready_valid/U822/ZN (inv0d0)                            0.35      37.96 r
ready_valid/U3809/ZN (oai22d1)                          0.10      38.06 f
ready_valid/output_buffer_reg_15__2__0_/D (dfnfb1)      0.00      38.06 f
data arrival time                                                 38.06

clock clk (fall edge)                                  54.25      54.25
clock network delay (ideal)                             0.00      54.25
ready_valid/output_buffer_reg_15__2__0_/CPN (dfnfb1)
                                                        0.00      54.25 f
library setup time                                     -0.18      54.08
data required time                                                54.08
--------------------------------------------------------------------------
data required time                                                54.08
data arrival time                                                -38.06
--------------------------------------------------------------------------
slack (MET)                                                       16.02
```

figure 44 Timing Analysis Report of the Synthesis

As can be observed the slack (the difference between the required time and arrival time) is positive, so the path isn't violating the setup time.

Following the Power Analysis Report:

```
Loading db file '/tools/kits/tower/PDK_TS18SL/FS120_STD_Cells_0_18um_2005_12/DW_TOWER_tsl18fs120/2005.12/synopsys/2004.12/models/tsl18fs120_typ.db'
Information: Propagating switching activity (low effort zero delay simulation). (PWR-6)
Warning: Design has unannotated primary inputs. (PWR-414)
Warning: Design has unannotated sequential cell outputs. (PWR-415)

****************************************
Report : power
        -analysis_effort low
Design : AdaptiveSobelTop
Version: U-2022.12
Date   : Tue May 13 17:12:31 2025
****************************************


Library(s) Used:

    tsl18fs120_typ (File: /tools/kits/tower/PDK_TS18SL/FS120_STD_Cells_0_18um_2005_12/DW_TOWER_tsl18fs120/2005.12/synopsys/2004.12/models/
tsl18fs120_typ.db)


Operating Conditions: tsl18fs120_typ   Library: tsl18fs120_typ
Wire Load Model Mode: enclosed

Design        Wire Load Model            Library
------------------------------------------------
AdaptiveSobelTop       70000             tsl18fs120_typ
RGB_to_GRAY            4000              tsl18fs120_typ
GRAY_to_RGB            ForQA             tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20
                       70000             tsl18fs120_typ
RGB_to_GRAY_DW01_add_0_DW01_add_52
                       ForQA             tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_0
                       4000              tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_1
                       4000              tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_2
                       4000              tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_3
                       4000              tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_4
                       4000              tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_5
                       4000              tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_6
                       4000              tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_7
                       4000              tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_8
                       4000              tsl18fs120_typ
Sobel_threshold_scale1_DATA_WIDTH8_FRAME_WIDTH20_DW_sqrt_9
                       4000              tsl18fs120_typ
```

figure 45 Power Analysis Report of the Synthesis

```
Global Operating Voltage = 1.8
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


Attributes
----------
i - Including register clock pin internal power


  Cell Internal Power  =   3.2701 mW   (88%)
  Net Switching Power  = 428.3462 uW   (12%)
                         ---------
Total Dynamic Power    =   3.6985 mW  (100%)

