

Cross Traffic Management

Masrur Jamil Prochchhod
Department of Electronic
Engineering
Hochschule
Hamm-Lippstadt
masrur-jamil.prochchhod
@stud.hshl.de

Mehedi Hasan
Department of Electronic
Engineering
Hochschule
Hamm-Lippstadt
mehedi.hasan@stud.hshl.de

Pisula Guruge
Department of Electronic
Engineering
Hochschule
Hamm-Lippstadt
deundara-palliya-pisula-mayan.
guruge@stud.hshl.de

July 18, 2024

Abstract Efficient traffic management at intersections is crucial for reducing congestion, improving road safety, and enhancing overall traffic flow. This paper aims to design and implement an efficient traffic control system for a four-way intersection, catering to autonomous vehicles. The system employs a round-robin scheduling algorithm to manage vehicle crossing requests from different directions (North, South, East, and West). The project is developed in multiple stages: initial implementation in C, integration with FreeRTOS for real-time task management, verification using UPPAAL to ensure system correctness, and hardware design using VHDL to simulate the control logic in a digital circuit. This comprehensive approach ensures a robust traffic control system that optimizes traffic flow and enhances safety at intersections, making it suitable for real-world applications.

Introduction

One of the most serious urban problems is traffic congestion at intersections, delaying, consuming more fuel, and increasing the risks of accidents. The vehicle flow dynamics at intersections are beyond what conventional methods of traffic control—traffic lights and stop signs—are capable of managing, much more now that autonomous vehicles are coming into service. Such vehicles will require more advanced control systems so that they can navigate the intersections safely and effectively. Therefore, this project mainly focuses on developing an advanced system of traffic control at a four-way intersection. In the round-robin scheduling algorithm, the requests for four-way vehicle crossing from directions North, South, East, and West are going to be managed. Since this gives adequate opportunities for each way to cross the intersection, fairness is guaranteed, which will naturally reduce waiting periods and causes no congestion. The project is implemented in a number of steps.

First, a form of the program is realized in C, which will establish the basic logic of the system. Integration of FreeRTOS—the real-time operating system—runs different tasks and guarantees the right timely execution of tasks. Subsequently, the correctness of the system and performance will be checked using a modeling tool for verification of real-time systems called UPPAAL. Finally, the control logic is realized using VHDL into a digital circuit simulation by Modelsim, and the hardware solution can efficiently work in real-time environments. Only such an integrated approach, based on both software and hardware techniques, is needed for a strong, scalable traffic control system. In this way, the improvement of the flow of traffic will be realized along with other additional advantages of this approach, such as reducing fuel consumption and enhancing safety at the intersection, so it could present a viable solution for modern urban traffic management. [1]

SysML Diagrams

SysML (Systems Modeling Language) diagrams provide a visual representation of the system's structure and behavior. They help in understanding the flow of information and control within the traffic control system. In this project, we use three key SysML diagrams: Use Case Diagram, Activity Diagram, and Sequence Diagram.

Requirement Diagrams

The diagram shows a centralized traffic management system in charge of the flow of autonomous vehicles at intersections. At the core of this system, there is an ICU, which does the best to make the vehicles move both smoothly and safely. Detection of a vehicle by the Vehicle Detection identifies approaching autonomous vehicles, capturing their position, direction, and speed. Such

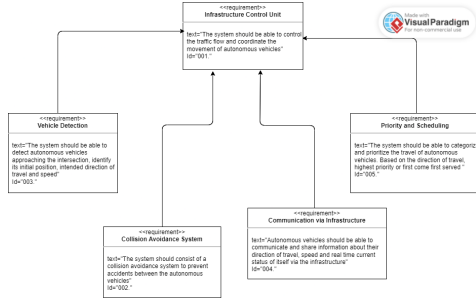


Figure 1: Requirement Diagram

data aids the ICU in understanding the condition of the traffic flow and making inferences. The Collision Avoidance System detects potential collisions and alerts the ICU to adjust the traffic lights and vehicle directions to ensure safety.

