DOCUMENTATION

Team Name:MATRIX

Event Name:ELECTRO HACK

Project Title: Intelligent Traffic Management Using FPGA and Computer Vision

Presented BY:

Y.Jaya Kushal(22BEC0263)
P.Ravi Teja(22BEC0754)
Rama Nokshith(22BEC0577)
Y.Yokeswara(22BEC0759)
M.Nabeel(22BEC0759)

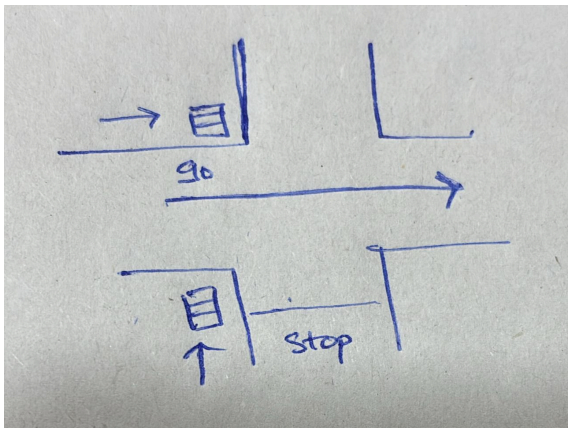
Case Study: Urbanization Challenges in Metro Cities of India

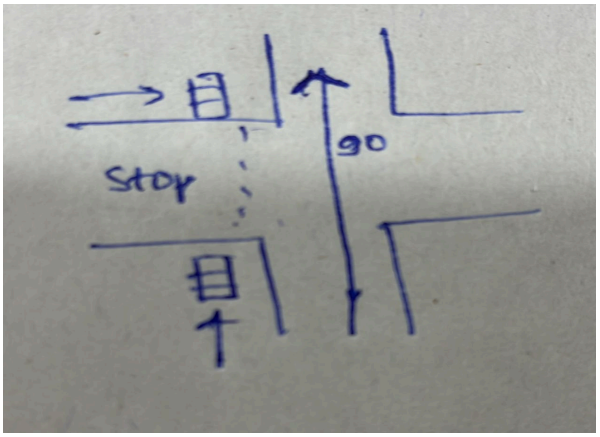
Overview India's metro cities like Delhi, Mumbai, Kolkata, Chennai, Bengaluru, Hyderabad, Ahmedabad, Pune, Surat are economic powerhouses driving the region's growth. However, rapid urbanization has brought about significant challenges, including traffic congestion, inadequate housing, poor waste management, and pressure on essential services. Addressing these issues is critical to ensuring sustainable urban development in line with SDG Goal 11: Sustainable Cities and Communities.

Key Urbanization Challenge : Traffic Congestion, Overcrowded roads lead to delays, fuel wastage, and increased pollution. Lack of efficient public transportation systems exacerbates the issue.

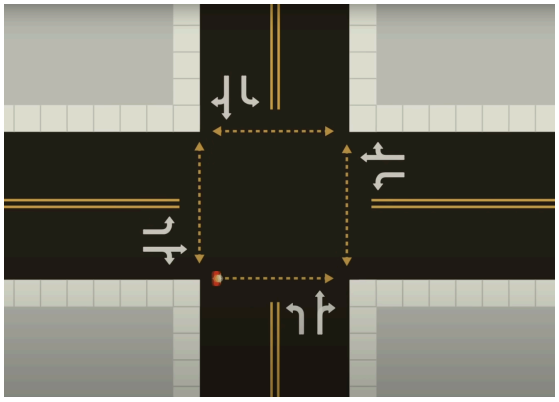
Traffic Algorithm

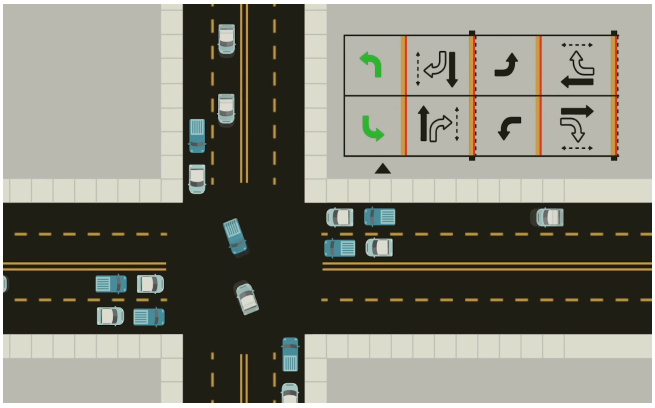
1. One lane 2 way





2. Ideal 2 lane 4 way intersection



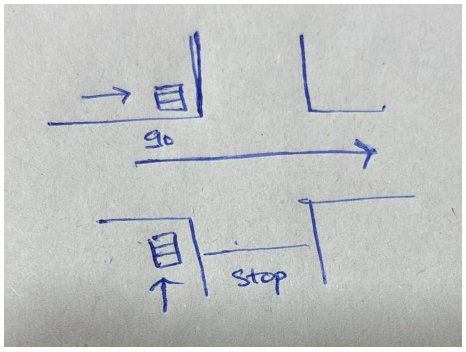


To understand how this algorithm works, refer to

How Do Traffic Signals Work?

