



**SİVAS
BİLİM VE TEKNOLOJİ
ÜNİVERSİTESİ**

PROJECT REPORT OF
TRAFIC LIGHT CONTROLLER WITH SENSOR

DEPARTMENT OF COMPUTER ENGINEERING

Student's Name:

Mehmet Yusuf OCAK

github.com/YusufOck

Student's Name:

Mehmet Yasin UZUN

github.com/mehmetyasinuzun

Student's Number:

220201015

Student's Number:

220201023

Module Courses:

➤ Dijital System Disagn
Asst. Prof. Dr. Nurbanu Güzey

➤ Microprocessors
Asst. Prof. Dr. Recep EMİR

Context:

Context:	2
PREFACE	4
DIJITAL SYSTEM DISIGN	5
1.INTRODUCTION	5
2.LITERATURE REVIEW	6
1. Flip-Flops (D Tipi Flip-Flop)	6
2. 555 Timer	6
3. Logic Gates	6
4. Finite State Machine (FSM)	7
5. Sensor Integration	7
6. Proteus Simulation and PCB Design	7
3.SYSTEM DESIGN	8
1.Circuit Overview	8
2. Fundamental Component: D-Type Flip-Flop	9
3.Flip-Flop Utilization and FSM Design in the Traffic Light Controller:	10
4.PHYSICAL IMPLEMENTATION: BREADBOARD AND PCB	15
2.BREADBOARD IMPLEMENTATION	15
1.PCB PART	15
3.PRODUCTION STAGE OF PCB	16
4.PCB IMPLEMENTATION	17
CONTROLLING TRAFFIC LIGHTS WITH AN IR SENSOR AND ARDUINO MICROPROCESSORS	18
1. INTRODUCTION	18
2.SYSTEM DISIGN	19
2.1. Automatic (Cyclic) Operation	19
2.2. IR Remote “Override” Mode	19
2.3. Operating Modes in the Project	19
3. Hardware Overview	20
3.1. Main Components	20
3.2. Arduino Pin Assignments for Each Direction’s LEDs	20
4. Circuit Diagrams	21
4.1. Proteus Simulation Schematic	21
4.2. Physical Breadboard Layout (Photograph)	21
5. Software and Code Explanation	22
5.1. Pin and IR-Button Definitions	22
5.2. Enumeration of Modes and States	23
5.3. Helper Functions	24
5.4. void setup()	27

5.5. void loop().....	28
Code Flow Summary:	29
6. Experiment and Results	30
6.1. Experimental Setup	30
6.2. Observations in Automatic Mode	30
6.3. Observations in IR “Override” Mode	30
6.4. Potential Improvements	31
CONCLUSION.....	32
REFERENCES	33

PREFACE

In the fields of digital systems, logic design, and microprocessor applications, autonomous control mechanisms like traffic light controllers provide a strong foundation for understanding sequential circuits, timing, and sensor-based transitions. This report presents a project that addresses traffic control through two complementary approaches: a hardware-based Finite State Machine (FSM) implementation and a microprocessor-based Arduino system with IR remote override.

The first phase, within the scope of Digital System Design, focuses on a hardware-only solution using D-type flip-flops, 555 timers, logic gates, and sensor inputs. It demonstrates FSM modeling and timing coordination, from Proteus simulation to PCB fabrication, emphasizing hands-on circuit design and system-level integration.

The second phase, under Microprocessors, uses an Arduino Uno to control a four-way intersection with two modes: automatic (cyclic) operation and an override mode activated via IR remote for priority traffic. This part highlights software-based control, real-time decision-making, and external signal interaction.

This report documents both systems in detail, comparing hardware- and software-based approaches, and reflects on the engineering experience and practical skills gained throughout the development process.

DIJITAL SYSTEM DESIGN

1.INTRODUCTION

The increasing demand for intelligent and adaptive traffic management systems necessitates the integration of digital logic-based control units in modern infrastructure. Traffic congestion, particularly at intersections involving low-traffic side roads, calls for responsive systems that can manage timing dynamically based on real-time conditions. Traditional time-fixed controllers fall short in these scenarios, either causing unnecessary delays or underutilizing available green time. In this context, the design of a hardware-based traffic light controller using flip-flops and logic gates presents a cost-effective and reliable solution, particularly in systems where microcontrollers or programmable logic are not preferred.

This project focuses on the design and implementation of a finite state machine (FSM)-based traffic light control system using discrete digital components. The controller operates in four primary states—Main Road Green, Main Road Yellow, Side Road Green, and Side Road Yellow—and transitions between these states based on a periodic timing signal generated via a 555 timer circuit. Importantly, a digital sensor (implemented via a push-button) is used to simulate vehicle presence on the side road. When no vehicle is detected, the system keeps the main road green by default, optimizing traffic flow. Upon sensor activation, the FSM engages and transitions the light cycle, allowing side-road traffic to proceed.

Unlike software-driven controllers, the proposed design employs synchronous state transitions using D flip-flops, allowing for deterministic operation without code. Logic gates are used to decode states and drive the corresponding LED outputs representing traffic signals. Adjustable timing is achieved through resistor-capacitor networks or configurable logic, ensuring flexibility in light durations. The final system was simulated and verified in Proteus, then translated into a printed circuit board (PCB) for practical implementation and testing.

This report outlines the design methodology, logic circuit implementation, and PCB realization of the system, providing a complete academic perspective from theory to application.