Another critical component is Communication via Infrastructure, wherein the self-driven vehicles would share real-time data about their direction, speed, and status, thus keeping the ICU updated for efficient management of traffic.

Activity Diagram

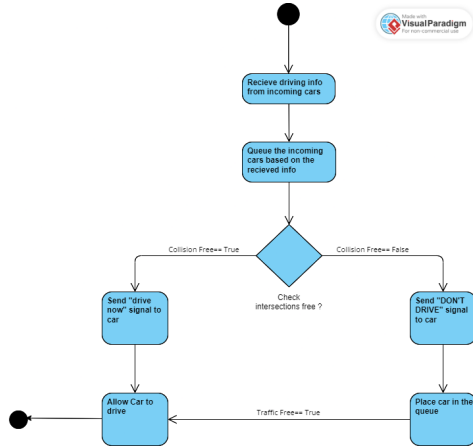


Figure 2: Activity Diagram

The activity diagram shows the autonomous vehicle control in the intersection by the centralized traffic management system. It starts by gathering the driving information of every arriving car, including position, speed, and intended direction of travel. This data will help in determining a clear picture of the status of the traffic scenario.

It then queues the cars according to this information, putting them in line to the systematic management of the intersection. It reads out the status at the intersection to make sure that it is free from any other collisions, a very critical step in safety matters as it avoids accidents, thereby allowing smooth flow in traffic.

If the intersection is clear of any collision, it will send a "drive now" signal back to the lead car in the queue, allowing it to pass the intersection. In case of a probable collision, it will send back a "DON'T DRIVE" with a message to wait, putting the car back in the queue for periodic reconsideration in the next cycle.

Use Case Diagram

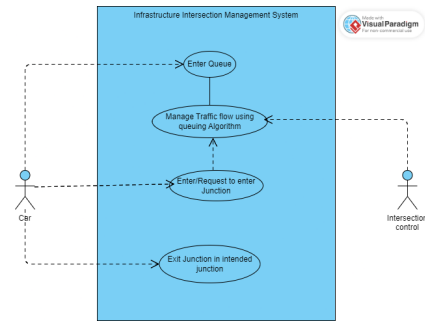


Figure 3: Use Case Diagram

The process initiates when a vehicle approaches the system. The vehicle first enters the "Enter Queue" state, whereby it enters a holding queue of vehicles trying to drive through an intersection.

Subsequently, the system integrates a queuing algorithm for traffic control. This algorithm, in a systematic way, positions each vehicle in the queue, figuring out the best way by which they ought to proceed to pass through the intersection and thus ensuring smooth and effective management of traffic.

When a vehicle reaches the front of the queue, it changes to the state "Enter/Request to enter Junction." At this stage, the vehicle requests permission to enter the intersection.

After getting permission from the intersection control, the vehicle afterwards enters the intersection and exits it on the intended path. This is represented by the state "Exit Junction in intended junction."

State Machine Diagram

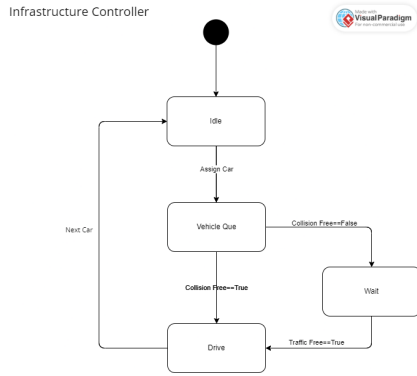


Figure 4: State Machine Diagram

In the "Vehicle Queue" state, the system checks for potential collisions. If the route is collision-free, the car proceeds to the "Drive" state. If a potential collision is detected, the system transitions to a "Wait" state.

In the "Wait" state, the system monitors the traffic conditions. Once the traffic is clear (traffic-free), the system allows the car to move to the "Drive" state. After the car has driven, the system is ready to handle the next car, looping back to the "Idle" state, and the process repeats. This sequence ensures a safe and efficient flow of vehicles through the infrastructure, handling each car one at a time while avoiding collisions and managing traffic conditions.

