Contents

[Literature Review 2](#_Toc201759358)

[intro 4](#_Toc201759359)

[Stage 0: Image Cropping (Region of Interest Extraction) 5](#_Toc201759360)

[Stage 1: Grayscale Conversion 8](#_Toc201759361)

[Stage 2: Noise Averaging Filtering 11](#_Toc201759362)

[Stage 3: Sobel Edge Detection Technique 15](#_Toc201759363)

[Stage 4: decision module 19](#_Toc201759364)

[System Architecture Overview 24](#_Toc201759365)

[System top level block diagram 25](#_Toc201759366)

[Hardware implementation 26](#_Toc201759367)

[hardware utilization of the lane detection system 26](#_Toc201759368)

[hardware utilization percentage of the ZCU102 27](#_Toc201759369)

[Timing report 27](#_Toc201759370)

[Power Report Summary 28](#_Toc201759371)

[Throughput Calculation based on the region of interest ROI 29](#_Toc201759372)

[Comparing software approaches 30](#_Toc201759373)

[Results and Testing 32](#_Toc201759374)

[References: 42](#_Toc201759375)

# Literature Review

Regardless of hardware compatibility or computational complexity, lane detection systems have employed a wide range of algorithms aimed at accurately identifying lane boundaries under various driving conditions. Among these, the Canny edge detection and Hough Transform algorithms are often considered leading approaches due to their strong performance in structured environments. The Canny algorithm, although accurate, presents significant complexity stemming from its multi-stage processing—comprising Gaussian filtering, gradient calculation, non-maximum suppression, and double thresholding. This level of computation makes it challenging to implement efficiently on hardware platforms, particularly on FPGAs or other resource-constrained devices [3].

To mitigate these issues, researchers have explored adaptations of classical algorithms for hardware-friendliness. Malmir and Shalchian (2019) implemented the Hough Transform on an FPGA, enhancing its effectiveness by incorporating a backup filter stage for stripe detection. This solution improved lane detection reliability but came at the cost of high resource utilization, consuming over 70K Look-Up Tables (LUTs). Furthermore, the system lacked functionality for identifying the current lane index or the total number of lanes—a limitation that restricts its usability in advanced driver-assistance systems (ADAS) [4].

In response to such resource constraints, other works have focused on edge detection techniques with simpler computation, such as the Robinson, Sobel, Laplacian, and Laplacian of Gaussian (LoG) filters. Humaidi et al. (2018) selected the Sobel operator for its hardware efficiency, driven by its reduced number of convolutions and relatively low computational load. Despite its simplicity, the Sobel operator demonstrated competitive performance in identifying lane boundaries in various conditions, making it a strong candidate for real-time systems [5].

More recently, Heuijee and Daejin (2024) conducted a comparative study on the hardware deployment of the Canny and Hough algorithms across multiple platforms, including FPGAs and Hybrid Processing Systems (HPS). Their findings revealed that although both methods are theoretically robust, their practical implementation resulted in relatively low throughput, thereby limiting their suitability for high-speed, real-time applications such as autonomous driving [6].

Beyond algorithmic complexity, real-world image processing systems must contend with environmental noise that degrades image quality. Such noise often arises from the image acquisition process and can obscure lane markers or generate false edges. To address this, pre-processing techniques like filtering are applied. Among the various noise reduction methods studied, both Gaussian and averaging filters have been extensively analyzed. While Gaussian filters offer smooth noise reduction, averaging filters have shown superior performance in preserving structural details essential for lane detection, and they exhibit greater compatibility with real-time hardware implementations due to their simpler structure [5].

Building upon these findings, this project proposes a five-stage hardware lane detection architecture optimized for efficiency and reliability. The system initiates by converting incoming RGB images to grayscale to reduce data complexity. It then applies an averaging filter to minimize the impact of noise, followed by Sobel edge detection for extracting prominent lane boundaries. A single-stage thresholding operation is used for post-processing to isolate significant edges. The final stage comprises a lane identification block that analyzes the processed edge map to determine the number of lanes, identify the current lane index, and detect the boundaries of the active lane. Additionally, a valid signal is generated to interface seamlessly with any subsequent modules that rely on accurate lane information. This architecture strikes a balance between algorithmic effectiveness and hardware efficiency, making it suitable for integration into embedded automotive systems.

# intro

Lane detection plays a fundamental role in autonomous driving and advanced driver-assistance systems (ADAS), as it enables the vehicle to understand and interpret road structure in order to navigate safely and maintain lane discipline. This system is designed to operate in real-time by analyzing video frames stored in memory, providing continuous feedback about lane positioning. the system works with live data, ensuring responsiveness and adaptability to changing road conditions.

The primary objective of the lane detection system is to accurately identify the number of lanes on the road, detect the specific lane in which the vehicle is currently traveling, and determine the left and right boundaries of that lane. This information is critical for maintaining the vehicle’s trajectory and supporting lateral control strategies such as lane keeping or lane centering. Although the output of this module is passed to a downstream control system, the Speed and Steering Control Unit (SSCU) this report focuses exclusively on the detection phase, where visual information is processed and converted into structured data about the environment.