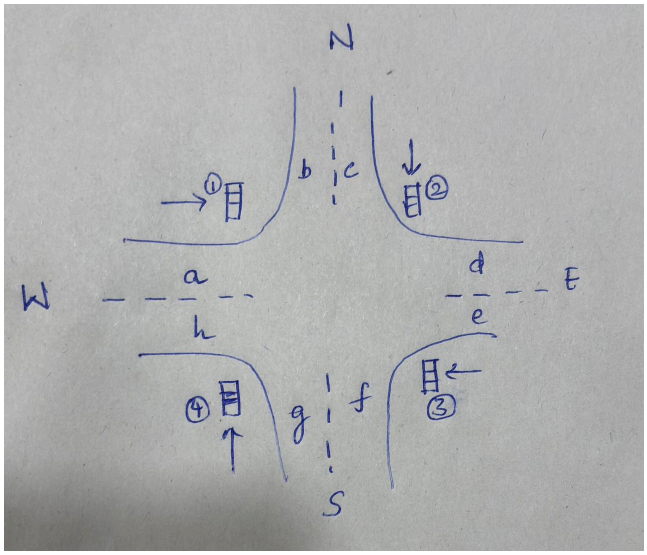
How is it Implemented in our Algorithm

One lane 2 way



- When vehicles in lane 1 are green ,lane 2 is stop,vice versa

2 lane 4 way intersection Algorithm



For the implementation of this algorithm in Verilog, we consider the following cases:

- Lane A and E when on Signal one are allowed to go straight and left ,while vehicles in Lane C and G are on signal three,i.e stop.
- Then,Lane A and E when on Signal two are allowed to go right ,while vehicles in Lane C and G are on signal three,i.e stop.
- Lane C and G when on Signal o are allowed to go straight and left ,while vehicles in Lane A and E are on signal three,i.e stop.
- Lane C and G when on Signal o are allowed to go Right ,while vehicles in Lane A and E are on signal three,i.e stop.

The above mentioned cases will be used as 4 different states in a FSM.

❖ FPGA Implementation:

❖ Introduction:

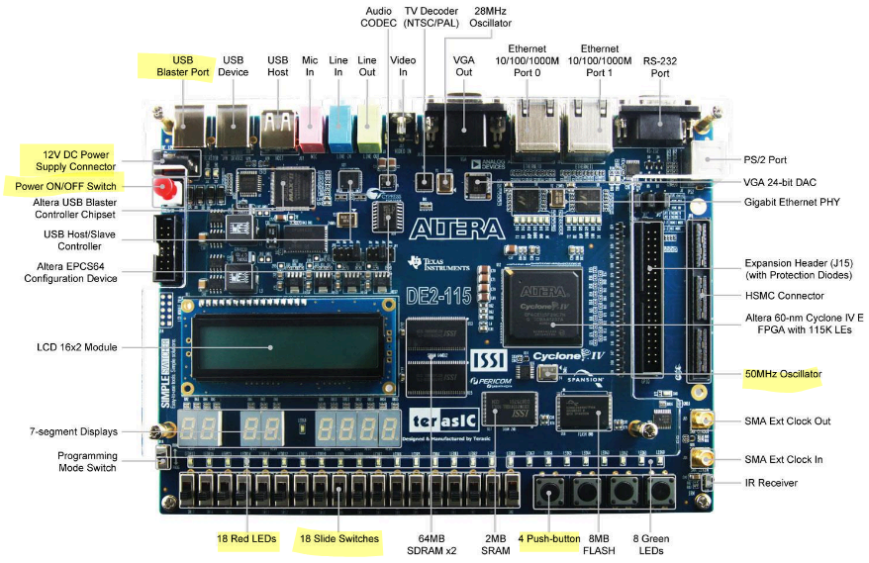
The Algorithm desired to solve the problem statement is presented in Verilog to the FPGA Board where it receives the traffic data from the traffic analysis python code to process a solution for the heavy traffic density.