Cell Leakage Power     =   1.2545 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000            0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000            0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000            0.0000             0.0000  (   0.00%)
clock_network      2.8943           0.0000            0.0000             2.8943  (  78.23%)  i
register       7.5600e-03        1.4393e-02        1.7719e+05         2.2126e-02  (   0.60%)
sequential     3.6289e-03        4.5767e-04          725.2160         4.0873e-03  (   0.11%)
combinational      0.3646           0.4135        1.0766e+06             0.7792  (  21.06%)
--------------------------------------------------------------------------------------------------
Total              3.2701 mW        0.4283 mW     1.2545e+06 pW         3.6997 mW
1
```

figure 46 Power Analysis Report of the Synthesis

The post-synthesis power report indicates that the design consumes a total power of **3.6997 [mW]**, comprising **3.2701 [mW]** of internal power and **0.4283 [mW]** of switching power. These results demonstrate that the design is highly efficient and power-conservative, particularly given the computational nature of the Sobel-based

image processing pipeline. The low switching activity and well-optimized logic contribute to minimal dynamic power dissipation, making the implementation suitable for energy-constrained or embedded applications.

4.2 Layout

Chapter 5: Results

Rigorous performance evaluations of the Sobel edge detection hardware were conducted specifically using high-definition (HD) resolution frames. In total, six tests were conducted, with three distinct threshold scale parameters applied to each of the two frames tested.

Frame 1: A relatively simple frame containing a single object—a clear image of a car located centrally within the frame, with minimal complexity aside from a slight shadow beneath the car. Three tests were conducted using different threshold scales to evaluate the adaptive performance of the Sobel implementation.

Before Adaptive Sobel edge detection:

figure 47 frame 1 as an input- A car

Test 1-Threshold Scale = 1 (Classic Sobel): This parameter represents the standard Sobel algorithm without scaling adjustments. The edges detected were clear and accurately outlined the car's boundary, demonstrating the classic Sobel's baseline performance.

figure 48 The ouput after SV test for frame 1 - Threshold=1 (classic Sobel)

Test 2-Threshold Scale = 0.5: Lowering the threshold scale decreased the edge magnitude sensitivity, resulting in fewer edges appearing. This also resulting in thinner, less pronounced edges. Finer details became less noticeable, emphasizing only the strongest intensity transitions. Only the most pronounced edges remained visible, leading to a simpler, cleaner edge image with less detail.

figure 49 The ouput after SV test for frame 1 - Threshold=2 (classic Sobel)