The system is architected as a four-stage image processing pipeline. The first stage involves converting RGB images to grayscale, reducing data complexity while preserving essential structural information. The second stage applies an averaging filter to smooth the grayscale image and suppress noise, which improves the robustness of subsequent edge detection. In the third stage, the Sobel filter is used to highlight horizontal and vertical edges by detecting intensity gradients. The result is a binary edge map where pixel values indicate the presence or absence of edges. Finally, the fourth stage involves a decision-making algorithm that processes the edge map to determine lane positions and boundaries based on geometric and spatial characteristics.

In the following sections, each stage of the lane detection system is discussed in detail, covering both the theoretical foundation and the practical implementation strategy used in the project. Verification methods and integration considerations are also presented to ensure the reliability and accuracy of the system under typical highway driving conditions.

# Stage 0: Image Cropping (Region of Interest Extraction)

**Purpose:**

In a lane detection system, processing the entire video frame introduces unnecessary computational load and can slow down the system’s response time. Many areas in a typical road scene, such as the sky, nearby buildings, or parts of the vehicle dashboard, do not contribute any useful information for detecting lane lines.

To improve efficiency, the Image Cropping stage isolates only the relevant portion of each frame, known as the Region of Interest (ROI). This module receives the image data pixel by pixel through an AXI-Stream interface and uses the vertical position, or row index, to determine which pixels to keep. It discards all pixels outside a predefined vertical range and forwards only the meaningful portion to the next processing stage.

This focused approach reduces the amount of data that must be handled by subsequent stages like filtering and edge detection, which improves the overall system performance. It also enhances the accuracy of lane detection by ensuring that only the relevant parts of the image are analyzed.

In summary, this stage functions as a smart filter that removes irrelevant image data. It helps maintain a balance between speed and accuracy, which is essential for reliable real-time lane detection.

Figure 1 shows the original RGB image, while Figure 2 presents the result after cropping, highlighting the region used for further analysis.

**Design and Implementation**

The image cropping module is implemented in Verilog and designed to operate in real-time using the AXI-Stream video protocol. It receives the video frame pixel by pixel, row by row, and selectively enables only the pixels belonging to the region of interest. The input to the cropping module is 416 \* 416 and the output is the ROI which is 150 \* 416.

Key aspects of the design and implementation include:

* Row Tracking Mechanism:  
  A row counter increments on the assertion of tlast, which signals the end of a row in AXI-Stream. This counter is used to determine the current row number in the incoming frame.
* Region Filtering Logic:  
  The module accepts three parameters:
  + AXI\_IMG\_HEIGHT: Total number of rows in the frame.
  + FRAME\_START: The starting row of the ROI.
  + FRAME\_HEIGHT: The number of rows to process from the starting point.

The region ends at FRAME\_END = FRAME\_START + FRAME\_HEIGHT. The core logic checks if the current row lies within [FRAME\_START, FRAME\_END). If it is true, the pixel is passed forward by asserting the output signal valid\_cnvrtr.

* AXI Compliance:  
  The module maintains full compatibility with AXI-Stream handshake signals (tvalid, tready, and tlast). It does not interfere with the pixel flow timing or introduce any buffering.
* Efficiency:  
  This design is highly efficient, using only a small counter and a comparator. It works directly on the pixel stream with no need for frame buffering or memory, enabling low-latency and low-resource operation.
* Scalability:  
  The cropping region is configurable through parameters, making the module reusable across different frame sizes or ROI needs. It can be synthesized for different image heights or resolutions without changing the core logic.

This stage acts as a real-time gatekeeper, forwarding only the portion of the frame necessary for lane detection, thus optimizing the performance of the entire pipeline.

A highway with cars on it

AI-generated content may be incorrect.

Figure 1: Original image

A highway with cars on it

AI-generated content may be incorrect.Figure 2: output from RTL stage

# Stage 1: Grayscale Conversion

**Purpose:**

The grayscale conversion stage transforms the cropped RGB image into a single-channel grayscale representation. This step simplifies the image data by reducing it from three color components (Red, Green, Blue) to one intensity value per pixel. Since lane detection is based primarily on contrast and edge patterns rather than color, grayscale processing is sufficient and significantly reduces computational complexity for subsequent stages. Figure 3 shows the input to this stage (the cropped RGB image), while Figure 4 presents the corresponding output grayscale image. This comparison clearly illustrates how the intensity-based format preserves the structural details necessary for lane detection.

**Design and Implementation:**

This stage converts each valid RGB pixel from the crop module into an 8-bit grayscale equivalent using a hardware-friendly approximation of the luminance formula. The module does not operate directly on an AXI-Stream interface. Instead, it processes pixels marked as valid by the cropping stage, ensuring that only pixels within the Region of Interest are converted.

The input is a 24-bit RGB signal received directly from the cropping module. A one-bit valid signal from the cropping module indicates when a pixel lies within the Region of Interest and should be processed. The output grayscale value is written to a FIFO using cnv\_avr\_fifo\_wr\_data when cnv\_avr\_fifo\_wr\_en is active.