Adhering to solve the Urbanization problem of huge metropolitan cities, the FPGA Board proceeds using the combination of Logics at Verilog level in real time to derive the required Timers that decide the state of the signal for the respective traffic. This is the main agenda of the FPGA implementation

Software used: AMD Vivado for Simulation and debugging
Intel Quartus Prime v23.0 for FPGA implementation
Tools Used: Altera Cyclone IV E (DE2-115), USB Blaster for FPGA Programming.
OS Used: Windows 11

❖ Cyclone IV E:

User Manual:https://www.terasic.com.tw/attachment/archive/502/DE2_115_User_manual.pdf



A Top view of the FPGA Board- Used Ports/elements are highlighted

The Cyclone IV E board manufactured by Altera is

The Cyclone IV E DE2-115, developed by Terasic Technologies, is a robust FPGA development board centered around Intel's (formerly Altera) Cyclone IV EP4CE115F29C7 FPGA. Designed for both educational and professional use, it provides

a comprehensive platform for prototyping digital systems, embedded designs, and FPGA-based applications. The board features 114,480 logic elements, 3,888 Kb of embedded memory, and 266 multipliers, offering ample resources for complex projects. Its rich suite of peripherals includes toggle switches, LEDs, 7-segment displays, VGA and audio ports, SRAM, SDRAM, SD card slots, USB, and Ethernet interfaces, enabling diverse hardware experimentation. Expansion options via GPIO headers and support for Intel's Quartus Prime software further enhance its versatility.

- ❖ **Verilog Implementation of Smart Traffic Controller for Metropolitan Cities:**
  Developed by Team Matrix | Repository: https://github.com/JayaKushal24/ElectroHack

This project implements a Smart Traffic Controller optimized for metropolitan traffic management using Verilog. The system comprises four interconnected modules, each designed to ensure efficient traffic flow synchronization. Below is the modular breakdown:

1. **Traffic Controller (Core Module):**

Functionality: Acts as the central processing unit, integrating sub-modules to execute the traffic control algorithm.

Key Features:
->Coordinates signal timing and state transitions across intersections.
->Aggregates outputs from peripheral modules (Reset, Clock Generator, State Machine) to enforce synchronized traffic flow.

2. **Reset Module:**

Purpose: Implements a delayed reset mechanism to ensure system stability during initialization.

Design Logic:
-> Introduces a configurable delay post-reset to allow transient traffic (e.g., emergency vehicles) to clear before normal operation resumes.
-> Synchronizes reset deactivation with the global clock to prevent metastability.

3. **1 Hz Clock Generator:**

Role: Derives a 1 Hz clock signal from the FPGA's onboard high-frequency oscillator.

Technical Detail:

->Utilizes a clock divider to downscale the FPGA's base clock (e.g., 50 MHz → 1 Hz).

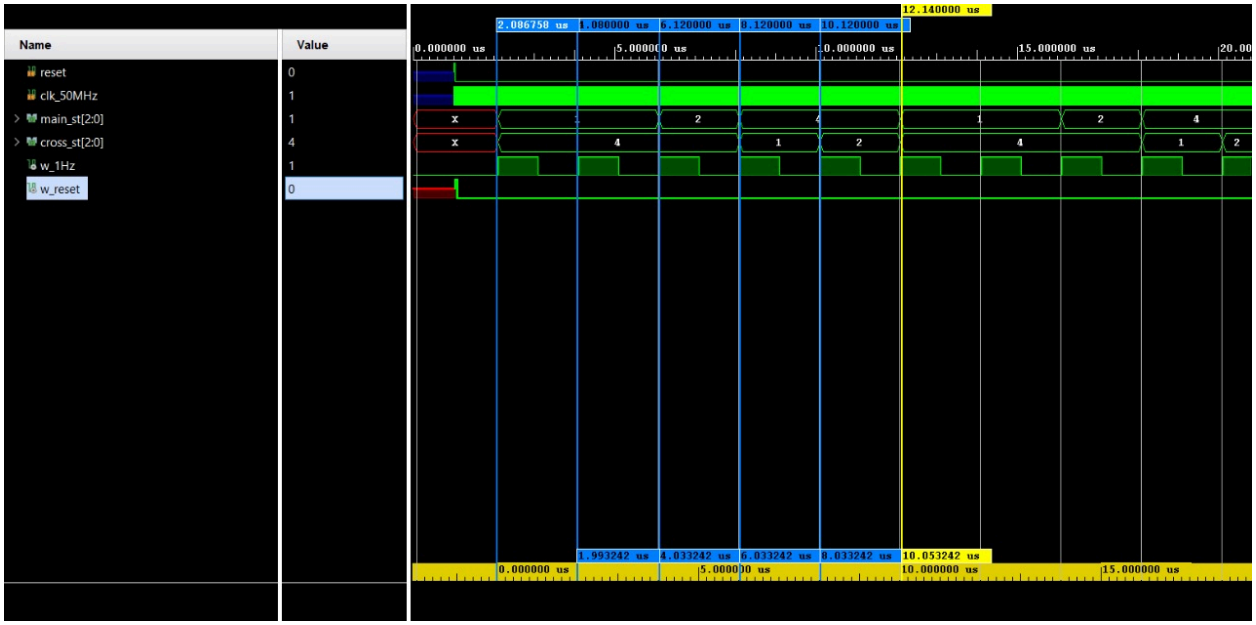
->Provides a timebase for the state machine to manage traffic light durations in real-world seconds.