Sequence Diagram

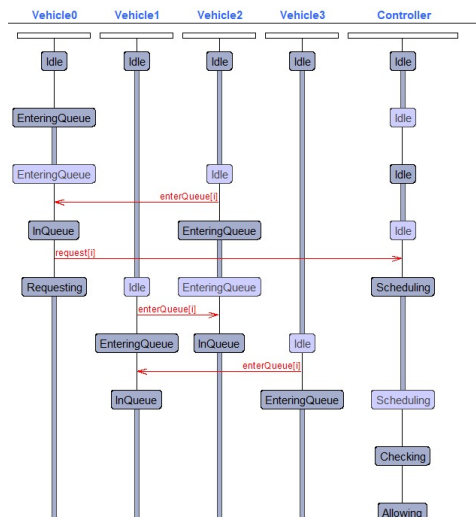


Figure 5: Sequence Diagram

This sequence diagram depicts the interactions between multiple vehicles (Vehicle0, Vehicle1, Vehicle2, Vehicle3) and the Controller within an "Infras-

tructure Intersection Management System." Initially, all vehicles and the Controller are in an idle state. As each vehicle approaches the intersection, it transitions from idle to entering the queue and then to in queue, sending a request to the Controller to enter the intersection. The Controller, upon receiving these requests, moves from idle to scheduling, managing the requests using a scheduling process. Each vehicle waits in the requesting state while the Controller checks the intersection. Once verified, the Controller authorizes the vehicles to proceed, ensuring an organized and efficient flow of traffic through the intersection. This system maintains order by managing vehicle requests and controlling their entry based on real-time traffic conditions and queue management.

Queueing Model

Our traffic control system maintains a different queue for each direction: North, South, East, and West. This will ensure that vehicles are attended to in the sequence of their arrival to guarantee fairness and order. Now, in this case, every vehicle reaching the junction joins a queue for its direction. Next, using a round-robin scheduling algorithm, these queues are processed one at a time, allowing one vehicle at a time to cross over the intersection.

Scheduling Algorithm

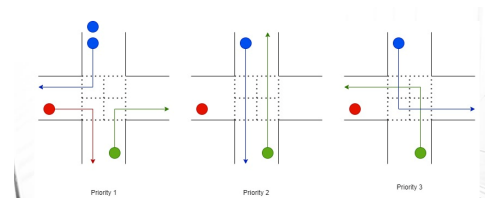


Figure 6: Scheduling Algorithm

In our centralized traffic management system, we have implemented a round-robin scheduling algorithm to efficiently manage autonomous vehicles at junctions. The intersection is divided into a number of lanes going towards different directions: northbound, southbound, eastbound, and westbound. Each lane obtains a fixed time slot, and the system cycles through all these lanes round-robin to guarantee every direction has an equal opportunity to proceed.

Parallelism is increased by allowing non-interfering movements to proceed simultaneously. For instance, the southbound left-turn lane may advance during the actuation of the northbound lane, provided that their paths never cross each other. This

increases the overall throughput at an intersection. [2]

The system continuously monitors traffic conditions using sensors and real-time data from vehicles. If a particular lane has higher traffic volume, the system dynamically adjusts the time slots, giving more time to that lane in the next cycle to reduce congestion.

Implementation in C

```
1 #include <stdio.h> // Include standard input-output header
2 #include <stdlib.h> // Include standard library header for functions like rand() and srand()
3 #include <time.h> // Include time header for time() function
4
5 #define MAX_VEHICLES 10 // Maximum number of vehicles in a queue
6 #define NUM_DIRECTIONS 4 // Number of directions (N, S, E, W)
7
8 typedef struct {
9     int id; // Vehicle ID
10    int speed; // Speed of the vehicle (1-3 units/time unit)
11    char direction; // Direction of the vehicle (N, S, E, W)
12 } Vehicle; // Define a structure for Vehicle
13
14 typedef struct {
15     Vehicle vehicles[MAX_VEHICLES]; // Array to store vehicles
16     int count; // Number of vehicles in the queue
17 } VehicleQueue; // Define a structure for VehicleQueue
18
19 VehicleQueue northQueue, southQueue, eastQueue, westQueue; // Define queues for each direction
20 char possibleDirections[] = {'N', 'S', 'E', 'W'}; // Array of possible directions
```