Grayscale conversion logic:

The grayscale value is computed using a weighted sum of the red, green, and blue channels according to Eq.1:



To reduce hardware complexity, the formula is implemented using approximate integer weights and shift-based divisions. The red weight is 11 (approximately 0.3), the green weight is 37 (approximately 0.59), and the blue weight is 15 (approximately 0.11). Dedicated logic functions such as div\_3, div\_58, and div\_11 handle multiplication and approximate division.

Control Logic  
An internal buffer named rgb\_buffer temporarily holds incoming RGB data if the FIFO is not ready. A signal named read\_ready determines whether the module is allowed to write to the FIFO. When both the valid signal from the cropping module and read\_ready are high, the RGB value is converted to grayscale and forwarded. Saturation logic ensures the final grayscale value does not exceed the 8-bit range by checking the ninth bit of the temporary result.

Output to FIFO  
The final grayscale pixel value is assigned to cnv\_avr\_fifo\_wr\_data. The signal cnv\_avr\_fifo\_wr\_en is asserted when the module is ready to write and the FIFO can accept data. The FIFO responds with cnv\_avr\_wr\_ack to acknowledge successful write operations.

A highway with cars on it

AI-generated content may be incorrect.

Figure 3: input to the stage

Cars on a highway with white lines

AI-generated content may be incorrect.  
 Figure 4: output from RTL stage

## Stage 2: Noise Averaging Filtering

**Purpose:**

After converting the image to grayscale, small fluctuations and noise may still be present, which can disrupt accurate lane detection. To reduce these unwanted details, the next processing stage performs image smoothing using a 3×3 averaging filter.

Each output pixel is calculated by summing the values of a 3×3 window centered on that pixel, including itself and its eight immediate neighbors. The total is then divided by 9 to produce the average value. This operation reduces high-frequency noise and preserves the general structure of the image, which is important for reliable edge detection.

The module implements this filtering in real time using a combination of line buffers and shift registers to construct the 3×3 window for each pixel as the image stream arrives. As new pixels enter the window from the input stream, old pixels are shifted out, maintaining the correct neighborhood structure.

To ensure consistent processing across the entire image, the module applies zero-padding to the borders. This means the first row, last row, first column, and last column are padded with zeros to simulate a valid 3×3 window even at the edges. As a result, the filtered output has the same dimensions as the input, and no part of the image is left unprocessed.

This smoothing step ensures that only meaningful changes in pixel intensity—like those found along lane markings are preserved, while insignificant variations are suppressed, improving the quality of the subsequent edge detection.

**Design and Implementation:**

This stage applies a 3×3 averaging filter on valid grayscale pixels received from the grayscale conversion module. It smooths the image by replacing each pixel with the average of its surrounding 3×3 neighborhood. The module is designed for stream-based processing, using a combination of line buffers and a shift-register window to maintain the required context without storing the entire frame.

The design employs three internal line buffers (prev\_data\_buffer, crnt\_data\_buffer, and nxt\_data\_buffer) to store pixel rows and ensure that, at any point, a full 3×3 window is available for averaging. The module processes each pixel only when the full window is populated, and edges are implicitly handled by skipping window formation near the image borders.

The input is an 8-bit grayscale pixel stream coming from the FIFO shared between the average filter and the RGB-to-grayscale converter. Whenever the average filter is ready and the FIFO is not empty, pixel data is read into the nxt\_data\_buffer. In each cycle, every buffer shift one pixel: the crnt\_data\_buffer receives data from the nxt\_data\_buffer, and the prev\_data\_buffer receives data from the crnt\_data\_buffer. This mechanism enables the construction of a dynamic 3×3 window where each buffer represents a line in the window, and shifting ensures continuous update of pixel context for accurate neighborhood averaging.

The average output pixel is written to the next FIFO stage using avr\_sobel\_fifo\_wr\_data when avr\_sobel\_fifo\_wr\_en is asserted.

averaging Logic:

The averaging logic begins forming a valid window after at least three lines of pixels are available. Each window consists of nine 8-bit pixel values arranged as Fig.5 and applies the equation Eq.2 to determine the average:

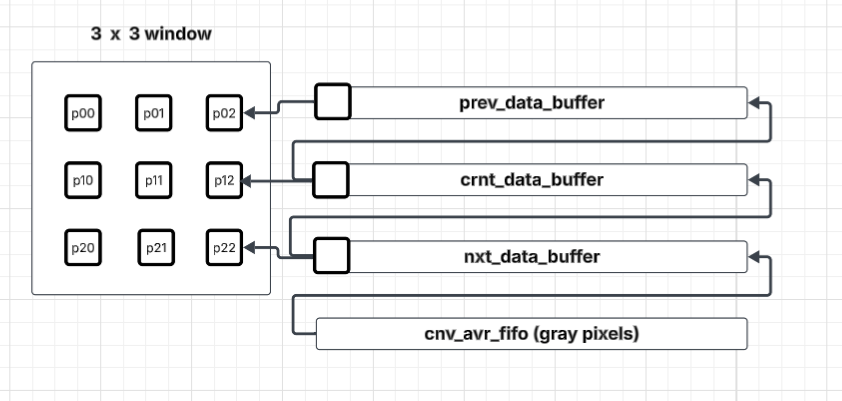


Figure 5: Proposed convolutional operation mechanism

Cars on a highway with white lines

AI-generated content may be incorrect.