4. **State Machine:**

Operation: Drives the system's behavior through predefined traffic states (e.g., Green-Yellow-Red transitions).

Key Responsibilities:
-> Dynamically adjusts counter values to set signal durations per street, prioritizing high-traffic lanes.
-> Supports configurable timing parameters for adaptive control in response to real-time traffic data.

A example of runtime in AMD Vivado:



Traffic Analysis Using YOLO and OpenCV

Problem: To develop a model for traffic analysis in python using YOLO and OpenCV

Why we choose YOLO and OpenCV:

When designing an intelligent traffic management system, the choice of tools and technologies is critical. In this case, we chose **YOLO** (You Only Look Once) for object detection and OpenCV for image processing because they offer unique advantages that are well-suited for real-time traffic monitoring and control. Below, we explain why these tools were selected and how they contribute to the overall system.

1. Why YOLO?

1.1 Real-Time Object Detection

- Key Feature: YOLO is specifically designed for real-time object detection , making it ideal for applications like traffic monitoring where quick decisions are essential.
- Advantage: Unlike traditional object detection algorithms (e.g., R-CNN or Fast R-CNN), which process images in multiple stages, YOLO processes the entire image in a single pass. This results in faster inference times while maintaining high accuracy.
- Impact on Traffic System: In traffic management, detecting vehicles, pedestrians, and other objects in real-time is crucial for dynamically adjusting traffic lights and ensuring safety. YOLO's speed ensures that the system can respond quickly to changing traffic conditions.

1.2 High Accuracy with Lightweight Models

- Key Feature: YOLO offers both full-scale models (e.g., YOLOv4, YOLOv5) for high accuracy and lightweight versions (e.g., Tiny-YOLO) for resource-constrained environments.
- Advantage: For FPGA-based systems, where computational resources are limited, lightweight versions like Tiny-YOLO can be deployed without sacrificing too much accuracy. This balance between performance and resource usage is critical for embedded systems.
- Impact on Traffic System: By using a lightweight model, we can efficiently run YOLO on an FPGA, ensuring that the system remains responsive even under heavy traffic loads.

1.3 Multi-Class Detection

- Key Feature: YOLO can detect multiple classes of objects (e.g., cars, buses, motorcycles, bicycles, pedestrians) in a single pass.

- Advantage: In traffic management, it's important to distinguish between different types of vehicles and road users. YOLO's ability to classify multiple objects simultaneously allows us to gather detailed information about traffic composition.
- Impact on Traffic System: This capability helps the system make more informed decisions, such as prioritizing buses or emergency vehicles, or adjusting pedestrian crossing times based on crowd density.

1.4 Scalability and Flexibility

- Key Feature: YOLO is highly flexible and can be fine-tuned for specific use cases (e.g., urban traffic, highway monitoring).
- Advantage: The model can be trained on custom datasets to improve its performance in specific environments, such as detecting vehicles in low-light conditions or identifying specific vehicle types (e.g., trucks, ambulances).
- Impact on Traffic System: By fine-tuning YOLO for traffic scenarios, we can improve the accuracy of vehicle detection and reduce false positives, leading to more reliable traffic control.

2. Why OpenCV?

2.1 Image Preprocessing

- Key Feature: OpenCV is a powerful library for image and video processing . It provides tools for tasks like resizing, filtering, and enhancing images before feeding them into the YOLO model.
- Advantage: Preprocessing steps like resizing images to the required input size (e.g., 416x416 for YOLO) or applying filters to reduce noise can significantly improve the accuracy of object detection.
- Impact on Traffic System: By preprocessing the video feed from traffic cameras, OpenCV ensures that the input to YOLO is clean and optimized, reducing the likelihood of errors in vehicle detection.