Figure 7: Structures

Achieving Parallelism

Our autonomous vehicle traffic control system makes use of the Round Robin real-time scheduling algorithm implemented using FreeRTOS. The bottom line in parallelism is to have all tasks managed simultaneously and not all tasks waiting for one to complete before the other starts. Each arriving vehicle at the junction is handled by its own task, and such tasks work independently. These tasks are responsible for implementing vehicle arrival and queuing tasks from the four different directions: North, South, East, and West. Using FreeRTOS, all of these tasks can concurrently run to exploit the real-time operating system in multitasking. Parallel execution of different tasks ensures that vehicles in different directions get enqueued without any delay.

The implementation for the control task uses the Round Robin scheduling algorithm, which processes every queue in cyclic order. The task will check in each queue for vehicles and let all of them that come from directions that do not conflict cross the intersection simultaneously. For instance, vehicles from the North and South queues can cross simultaneously, as can vehicles from the East and West queues. It ensures that parallel processing of several vehicles is done, paralleling the processes in execution to ensure efficiency and reduce waiting time. For smooth, efficient management of the intersection, the FreeRTOS scheduler shares the available CPU cycles fairly amongst all tasks. In this way, parallelism in handling multi-queues and allowing simultaneous crossing optimizes the flow of the traffic and improves the performance of the traffic control system.

The traffic control system is implemented in C with the definition of the structure for vehicles and queue, functions to add and remove vehicle in queues, and all the logic implementing the round-robin scheduling algorithm. We will present it initially as the basis for extensions as integral units. First, we will define the data structures and constants needed. We include standard input/output and standard utility libraries. Here, MAXVEHICLES is a constant that sets the maximum number of vehicles that can be in a queue, while NUMQUEUES refers to the number of four cardinal directions: North, South, East, West. We further define a Vehicle structure for the representation of every vehicle with its unique ID, speed, and direction. Struct Queue is a 4-way queue used to manage the vehicles that approach from each direction.

```
34 void addVehicleToQueue(VehicleQueue *queue, Vehicle vehicle) {
35     if (!isQueueFull(queue)) { // Check if the queue is not full
36         queue->vehicles[queue->count++] = vehicle; // Add vehicle to the queue and increment the count
37     } else {
38         printf("Queue is full, cannot add vehicle ID %d\n", vehicle.id); // Print message if queue is full
39     }
40 }
41
42 Vehicle removeVehicleFromQueue(VehicleQueue *queue) {
43     Vehicle emptyVehicle = {-1, 0, ' '}; // Define an empty vehicle to return if queue is empty
44     if (!isQueueEmpty(queue)) { // Check if the queue is not empty
45         Vehicle vehicle = queue->vehicles[0]; // Get the first vehicle in the queue
46         for (int i = 0; i < queue->count - 1; i++) {
47             queue->vehicles[i] = queue->vehicles[i + 1]; // Shift all vehicles to the left
48         }
49         queue->count--; // Decrement the count of vehicles
50         return vehicle; // Return the removed vehicle
51     }
52 }
```

Figure 8: Adds Vehicle

The addvehicletqueue function adds a vehicle to the back of the queue if it is not full. The back index is updated circularly. The removeVehiclefromqueue function removes a vehicle from the front of the queue if it is not empty. The front index is updated circularly. If the queue is empty, it returns an "empty" vehicle with ID -1.

```

void simulateTraffic(int totalVehicles) {
    int vehicleCount = 0; // Initialize the vehicle count
    while (vehicleCount < totalVehicles) { // Loop until all vehicles are created
        int direction = rand() % 4; // Randomly choose a direction: 0 = North, 1 = South, 2 = East, 3 = West
        switch (direction) {
            case 0:
                createVehicle(northQueue, vehicleCount + 1); // Create a vehicle in the north queue
                break;
            case 1:
                createVehicle(southQueue, vehicleCount + 1); // Create a vehicle in the south queue
                break;
            case 2:
                createVehicle(eastQueue, vehicleCount + 1); // Create a vehicle in the east queue
                break;
            case 3:
                createVehicle(westQueue, vehicleCount + 1); // Create a vehicle in the west queue
                break;
        }
        vehicleCount++; // Increment the vehicle count
    }
}