Figure 6: input to the stage

  
 Figure 7: output from RTL stage

# Stage 3: Sobel Edge Detection Technique

The Sobel Filter module is designed to detect edges in the grayscale image by highlighting regions of high spatial frequency, which typically correspond to object boundaries. It processes the smoothed output of the average filter using the Sobel operator, a gradient-based method that computes the intensity change in both horizontal and vertical directions.

The goal of this module is to generate an edge map that identifies the contours and structural features in the image. This edge information is essential for subsequent stages such as object detection, segmentation, or frame composition in the image processing pipeline.

**Design and Implementation:**

The Sobel filter module computes the image gradient magnitude using two 3×3 convolution kernels: Gₓ and Gᵧ, as shown in Equations 3 and 4. These kernels are responsible for detecting horizontal and vertical edges, respectively, and are applied using a sliding 3×3 window with a stride of 1 pixel.

The module architecture is composed of two main components: line buffers (same as the average filter lane buffers) and the sliding window processor. Pixels are read sequentially from the FIFO shared between the average and Sobel modules. This FIFO acts as a temporary buffer to decouple the timing between modules. The Sobel module receives 8-bit grayscale pixel values and streams them into its internal line buffers.

The line buffers are arranged to maintain three consecutive image rows simultaneously. As new pixels enter, they shift through the buffers, enabling the construction of a dynamic 3×3-pixel window centered around the current pixel. This setup allows the Sobel operator to perform convolutions on every pixel in real-time, producing a gradient value representing the intensity of the edge at that point.

Each pixel’s gradient magnitude is computed and compared against a predefined threshold (empirically selected based on real image trials) to determine whether the pixel belongs to an edge.

A diagram of a data flow

AI-generated content may be incorrect.

Figure 8: Proposed convolutional operation mechanism

After computing the horizontal and vertical gradients (Gₓ and Gᵧ), the edge detection logic uses an internal comparator module to evaluate the gradient magnitude against two threshold values: threshold₁ and threshold₂. This dual-threshold approach allows the system to generate two distinct output images tailored for different purposes.

The comparator takes the absolute values of Gₓ and Gᵧ as inputs and internally computes a gradient magnitude approximation. Each computed value is compared against both thresholds:

Output 1 corresponds to a lower threshold (threshold₁), resulting in a more detailed edge map. This version preserves finer edges and weak gradients, making it suitable for high-fidelity visual output—such as displaying on an HDMI interface.

Output 2 applies a higher threshold (threshold₂), suppressing minor noise and retaining only strong, dominant edges. This version is optimized for applications like lane detection, where a cleaner and more robust edge representation is essential for reliability. This is used by the next stage in the lane detection system.

Each output pixel is generated based on whether the gradient magnitude exceeds the corresponding threshold, producing binary edge maps that can be directly fed into subsequent modules or displayed.

The low-detail edge map is directly routed to the decision module, which is responsible for interpreting the binary edge data to identify and track road lane markers. The Sobel filter thus serves as a critical preprocessing step, converting raw image data into structured gradient information that simplifies the task of decision-making in the context of autonomous driving or lane assistance systems.



Figure 9: input to the stage

A video game of cars driving on a highway

AI-generated content may be incorrect.  
 Figure 10: output from RTL stage

# Stage 4: decision module

**Purpose**

The Decision Module is the final stage in the image-processing pipeline, designed to interpret the processed edge image (produced by the Sobel filter) and extract meaningful information about the road environment, particularly lane detection. Its primary goal is to analyze the binary edge map, identify lane lines, and generate control signals or directional decisions based on the position and orientation of these lanes.

This module plays a crucial role in enabling autonomous navigation by:

* Detecting the presence of left, right, or centered lane boundaries.
* Determining whether the vehicle needs to steer left, right, or maintain a straight path.
* Supporting potential integration with higher-level driving logic, such as warning systems or path planning.

The decision module operates in real-time and is optimized for fast, reliable detection.

**The Algorithm**

This custom lane detection decision algorithm is designed for real-time, hardware-friendly deployment, operating in a streaming, pipelined fashion. It processes the output of the Sobel edge detector one pixel at a time, requiring no frame buffers or memory.

As each pixel exits the Sobel filter, it is immediately examined by the decision logic to determine whether it contributes to a valid lane boundary. The algorithm relies on local spatial context, such as the current pixel’s row position and the continuity of detected edge pixels over successive columns, to make decisions in real time.