Test 3-Threshold Scale = 2: Increasing the threshold scale enhanced edge detection sensitivity, resulting in more pronounced and numerous edges. This highlighted additional fine details and texture variations within the image.

figure 50 The ouput after SV test for frame 1 - Threshold=2 (classic Sobel)

This set of tests confirms the adaptive nature of the implementation, showcasing how different scaling parameters influence edge visibility and detection quality.

Additionally, Figure 16 in section 2.2 illustrates the outputs from the Python proof-of-concept (PoC), demonstrating the high similarity between the hardware SystemVerilog

implementation outputs and the Python-generated frames. This comparison further validates the correctness and accuracy of the hardware logic design.

Frame 2: A more complex frame depicting a multi-lane road populated with several cars and richer RGB color diversity compared to Frame 1. The higher complexity of this frame makes the results of the Sobel edge detection particularly insightful, as detecting precise edges within a complicated scene has practical relevance for various real-world applications.

Before Adaptive Sobel edge detection:

figure 51 Original Frame 2 – Input (Highway Scene)

Test 1- Threshold Scale = 1 (Classic Sobel): Applying the standard Sobel algorithm produced clear and distinct edges, effectively delineating vehicles, lane markings, and other critical elements within the frame.

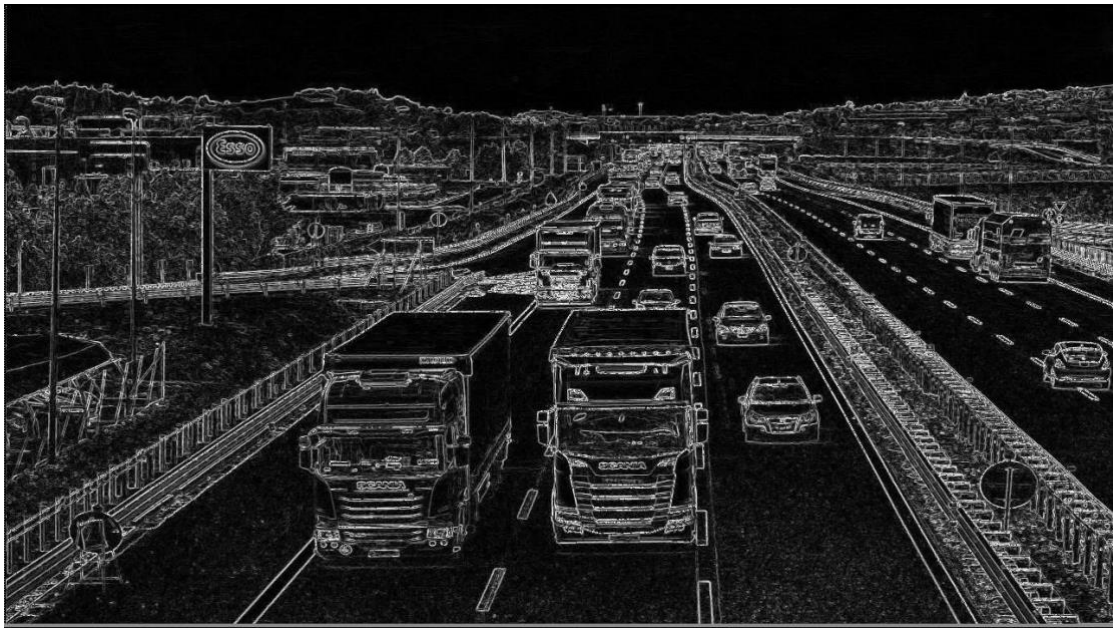


figure 52 Output of frame 2– Threshold Scale = 1 (Classic Sobel)

Test 2- Threshold Scale = 0.5: Reducing the threshold scale decreased the sensitivity to edge magnitude, resulting in fewer and less detailed edges. This simplified the frame by retaining only the most significant edges and thereby reducing visual complexity.

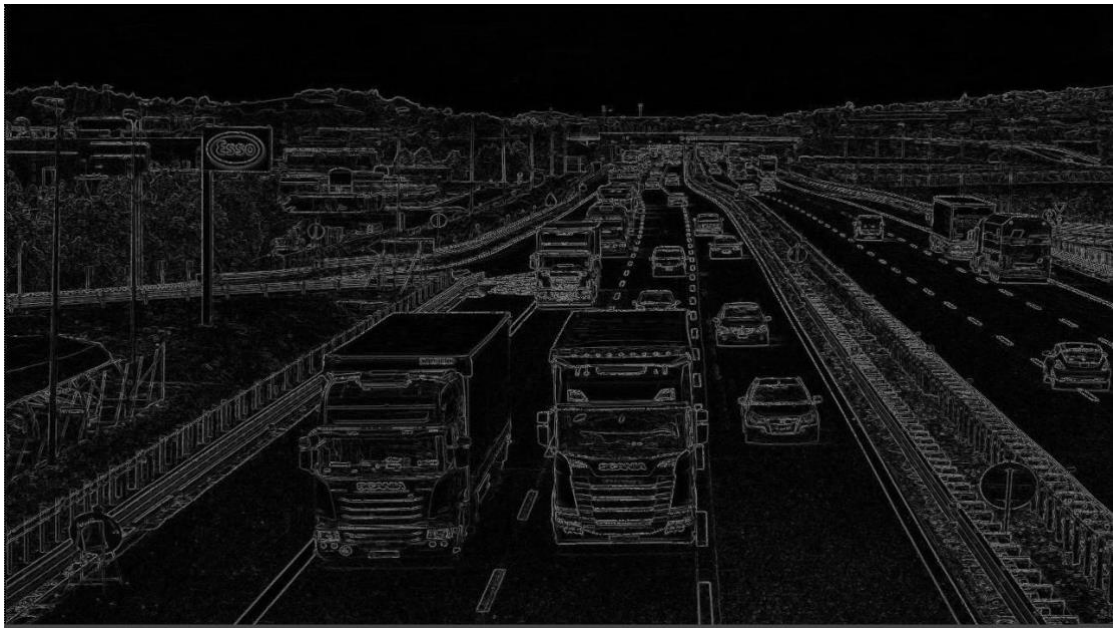


figure 53 Output of frame 2– Threshold Scale = 0.5

Test 3- Threshold Scale = 2: Increasing the threshold scale significantly enhanced edge sensitivity, capturing a greater number of edges, including finer details and subtle texture variations. This provided a more detailed representation of the complex scene, useful for further analysis.

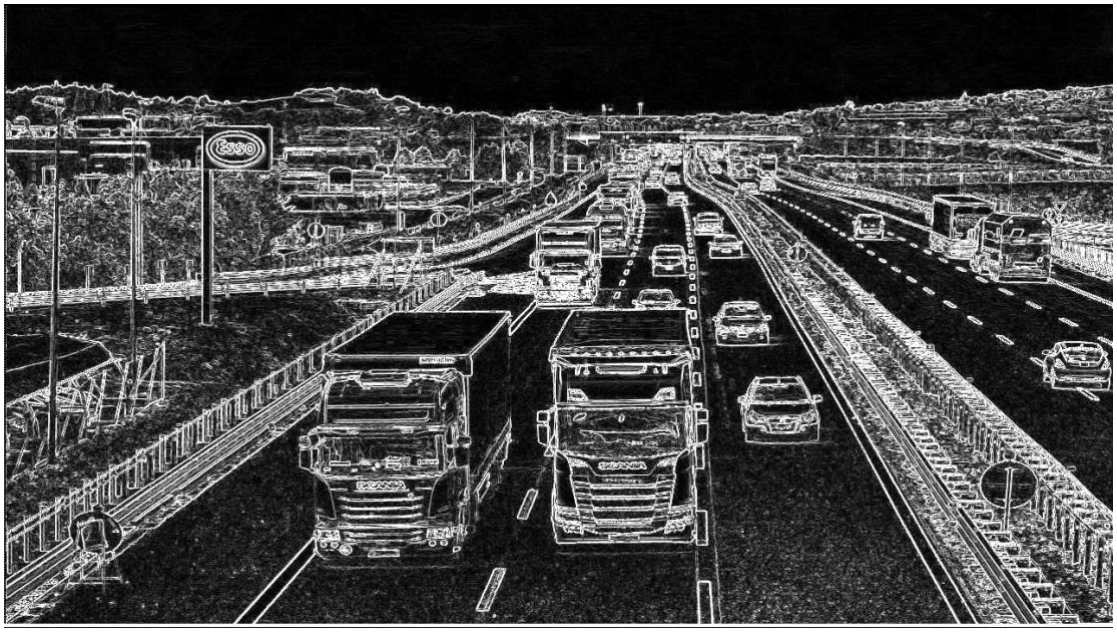


figure 54 Output of frame 2– Threshold Scale = 2

Figure 19 demonstrates the similarity between the SystemVerilog hardware implementation outputs and the Python-generated outputs for Frame 2, further reinforcing the accuracy and reliability of the hardware design.

5.1 Discussion

During the development of the adaptive Sobel edge detection project in SystemVerilog, several noteworthy challenges and design issues emerged. These challenges are worth presenting not only as technical hurdles that were overcome but also as valuable learning opportunities regarding hardware-oriented thinking, resource planning, and the constraints imposed by real-time processing architectures.