2.2 Video Capture and Streaming

- Key Feature: OpenCV can capture video streams from cameras and process them frame by frame in real-time.
- Advantage: OpenCV supports various camera interfaces (e.g., USB, IP cameras), making it easy to integrate with existing traffic monitoring infrastructure.
- Impact on Traffic System: OpenCV allows the system to continuously monitor traffic flow by capturing live video feeds, enabling real-time decision-making based on current traffic conditions.

2.3 Post-Processing and Visualization

- Key Feature: After YOLO detects objects, OpenCV can be used to draw bounding boxes, annotate detected objects, and visualize the results.

- Advantage: OpenCV provides simple functions to overlay bounding boxes and labels on the video feed, making it easy to visualize the detected vehicles and other objects.
- Impact on Traffic System: Visual feedback is useful for debugging and monitoring the system's performance. It also helps operators understand how the system is interpreting traffic conditions.

2.4 Integration with Other Algorithms

- Key Feature: OpenCV can be combined with other computer vision techniques (e.g., background subtraction, motion detection) to enhance the system's capabilities.
- Advantage: For example, background subtraction can help identify moving vehicles, while motion detection can be used to track vehicle trajectories.
- Impact on Traffic System: By integrating additional algorithms, OpenCV allows the system to perform more complex tasks, such as tracking vehicle movement over time or detecting anomalies like accidents or stopped vehicles.

3. Collaboration Between YOLO and OpenCV

3.1 End-to-End Pipeline

- How They Work Together: OpenCV handles the video capture and preprocessing of the input data, while YOLO performs the object detection . After detection, OpenCV is used again for post-processing and visualization .
- Advantage: This combination creates a seamless pipeline from raw video input to actionable insights (e.g., vehicle counts, traffic density).
- Impact on Traffic System: The synergy between YOLO and OpenCV ensures that the system can efficiently process video feeds, detect vehicles, and provide visual feedback, all in real-time.

3.2 Real-Time Performance

- How They Work Together: OpenCV's efficient video processing capabilities complement YOLO's fast object detection, resulting in a system that can operate in real-time.
- Advantage: Both tools are optimized for performance, allowing the system to handle high-resolution video feeds and detect objects with minimal latency.
- Impact on Traffic System: Real-time performance is critical for traffic management, where delays in detecting vehicles or adjusting traffic lights can lead to congestion or accidents.

3.3 Open-Source and Community Support

- How They Work Together: Both YOLO and OpenCV are open-source and have large, active communities. This means there are plenty of resources, tutorials, and pre-trained models available.
- Advantage: Using open-source tools reduces development time and costs, and the community support ensures that any issues can be quickly resolved.
- Impact on Traffic System: The availability of pre-trained models and community support allows for rapid prototyping and deployment of the traffic management system.