Key Characteristics:

* No image storage: The algorithm works entirely in streaming mode, analyzing each pixel on-the-fly as it arrives.
* Local decision logic: Using a small window of neighboring pixels (or state from recent pixels in the same row), it identifies patterns such as:
  + Consecutive edge pixels indicating a boundary
  + Spacing between parallel edges suggesting multiple lanes
* Thresholding and width estimation: Lane edges are validated based on local pixel run lengths (e.g., expected lane width in pixels), and only continuous segments passing width/length thresholds are flagged as candidates.
* Output per pixel: For each incoming pixel, the decision logic outputs:
  + Whether it belongs to a lane edge (left/right)
  + Whether it starts or ends a valid segment
  + Whether it falls within the current lane

At the end of each row, the system resets its short-term counters or tracking flags, maintaining statelessness across rows, which ensures the design remains lightweight and FPGA-friendly.

**Why This Algorithm Excels**

This minimalist and stateless design is ideal for high-speed embedded systems where latency, area, and power consumption are critical. By avoiding memory-intensive processing or global image analysis, it achieves:

* Low-latency outputs (results available as pixels arrive)
* Consistent behavior, even in high-resolution or high-frame-rate systems
* Simplified hardware implementation, well-suited to pure RTL or HLS design without external RAM

Despite its simplicity, the algorithm is capable of accurately detecting lane boundaries and estimating the vehicle's position relative to them, especially when coupled with a well-designed Sobel and ROI filter pipeline.

**Design and implementation**

1. Sample Sobel Results:
   * Capture the first\_threshold\_sobel\_result and second\_threshold\_sobel\_result in internal registers.
   * Assert a start\_flag signal when sobel\_out\_valid is high to begin pipeline execution.
2. Column & Row Pixel Tracking:
   * Use counters to scan through image columns (column\_counter) and rows (row\_counter) synchronized with the clock.
   * Enable the row counter after completing a full image row (column\_counter == IMG\_WIDTH - 1).
3. Edge Detection (Cluster Identification):
   * Use a shift register to detect consecutive edge pixels (1's) in a row, indicating potential lane markers.
   * Mark a cluster\_detected flag when a sequence exceeds a threshold.
4. Lane Boundary Estimation:
   * When a cluster is detected:
     + Check if it's close to an existing lane boundary (within ±LANE\_SPACING).
     + If yes: Update the matched lane's position and increase its accuracy count.
     + If no: Allocate a free slot to store this new boundary.
   * When no cluster is detected, gradually decay the accuracy of previously detected boundaries.
5. Filtering and Sorting:
   * Filter out closely positioned duplicates in the boundary array.
   * Sort the filtered boundaries to prepare for final lane count analysis.
6. Lane Count Estimation:
   * Count the number of valid lane boundaries based on filtered/sorted data.
   * Determine the current lane by comparing boundary positions to the image center.
7. Output Results:
   * Output the number of detected lanes, current lane index, and its left and right boundaries.
   * Assert decision\_out\_valid when the result is ready for the next pipeline stage (e.g., speed control).

A video game of cars driving on a highway

AI-generated content may be incorrect. Figure 11: input to the stage

A video game screen shot of a road

AI-generated content may be incorrect.  
 Figure 12: output from RTL stage

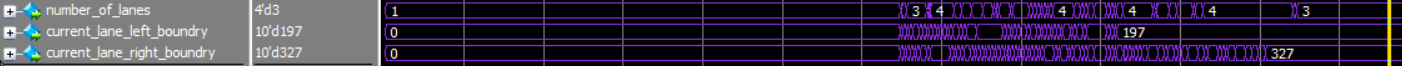


Figure 13: number of lanes and lanes boundaries

# System Architecture Overview

The architecture consists of the following sequential stages:

1. Image Crop Module  
   The pipeline begins with the Image Crop module, which selects a specific Region of Interest (ROI) from the full frame to reduce computational overhead and focus processing on the relevant part of the scene (e.g., road area). This is critical for both performance and accuracy.
2. RGB to Grayscale Conversion  
   The cropped image is then passed to the RGB-to-Grayscale Converter. This stage simplifies the data by reducing the color channels from three to one, significantly decreasing processing complexity without affecting the edge detection results.
3. Shared FIFO – Grayscale to Averaging Filter  
   A shared FIFO buffer is used to decouple the grayscale stage from the subsequent averaging filter. This allows for smooth data flow and enables pipelining between asynchronous processing blocks.
4. Averaging Filter (Noise Reduction)  
   This stage applies an averaging filter to the grayscale image to suppress high-frequency noise. The filter improves stability and clarity of edge detection, especially in outdoor conditions with variable lighting.
5. Shared FIFO – Averaging Filter to Sobel  
   Another shared FIFO separates the averaging filter from the edge detection stage. This ensures continuous data streaming and isolates the processing latencies of each module.
6. Sobel Edge Detection  
   The denoised image is then processed using the Sobel operator. This module detects edges by calculating intensity gradients in horizontal and vertical directions, highlighting lane markers effectively.
7. Decision Module  
   Finally, the Decision Module takes the output from the Sobel filter and processes it to:
   * Determine the number of visible lanes
   * Identify the current lane index
   * Detect the left and right boundaries
   * Output a valid flag to indicate successful detection and provide synchronization to downstream systems

# System top level block diagram

A diagram of a computer

AI-generated content may be incorrect.