A key difference between SystemVerilog and high-level programming languages lies in the nature of hardware description. Unlike software, hardware must operate with fixed-width binary representations and deterministic timing. This constraint required certain approximations during implementation. For example, in the RGB to Grayscale conversion block, the standard formula involving real-valued coefficients could not be directly used. Instead, a binary-friendly approximation of the grayscale equation was employed, using integer multipliers and bit shifts to preserve resource efficiency while maintaining acceptable image quality.

Another significant consideration was memory usage. Initially, the design was intended to interface with AXI-based external memory to store image frames. However, this approach was later abandoned. Integrating external memory significantly increased latency and imposed complex synchronization requirements, which would have slowed down the edge detection pipeline. By redesigning the

architecture to avoid AXI memory and rely solely on internal line buffers, the design became more streamlined and real-time-capable.

Additionally, careful planning was required for coordinating the AXI Stream Slave interface with the Sobel block. Since the Sobel filter operates on a sliding 3×3 window, proper timing between input line availability and processing readiness was essential. This required handshake-driven communication between the AXI Slave and the Sobel module to ensure correct line sequencing, especially after the initial buffering phase.

One specific problem encountered during development involved the use of three-line buffers for both input and output. Initially, each time the Sobel module signaled readiness, the AXI slave provided three new lines of pixel data. However, this led to incorrect behavior: only the middle line in each 3-line window was being updated by the Sobel filter, resulting in gradient computation being effectively limited to the central row of each block. This issue manifested visually in the output image as repetitive bands, where only alternating rows showed edge responses.

Figure 55 Output Image Showing Line Buffering Artifact Before Cyclic Update Fix

To solve this, an unconventional yet effective approach was adopted. Instead of replacing all three lines after each Sobel operation, the design was modified to **cyclically update only one line at a time**. Once the initial three lines were buffered, each subsequent processing cycle involved replacing just the oldest line with a new one from the AXI slave. This maintained a continuously sliding window, ensuring proper 3×3 convolution across all lines. A similar cyclic mechanism was also applied to the output buffer to preserve alignment. This change resolved the issue, as evidenced in the figure below, where the previously observed white stripes (resulting from static gradients in unchanged regions) were eliminated, indicating proper gradient calculation across the image.

In the case of border pixels, where the Sobel filter cannot access a complete 3×3 neighborhood (e.g., at the top, bottom, or image edges), the design does not compute