```

Figure 9: Traffic Simulation

The simulateTraffic function creates a specified number of vehicles and distributes them randomly among four direction queues: North, South, East, and West. It initializes a counter, vehicleCount, to zero and uses a while loop to keep creating vehicles until the total specified number (totalVehicles) is reached. Inside the loop, the function generates a random number between 0 and 3 to determine the direction of each vehicle, with 0 representing North, 1 representing South, 2 representing East, and 3 representing West. A switch statement then adds the vehicle to the corresponding queue by calling the createVehicle function, which assigns a unique ID, random speed, and direction to the vehicle. [4] After adding the vehicle to the appropriate queue, the counter is incremented, and the process repeats until all vehicles are created and distributed. This approach simulates the random arrival of vehicles from different directions, creating a varied and realistic traffic flow in the simulation

FreeRTOS

We integrate our traffic control system with FreeRTOS, which is a real-time operating system to increase the realism of our traffic control system and process it in real-time. FreeRTOS handles tasks such as vehicle spawning and round-robin scheduling for the timely and concurrent execution of tasks. FreeRTOS helps create multiple tasks and manages them, making sure each one runs efficiently. In this project, FreeRTOS will be used for task management of vehicle spawning and Round-robin scheduling of vehicle crossing, and for synchronization between tasks using semaphores and mutexes.

```

// sketch_ju30n.ino
1 #include <Arduino.h>
2 #include "freertos/freertos.h"
3 #include "freertos/task.h"
4 #include "freertos/queue.h"
5 #include "freertos/semphr.h"
6
7 // Define the Vehicle structure
8 struct Vehicle {
9     int id;
10    int speed; // 1-3 units/time unit
11    char direction; // N, S, E, W
12 };

```

Output Serial Monitor <x

Message (Enter to send message to 'ESP32S3 Dev Module' on 'COM7')

Vehicle ID 3 from North queue is crossing the intersection with speed 2
Vehicle ID 4 from South queue is crossing the intersection with speed 1
Vehicle ID 1 from East queue is crossing the intersection with speed 2
Vehicle ID 12 from North queue is crossing the intersection with speed 1
Vehicle ID 14 from South queue is crossing the intersection with speed 1
Vehicle ID 2 from East queue is crossing the intersection with speed 1
Vehicle ID 8 from North queue is crossing the intersection with speed 1
Vehicle ID 6 from South queue is crossing the intersection with speed 3
Vehicle ID 13 from East queue is crossing the intersection with speed 3
Vehicle ID 5 from North queue is crossing the intersection with speed 1
Vehicle ID 9 from East queue is crossing the intersection with speed 2
Vehicle ID 15 from North queue is crossing the intersection with speed 2
Vehicle ID 16 from East queue is crossing the intersection with speed 2
Vehicle ID 17 from North queue is crossing the intersection with speed 3
Vehicle ID 10 from East queue is crossing the intersection with speed 1
Vehicle ID 19 from North queue is crossing the intersection with speed 1
Vehicle ID 18 from East queue is crossing the intersection with speed 2
Vehicle ID 11 from East queue is crossing the intersection with speed 3
Vehicle ID 7 from East queue is crossing the intersection with speed 2
Vehicle ID 20 from East queue is crossing the intersection with speed 3

Figure 10: FreeRTOS Implementation

we used ESP32 for implementing FreeRTOS in arduino. It has a Vehicle structure with an ID, a speed, and a direction. It also has four queues representing the four directions: North, South, East, West, that are initialized to hold vehicles. The vehicles are spawned with random attributes and assigned to one of the direction queues. when the cars cross the intersection while details get printed on the serial monitor. For a program, a semaphore is needed to ensure that each queue for vehicles and let all of them that come from directions that do not conflict cross the intersection simultaneously. The setup function initializes the serial communication, random seed, queues, semaphore, and creates tasks for vehicle handling and traffic control. The loop function does nothing because all FreeRTOS tasks handle execution of a program.