Figure 14: top level block diagram

# Hardware implementation

# hardware utilization of the lane detection system

A screenshot of a computer

AI-generated content may be incorrect.Figure 15: hardware utilization

# hardware utilization percentage of the ZCU102

A graph with numbers and a number

AI-generated content may be incorrect.

Figure 16: hardware utilization in percentage

# Timing report

A screenshot of a computer

AI-generated content may be incorrect.

Figure 17: timing report

# Power Report Summary

| **Metric** | **Value** |
| --- | --- |
| **Total On-Chip Power** | **1.762 W** |
| **Dynamic Power** | 1.619 W (92%) |
| • PS7 (Processing System) \*\* | **1.256 W** (77%) |
| • Clocks\*\* | 0.047 W (3%) |
| • Logic\*\* | 0.134 W (8%) |
| • Signals\*\* | 0.170 W (10%) |
| **Static Power** | 0.143 W (8%) |
| **Junction Temp.** | 45.3 °C |
| **Thermal Margin** | 3.3 W |

# Throughput Calculation based on the region of interest ROI

1. System Parameters

The lane detection system operates under the following conditions:

* Clock Frequency 150 MHz
* Image ROI Resolution: 150 rows × 416 columns = 62,400 pixels per frame
* Pixel Rate: 1 pixel per clock cycle (due to full pipelining)
* Output Rate: One decision per full image frame

2. Time to Process One Frame

Since the architecture is fully pipelined, one pixel is processed every clock cycle. Therefore, the number of clock cycles required to process an entire frame is equal to the total number of pixels in the image:

Because each pixel is processed in one clock cycle, the number of clock cycles per frame is:

The time to process one frame is then given by:

Substituting values:

3. Frames Per Second (Throughput)

Final Throughput Result:

4. Remarks

This high throughput is achievable due to:

* Fully pipelined architecture, ensuring continuous processing of pixels.
* High clock frequency (150 MHz).
* Minimal per-frame logic — the decision module processes pixels in real-time without buffering the entire frame.

This performance makes the system suitable for real-time applications such as ADAS (Advanced Driver Assistance Systems) or autonomous vehicles, where both low latency and high throughput are essential.

# Comparing software approaches

| **Implementation** | **Platform** | **Clock / Specs** | **Frame Size** | **Frame Rate (fps)** | **Latency / Frame** | **Notes** |
| --- | --- | --- | --- | --- | --- | --- |
| the proposed design. | Custom pipelined FPGA | 150 MHz | 150 × 416 | ~2,400 fps | ~416 μs | Fully pipelined, pixel-per-cycle, output per frame, low latency |
| OpenCV (Canny + Hough) | CPU (e.g., Speedgoat RT) | ~2.6 GHz (quad-core) | Varying | ~10 fps | ~100 ms | Real-time system using OpenCV methods; non-pipelined |
| SwiftLane (DL method) | Embedded GPU (Jetson AGX) | NVIDIA Xavier @ 30W | 640 × 480 | ~56 fps | ~18 ms | Highly optimized deep learning-based method |
| SwiftLane (DL method) | Desktop GPU | NVIDIA RTX GPU | 640 × 480 | ~411 fps | ~2.4 ms | Requires powerful GPU; optimized batch processing |

Key Insights:

* Your FPGA architecture greatly outperforms traditional software-based techniques, achieving 2,400 fps with minimal latency (~416 μs).
* Compared to OpenCV on CPU, your system is roughly 240× faster.
* Compared to optimized deep learning models on embedded GPUs, your design is still ~40× faster in throughput.
* FPGA also has the lowest latency, which is crucial for real-time autonomous applications.
* Resource Efficiency: FPGA uses LUTs/BRAMs but within a small portion of capacity; GPU systems require heavy computing resources and high power.
* FPGA Design: Estimated nearly 2 W, highly power efficient. Low resource usage (~10%–20% of a mid-range FPGA).

# Results and Testing

To verify the functionality and performance of the lane detection hardware module, testing was carried out using a combination of RTL testbenches and MATLAB visualization. The goal was to validate the detection of lane boundaries and their correct identification across static and dynamic scenarios.

Testing Methodology

The image or video data was first processed in MATLAB, which converted the image into RGB pixel values and stored them row-wise from the top-left to the bottom-right corner. These pixel values were then written to a text file which served as the input source for the RTL simulation.

The Verilog testbench (TB) was responsible for:

* Reading the pixel data file generated by MATLAB.
* Feeding the pixel stream to the hardware module under test.
* Capturing the output of the RTL system, including:
  + The left and right lane boundaries.
  + The number of lanes detected (in video mode).
  + The output binary image resulting from the Sobel edge detection.

These output values were stored in separate files for post-processing.