the gradient and instead outputs a zero value. This effectively renders the borders as black in the output image, which simplifies the hardware logic and avoids undefined behavior due to missing data. Since the design processes HD-resolution images or similarly large frames, the affected pixels are confined to a single row or column along each edge, making the effect visually negligible and practically unnoticeable in the final output.

Chapter 6: Conclusion

6.1 Summary of Project Achievements

The project successfully designed and implemented a hardware-based adaptive Sobel edge detection pipeline capable of receiving high-definition (HD) video input pixel-by-pixel via the AXI Stream protocol and support various resolutions. The system performs real-time edge detection using a configurable threshold scaling parameter, allowing dynamic adjustment of edge sensitivity. The pipeline includes grayscale conversion, line buffering, convolution using the Sobel operator, and output formatting, all synchronized through handshake mechanisms between blocks. The design supports frame resolutions, including 720×1280, and processes pixels at a clock frequency needed measured in MHz, enabling real-time video rates of 30 frames per second (fps). The output edge-enhanced frames exhibit improved structural contrast, emphasizing object boundaries effectively for further analysis or visualization.

6.2 Implications and Future Works

Edge detection is a foundational task in many computer vision applications. The implemented adaptive Sobel detector can be integrated into broader vision pipelines, especially in fields that require high-throughput and low-latency processing, such as autonomous vehicles, surveillance systems, and industrial inspection.

In the medical domain, this hardware solution can be particularly beneficial for real-time enhancement of diagnostic images, such as X-rays or MRI scans. For example, bone structures in X-ray images can be better emphasized using edge enhancement, aiding in early detection of fractures or abnormalities. The low-latency nature of the hardware implementation makes it suitable for embedded medical imaging systems. Looking forward, several extensions can be made to this project.

One promising direction is to integrate this module as a preprocessing stage in Canny edge detection, since Canny relies on Sobel filtering for gradient estimation. Offloading the Sobel stage to hardware could significantly improve the overall throughput of Canny-based systems.

Another enhancement involves incorporating adaptive filters or denoising stages before the edge detection, which could reduce false edges in noisy environments.

Additionally, implementing a closed-loop feedback mechanism that adjusts the threshold scaling dynamically based on local contrast could yield sharper and more adaptive results across varying image regions.

These improvements could expand the system's utility in both embedded real-time systems and scalable FPGA-based processing platforms.

Bibliography