1 Uppaal Implementation

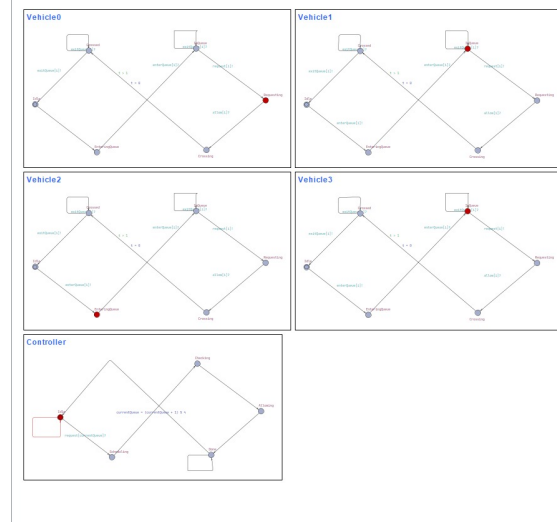


Figure 11: Uppaal Implementation

This image provides state machine diagrams for multiple autonomous vehicles (Vehicle0, Vehicle1, Vehicle2, Vehicle3) and a central controller involved

in the traffic management system. All vehicles enter the `idleStopped` state, specifying that they are not moving. Getting the signal from the central controller, they move to the `enterSignal` state, which is further followed by a transition into `Requesting` state to request permission to cross the intersection.

While in the `Safe?` state, each car checks if it is safe to proceed, and in particular, not to have any possible collisions. Assuming the intersection is clear, a vehicle transitions to the `Crossing` state to move through the intersection. Upon crossing, the vehicle enters the `exitQueue` state before going back to `idleStopped` for the next cycle.

The central controller is never in an idle state. It changes to the `Requesting` state upon receipt of a request from a vehicle for checking the current conditions in `Checking` state for safety. If the conditions are safe, the controller shall move into `Safe` state, permission to be given to the vehicle to cross, attending the status of intersection accordingly in `Attending` state.

It consists of a request from a vehicle to cross, followed by the controller checking the safety, granting permission, and then proceeding with the vehicles through the intersection in a round-robin continuous cycle. The structured approach ensures the efficient management of the flow of traffic because the safe and efficient movement of vehicles through the intersection A is implemented with a round-robin scheduling algorithm.

VHDL and ModelSim

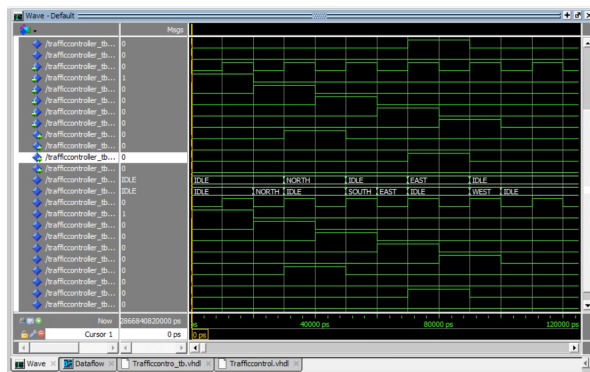


Figure 12: ModelSim

The image 13 shows a waveform simulation for a traffic controller in VHDL. It depicts the state transitions of the traffic signals managed by the controller. The signals transition between different states such as `IDLE`, `NORTH`, `SOUTH`, `EAST`, and `WEST`, reflecting the round-robin scheduling algorithm's operation. Each state indicates the active direction of traffic flow, ensuring each direction gets

an equal turn. The simulation confirms that the controller correctly cycles through the traffic signals as intended.

```

21 architecture Behavioral of TrafficController is
22     type state_type is (IDLE, NORTH, SOUTH, EAST, WEST);
23     signal state : state_type := IDLE;
24     signal next_state : state_type;
25
26 begin
27
28     process (clk, reset)
29     begin
30         if reset = '1' then
31             state <= IDLE;
32         elsif rising_edge(clk) then
33             state <= next_state;
34         end if;
35     end process;
36