MATLAB was then used again, not for capturing, but for reading the hardware-generated output files and visualizing the results. The analysis differed slightly based on the input type:

* Single-frame image:
  + MATLAB reads the left and right lane boundaries from the RTL output.
  + It overlaid green lines on the Sobel-filtered image to mark the detected lanes.
  + It computed the center of the current lane by calculating the midpoint between the two detected boundaries and drew it as an additional reference line.
* Video sequence:
  + MATLAB reads the lane boundaries frame-by-frame from the RTL output.
  + It overlaid the detected boundaries on each Sobel-filtered frame to evaluate lane tracking consistency over time.
  + This allowed visual assessment of the system's performance under real-world dynamics like curvature, occlusions, and lane transitions.

Test Dataset

* Static Images:
  + Clear road images, day-time lanes, highway curvature, multi-lane situations.
* Video Sequences:
  + Urban roads, highways with broken lanes, shadow interference, and lane merges/splits.

Output Highlights

The testing demonstrated:

* Accurate and stable detection of left and right lane boundaries.
* Robustness of the hardware in noisy and complex road conditions.
* Smooth tracking of lanes across consecutive video frames.
* Correct localization of the vehicle's current lane by computing the lane midpoint.

**Simulation Test Cases**

Test 1:

A car driving on a highway

AI-generated content may be incorrect.

Figure 22: Original image

A truck driving on a road

AI-generated content may be incorrect.

Figure 23: output from average stage

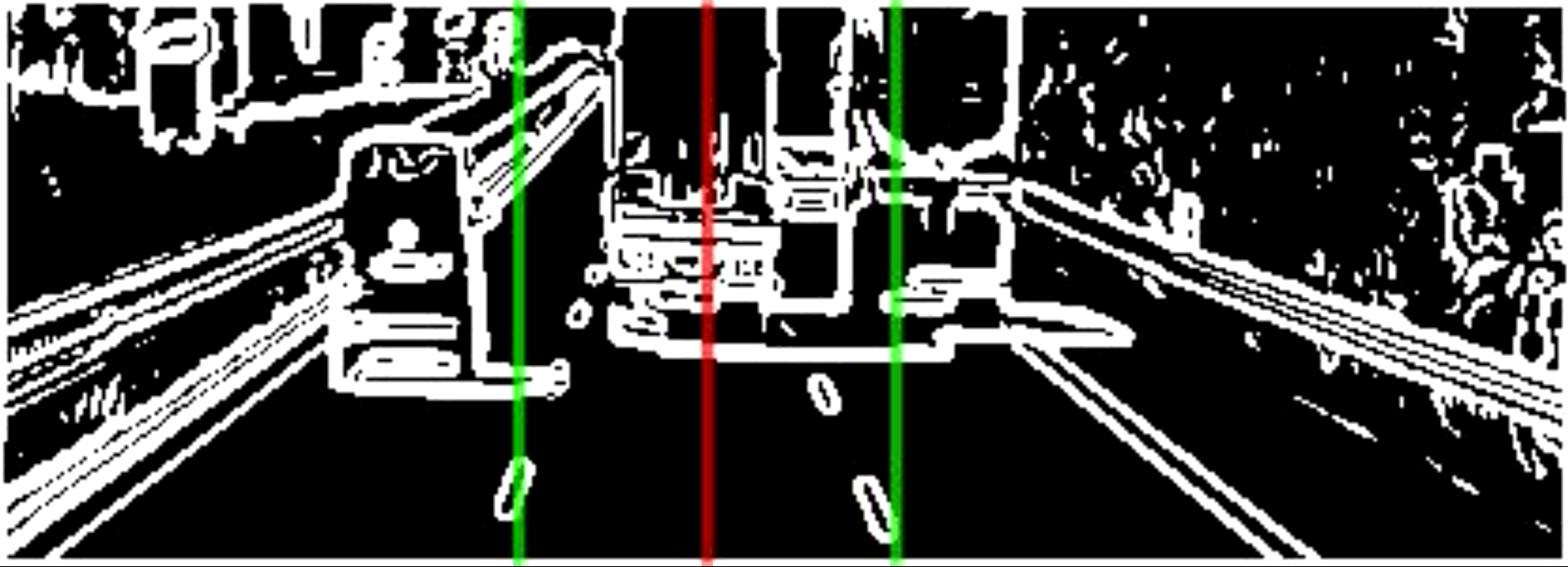


Figure 24: output of Sobel and decision boundaries

A screen shot of a computer

AI-generated content may be incorrect.

Figure 25: output of decision from simulation

Test 2:

A highway with a blue sign

AI-generated content may be incorrect.

Figure 30: Original image

A road with cars on it

AI-generated content may be incorrect.