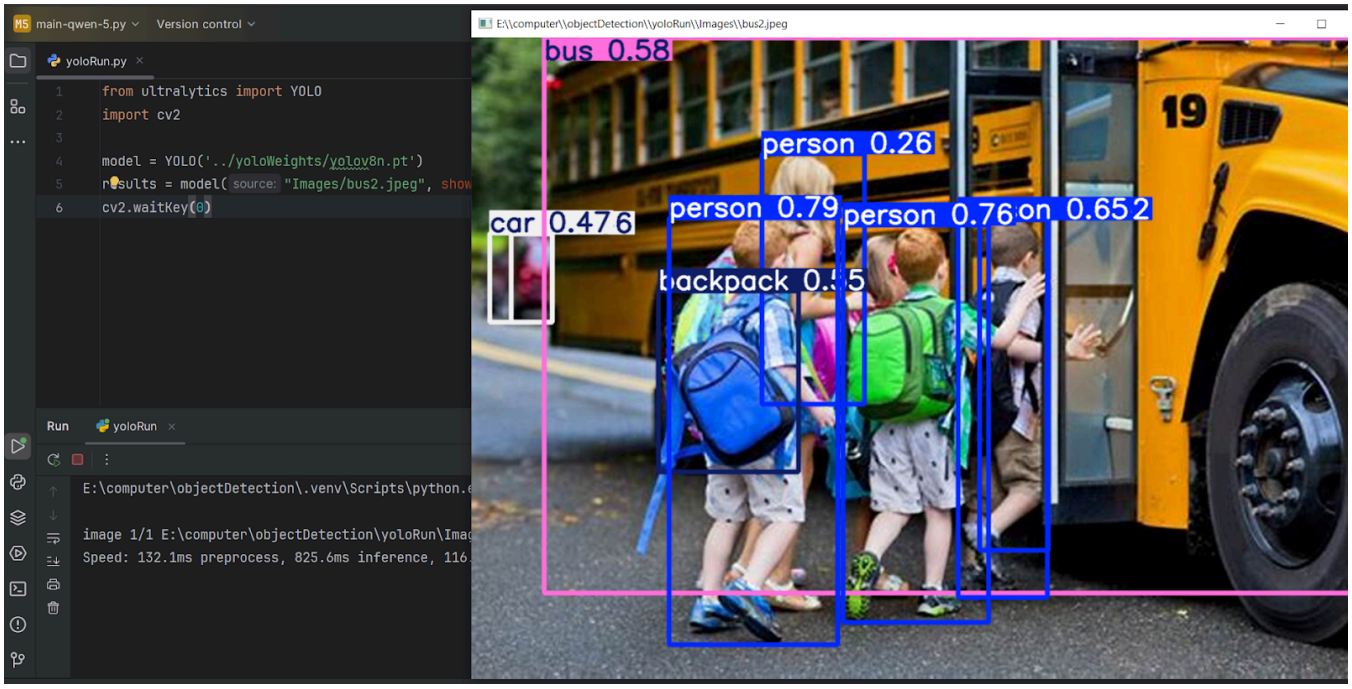
What we are doing

- **YOLO Model:**
  - Performs real-time object detection to identify and localize vehicles in video frames.
  - Classifies detected objects (e.g., cars, trucks, buses) and provides confidence scores for each detection.
  - Outputs bounding boxes around detected vehicles with color-coded labels to indicate classification and confidence levels.
- **OpenCV:**
  - Draws bounding boxes around detected objects and displays classification labels with confidence scores using distinct colors for clarity.
  - Implements masking to focus only on regions of interest (e.g., lanes or intersections), significantly reducing unnecessary computations.
  - Extracts frames at dynamic intervals (instead of processing every frame) to optimize processing time and resource usage.
  - Adjusts the frame capture interval dynamically based on traffic density, ensuring efficient processing without compromising accuracy.

Approach

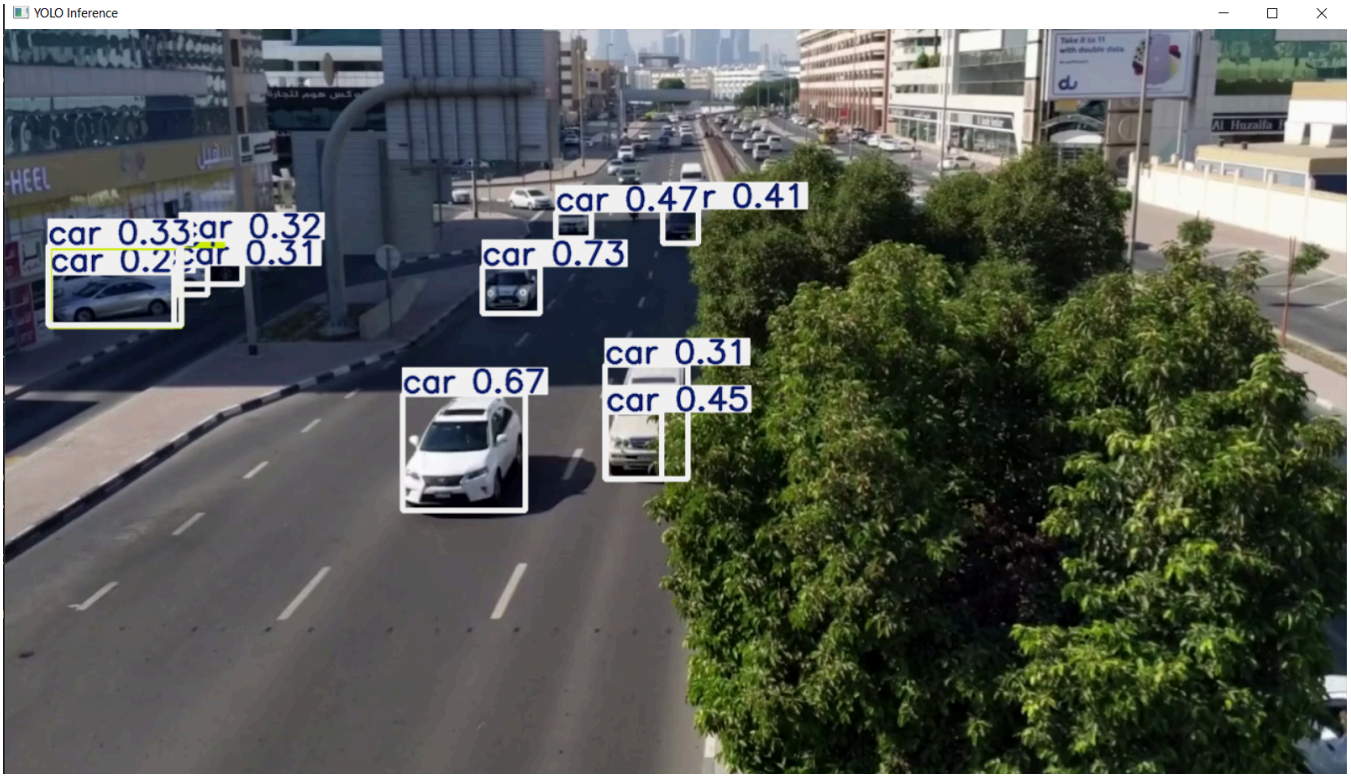
Project Approach and Development Process:

1. **Initial Setup and Model Selection:**
   - Established a Python virtual environment to streamline dependencies and ensure a consistent development environment.
   - Integrated the YOLO model for object detection, starting with the nano variant (locally stored for future scalability).
   - Conducted initial testing using static images, which demonstrated fast performance with the nano model.
   - Experimented with larger YOLO models, achieving higher classification accuracy but at the cost of increased processing time and resource consumption.
   - Decided to proceed with the nano model for prototyping due to its balance of speed and efficiency.



2. **Transition to Video Stream Processing:**
    - Advanced to real-time video stream processing by feeding frames into the YOLO nano model.
    - Encountered initial challenges with video input but resolved them through iterative debugging.
    - Identified an issue where vehicles outside the road were being detected unnecessarily. To address this:
        - Created a mask to focus only on regions of interest (e.g., lanes or intersections).
        - Used OpenCV to overlay the mask on the video stream and draw bounding boxes around detected objects in real time.

3. **Optimizing Frame Processing:**
    - Recognized that continuous frame processing was computationally expensive and unnecessary for traffic counting purposes.
    - Implemented a strategy to capture frames at fixed intervals, specifically when traffic light states changed, significantly reducing processing overhead.
    - Focused on detecting key vehicle classes (cars, motorbikes, trucks) while ensuring accurate classification and confidence scores.
    - Added debugging functionality to pause and inspect specific frames for validation purposes.



4. **Customizing Output and Dynamic Interval Adjustment:**
    - Modified the YOLO-generated output to produce a binary stream suitable for downstream processing.
    - Enhanced the system by introducing dynamic frame interval adjustment based on traffic density:
        - Assigned weights to detected vehicles, using these weights to determine optimal frame capture intervals dynamically.
    - Validated the approach with real-world scenarios, ensuring adaptability to varying traffic conditions.

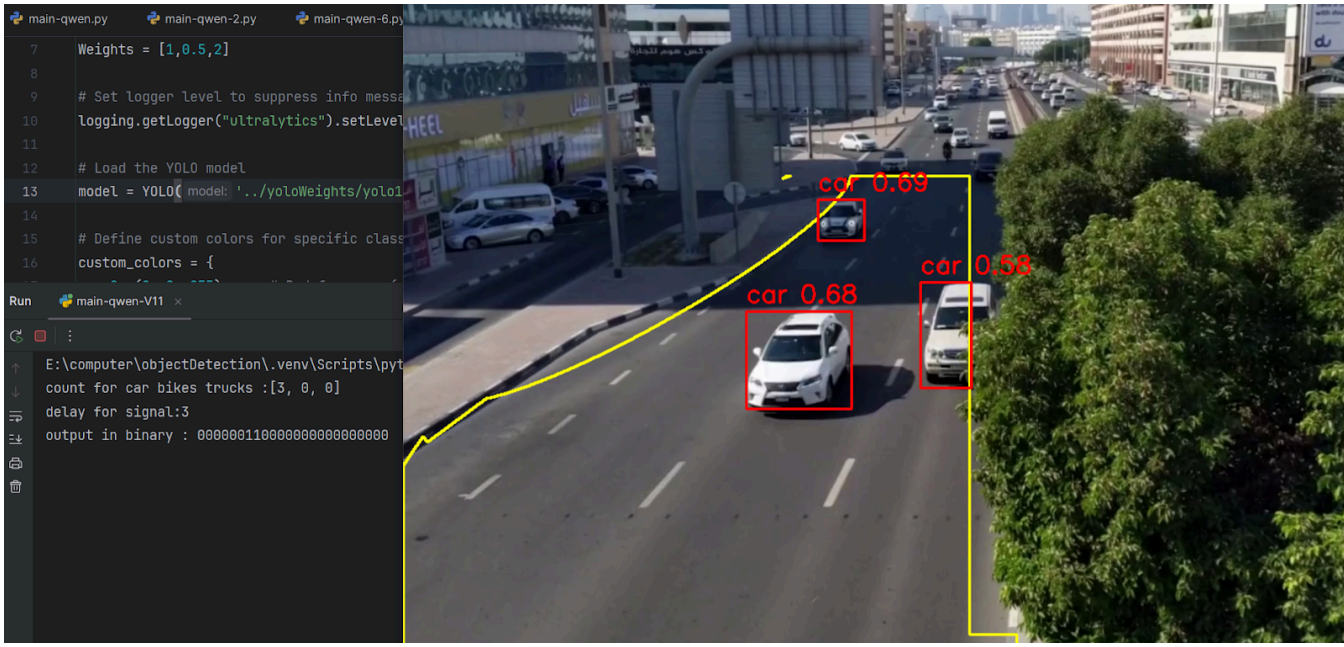
```
Speed: 2.0ms preprocess, 86.0ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)
Processing frame 240

0: 384x640 10 cars, 82.0ms
Speed: 1.0ms preprocess, 82.0ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)
Processing frame 300

0: 384x640 9 cars, 1 truck, 85.5ms
Speed: 2.0ms preprocess, 85.5ms inference, 2.0ms postprocess per image at shape (1, 3, 384, 640)
Processing frame 360

0: 384x640 1 person, 7 cars, 1 motorcycle, 1 bus, 1 truck, 87.5ms
Speed: 2.0ms preprocess, 87.5ms inference, 1.0ms postprocess per image at shape (1, 3, 384, 640)
```