[1] S. Leung, "Innovative flexible displays in modern electronic devices," *LinkedIn*, Aug. 30, 2024. [Online]. Available: https://www.linkedin.com/pulse/innovative-flexible-displays-modern-electronic-devices-shirley-leung-lbqqc.

[2] R. H. Friend, R. W. Gymer, A. B. Holmes, J. H. Burroughes, R. N. Marks, C. Taliani, D. D. C. Bradley, D. A. Dos Santos, J. L. Brédas, M. Löegdlund, and W. R. Salaneck, "Electroluminescence in conjugated polymers," *Nature*, vol. 397, no. 6715, pp. 121–128, Jan. 1999. [Online]. Available: https://www.nature.com/articles/16393.pdf

[3] Ruitao Su *et al.,* 3D-printed flexible organic light-emitting diode displays.*Sci. Adv.***8** ,eabl8798 (2022), 10.1126/sciadv.abl8798

[4] Chen, HW., Lee, JH., Lin, BY. *et al.* Liquid crystal display and organic light-emitting diode display: present status and future perspectives. *Light Sci Appl* **7**, 17168 (2018). https://doi.org/10.1038/lsa.2017.168

[5] Ilya I. Khalyapin, Natalia A. Mamedova, Huaming Zhang, Arkadiy I. Urintsov; Applying the CI/CD practices to mobile product development. *AIP Conf. Proc.* 24 February 2025; 3268 (1): 020003. https://doi.org/10.1063/5.0258386

[6] Kwak, K., Cho, K. & Kim, S. Stable Bending Performance of Flexible Organic Light-Emitting Diodes Using IZO Anodes. Sci. Rep. 3, 2787; DOI:10.1038/srep02787 (2013).

[7] P. B. Neware, P. U. Jadhao, B. S. Singh, U. W. Kaware, and R. K. Dehankar, "Flexible Organic Light Emitting Diodes-FOLED," *Jawaharlal Darda Institute of Engineering and Technology, Yavatmal, S.G.B.A. University, India*, vol. 5, no. 5, pp. -, Oct. 2015. Accepted: Oct. 22, 2015. Available online: Oct. 26, 2015.

[8] J Department of Defense Design Criteria Standard: Human Engineering , Military Specifi cation (MIL-STD-1472F), US DEPARTMENT OF DEFENSE , 1998.

[9] Y. Li and H. Meng, Eds., Organic Light-Emitting Diodes (OLEDs): Materials, Devices and Applications. Woodhead Publishing, 2013.

[10] N. T. Kalyani, H. Swart, and S. J. Dhoble, Principles and Applications of Organic Light Emitting Diodes (OLEDs). Woodhead Publishing, 2017.

[11] Kyonghwan Oh, Seong-Kwan Hong, and Oh-Kyong Kwon, "A Luminance Compensation Method Using Optical Sensors with Optimized Memory Size for High Image Quality AMOLED Displays," J. Opt. Soc. Korea 20, 586-592 (2016)
[12] M.-K. Han, “AM backplane for AMOLED,” in *Proc. of ASID ’06*, New Delhi, India, Oct. 8–12, 2006, pp. 53–58.
[13] C.-J. Chiang, C. Winscom, and A. Monkman, “Electroluminescence characterization of FOLED devices under two types of external stresses caused by bending,” Organic Electronics, vol. 11, no. 11, pp. 201–207, 2010.
[14] G. H. Lee et al., Sci. Adv. 8, eabm3622 (2022).
[15] Y. Wang, Y. Wen, X. Zhuang, S. Liu, L. Zhang, and W. Xie, "Flexible organic optoelectronic devices: Design, fabrication, and applications," *APL Photonics*, vol. 9, no. 9, p. 090901, Sept. 2024, doi: 10.1063/5.0220555.
[16] Lee, D., Kim, SB., Kim, T. *et al*. Stretchable OLEDs based on a hidden active area for high fill factor and resolution compensation. *Nat Commun* **15**, 4349 (2024). https://doi.org/10.1038/s41467-024-48396-w