Figure 31: output from average stage

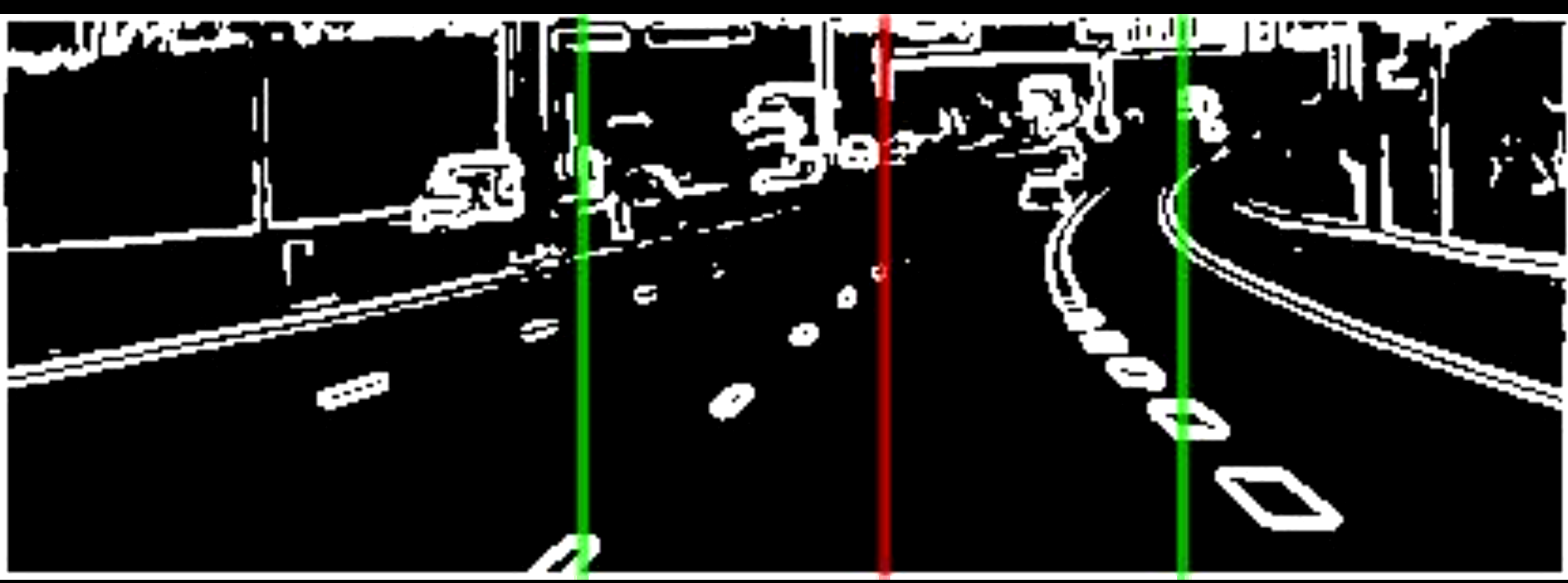


Figure 32: output of Sobel and decision boundaries

A grey and white text

AI-generated content may be incorrect.

Figure 33: output of decision from simulation

Test 3:

A highway with cars on it

AI-generated content may be incorrect.

Figure 34: Original image

Cars on a highway with a bridge over it

AI-generated content may be incorrect.

Figure 35: output from average stage

A black and white image of a room with a red and green line

AI-generated content may be incorrect.

Figure 36: output of Sobel and decision boundaries

A close up of a sign

AI-generated content may be incorrect.

Figure 37: output of decision from simulation

Comment:

In this test, the visible portion of the road was minimal. Despite this limitation, the algorithm successfully identified the current lane and its boundaries with high accuracy. However, it failed to detect the third lane.

To address this limitation, we will repeat the same test with an adjusted region of interest (ROI) to potentially improve lane detection performance.

Test 3 (changing the region of interest):

A highway with cars on it

AI-generated content may be incorrect.

Figure 38: Original image

A white line on a road

AI-generated content may be incorrect.

Figure 39: output from average stage

A black and red square with white text

AI-generated content may be incorrect.

Figure 40: output of Sobel and decision boundaries

A grey and white text

AI-generated content may be incorrect.

Figure 41: output of decision from simulation

# References:

[1] A.Broggi, M.Parent "*Intelligent Vehicles*" in Springer Handbook of Robotics, Springer, 2008

[2] D.Rufas, V.Ngo, J.Carazo, M.Codina, C.Sanchez, D.Gil, J.Carrabina "*A Survey of FPGA-Based Vision Systems for Autonomous Cars*". Survey [Online], Dec 2022. Available: https://drive.google.com/file/d/12YfB-mUJWl216LcCJ62m1yLKRmbxIB\_x/view?usp=drive\_link

[3] Educative.io *What is Canny edge detection*? [Online] Available: https://www.educative.io/answers/what-is-canny-edge-detection

[4] S.Malmir, M.Shalchian (2019) "*Design and FPGA implementation of dual-stage lane detection, based on Hough transform and localized stripe features*". Microprocessors and Microsystems [Online] Available: https://drive.google.com/drive/folders/1eBouQv95kzdembbE2lgQQjOl6Qxzgx3-

[5] A.Humaidi, S.Hasan, M. Abd-El Fadeel "*FPGA-Based Lane-Detection Architecture for autonomous vehicles: a real-time design and development*" [Online] (May, 2018)

[6] H.YUN, D.PARK (Jan, 2024) “Low-Power Lane Detection Unit With Sliding-Based Parallel Segment Detection Accelerator for FPGA”. *IEEE Xplore* [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10378646