5. **Performance Challenges and GPU Exploration:**
   - Tested the system using a webcam as the video source but encountered performance bottlenecks due to limited computational power, achieving only ~1 frame per second.
   - Replicated the issue with live streams from a mobile device, confirming hardware limitations as the primary constraint.
   - Researched potential solutions and identified GPU acceleration as a viable approach to enhance processing capabilities.
   - Currently in the process of integrating GPU support to improve real-time performance.



6. **Model Upgradation:**
    - Initially utilized YOLOv8 for object detection but sought further performance improvements.
    - Downloaded and integrated YOLOv11 (nano variant) to leverage advancements in accuracy and efficiency.

.

```
Weights = [1,0.5,2]

# Set logger level to suppress info messages
logging.getLogger("ultralytics").setLevel(logging.WARNING)

# Load the YOLO model
model = YOLO( model: '../yoloWeights/yolo11n.pt', verbose=False)

# Define custom colors for specific classes (BGR format)
custom_colors = {
```

"Interfacing Python with Intel Altera Cyclone IV FPGA (DE2-115): Ethernet and UART Communication Methods"

"UART-Based Communication: Transferring Python Code Output to an Intel Altera Cyclone IV FPGA (DE2-115)"

- To transfer Python code output to an Intel Altera Cyclone IV FPGA (such as the DE2-115 board) using UART (Universal Asynchronous Receiver-Transmitter), you can establish a serial communication link between the host computer running the Python script and the FPGA.
- First, ensure the FPGA is programmed with a hardware design that includes a UART interface, which can be implemented using a soft-core processor like Nios II or custom logic. On the Python side, use libraries like pySerial to configure the serial port (e.g., baud rate, parity, stop bits) and send data to the FPGA via a USB-to-UART cable connected to the FPGA's UART pins.
- The FPGA can receive the data through its UART receiver, process it using hardware logic or a soft-core processor, and generate the desired output. This method is ideal for low-speed, reliable communication and is commonly used for debugging, configuration, or control applications. Proper synchronization and error-checking mechanisms, such as parity bits or checksums, should be implemented to ensure data integrity.

Ethernet-Based Data Transfer: Sending Python Code Output to an Intel Altera Cyclone IV FPGA (DE2-115)

- To transfer Python code output to an Intel Altera Cyclone IV FPGA (such as the DE2-115 board) using an Ethernet-to-Ethernet cable, you can establish a communication protocol between the host computer running the Python code and the FPGA.
- First, ensure the FPGA is configured with a hardware design that includes an Ethernet interface, such as a Nios II processor with a lightweight IP (lwIP) stack or a custom Ethernet controller. On the Python side, use libraries like socket to create a TCP/IP or UDP connection to send data over Ethernet.
- The Python script can process data, generate the desired output, and then package it into Ethernet frames to transmit to the FPGA's IP address. On the FPGA, the received data can be processed by the hardware logic or a soft-core processor like Nios II, depending on the application.
- This setup allows real-time data transfer and interaction between the Python environment and the FPGA, enabling tasks like signal processing, control systems, or data analysis. Proper synchronization and error handling should be implemented to ensure reliable communication.

- usb serial port cable picture

Ethernet Cable