```

Figure 13: VHDL Code

```

when SOUTH =>
    allow_north <= '0';
    allow_south <= '1';
    allow_east <= '0';
    allow_west <= '0';
    next_state <= IDLE;

```

Figure 14: For Directions

The provided VHDL code describes a traffic controller using a finite state machine (FSM) that manages traffic from four directions: North, South, East, and West. The entity *TrafficController* defines inputs and outputs for the system, including a clock (`clk`), a reset signal (`reset`), request signals from each direction (`request_north`, `request_south`, `request_east`, `request_west`), and allow signals to permit traffic flow from each direction (`allow_north`, `allow_south`, `allow_east`, `allow_west`). The architecture of the traffic controller contains an FSM with states representing idle (`IDLE`) and each direction (`NORTH`, `SOUTH`, `EAST`, `WEST`). The current state of the FSM is stored in the signal `state`, which is initialized to `IDLE`, and the next state is determined by the signal `next_state`.

A process driven by the clock and reset signals updates the state to the `next_state` on the rising edge of the clock or resets the state to `IDLE` when the reset signal is active. Another process determines the `next_state` based on the current state and the request signals. When in the `IDLE` state, if a request signal from any direction is detected (`request_north`, `request_south`, `request_east`, or `request_west`), the FSM transitions to the corresponding state (`NORTH`, `SOUTH`, `EAST`, `WEST`). In each direction state, the corresponding allow signal (`allow_north`, `allow_south`, `allow_east`, `allow_west`) is set to '1', permitting traffic from that direction while the others remain '0'. After granting permission, the FSM returns to the `IDLE` state. If no request signals are detected, the FSM remains in the `IDLE` state. This design ensures that only one direction is allowed to move at a time, preventing collisions at the intersection. [3]

Evaluation and Conclusion

This project is on the implementation of a centralized autonomous vehicle traffic management system using the round-robin scheduling algorithm. The system efficiently controls the flow of traffic at an intersection in four directions: North, South, East, and West. A designed FSM will control the flow of traffic through a VHDL-based traffic controller to ensure that every direction gets its share of the green light.

The round-robin algorithm reduces the wait time and prevents one single line from monopolizing the intersection, as each turn gets a chance to proceed. Then, the FSM avoids collisions and increases the safety aspect by only allowing one flow of traffic at a time. The system does have scalability in mind; it can be increased and manage more directions or lanes if needed.

In general, this VHDL-based FSM traffic controller is an effective solution to handle intersection traffic, much needed. The implemented round-robin scheduling algorithm optimizes the movement opportunities given to all directions through a balanced and fair traffic flow. This will hence enhance the efficiency and safety of intersections in the view of autonomous vehicles.

References

- [1] Ian F Akyildiz and Ismail H Kasimoglu. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks*, 2(4):351–367, 2004.
- [2] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing*. Wiley-Interscience, 2005.
- [3] Volnei A Pedroni. *Circuit Design with VHDL*. MIT Press, 2004.
- [4] Andrew S Tanenbaum and David J Wetherall. *Computer Networks*. Pearson, 5 edition, 2011.

AFFIDAVIT

We hereby confirm that we have written this paper independently and have not used any sources or aids other than those indicated. All statements taken from other sources in wording or sense are clearly marked. Furthermore, we assure that this paper has not been part of a course or examination in the same or a similar version.




Masrur Jamil Prochchhod

Lippstadt, 18.07.2024



Mehedi Hasan

Lippstadt, 18.07.2024



Pisula Guruge

Lippstadt, 18.07.2024