2.LITERATURE REVIEW

In the design and implementation of digital control systems such as traffic light controllers, several key electronic components and theoretical concepts are foundational. This section presents a review of the main elements utilized in the project, including flip-flops, timers, logic gates, and the finite state machine (FSM) model.

1. Flip-Flops (D Tipi Flip-Flop)

Flip-flops are bistable multivibrators used to store binary data. The D (Data or Delay) flip-flop, in particular, is a fundamental sequential logic element that captures the value on its data input (D) at the moment of a clock edge and holds that value until the next clock cycle. In this project, D flip-flops were used to implement the memory elements of the FSM, where each flip-flop represents a state bit. This enables predictable and synchronized state transitions based on clock pulses.

2. 555 Timer

The 555 timer is a highly stable integrated circuit used for generating accurate time delays and oscillation. Configured in astable mode, it produces a continuous square wave output, which serves as the clock signal for the FSM in this system. The period and duty cycle of the signal are determined by the values of external resistors and a capacitor. The timer ensures consistent transitions between the traffic light states at predetermined intervals, which can be adjusted through RC network modification.

3. Logic Gates

Basic logic gates—AND, OR, NOT, NAND, NOR, XOR, and XNOR—are the building blocks of digital circuits. These gates are used to decode the current state of the FSM and to control the illumination of traffic lights (LEDs). For instance, specific combinations of state bits are detected using AND gates to determine when the main road should show green or when the side road should transition to yellow. The use of combinational logic allows for real-time reaction to sensor inputs and ensures deterministic behavior.

4. Finite State Machine (FSM)

A finite state machine is an abstract computational model composed of a finite number of states, transitions between those states, and outputs determined by the current state and inputs. In this traffic light controller project, the FSM governs the behavior of the lights based on two main factors: elapsed time and the presence of a vehicle on the side road (sensor input). States such as Main Green, Main Yellow, Side Green, and Side Yellow are encoded using flip-flops, and the transitions between these states are triggered by clock pulses and influenced by the sensor.

5. Sensor Integration

A push-button represents the vehicle detection sensor in this system. When unpressed, the circuit continuously cycles the main road green light, simulating a scenario with no side-road traffic. When the button is pressed, the FSM is activated, allowing a complete light cycle including the side road to be executed. This approach adds conditional logic to the system, allowing adaptive control based on real-time demand.

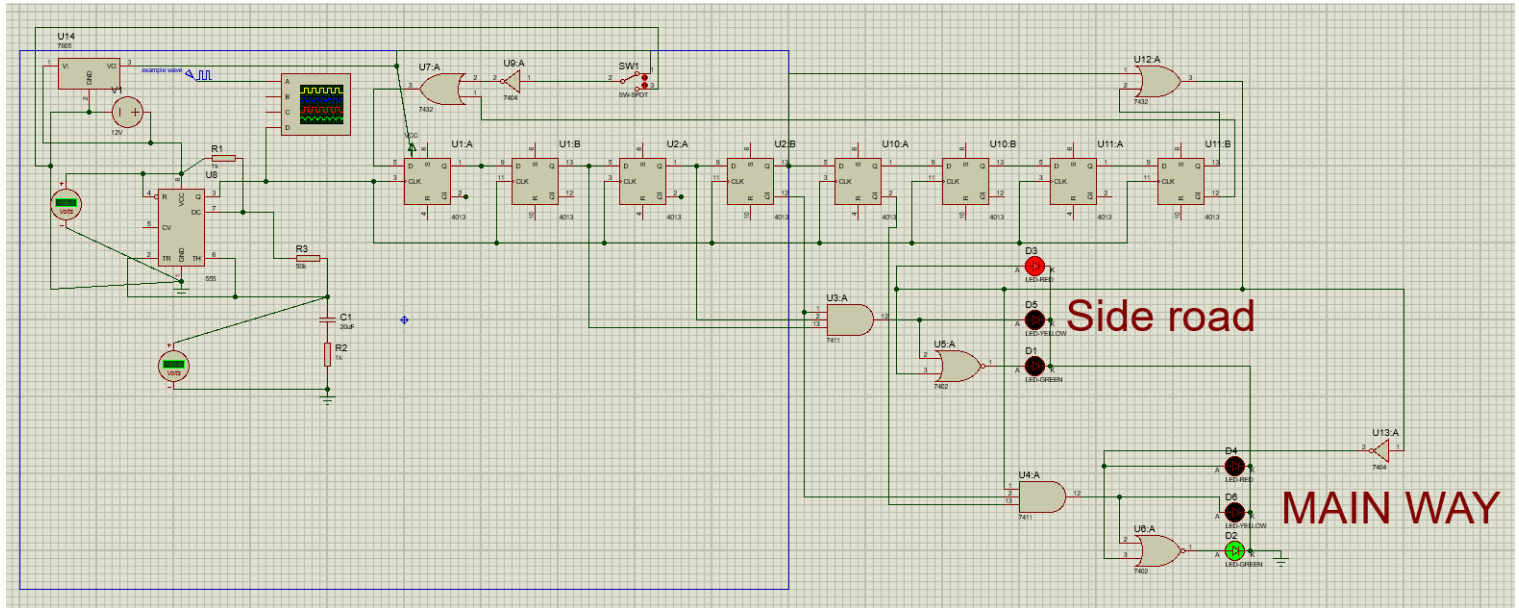
6. Proteus Simulation and PCB Design

In this project, Proteus Design Suite was used to simulate the digital traffic light controller circuit before hardware implementation. The simulation environment allowed for verifying correct state transitions, timing control via the 555 timer, and sensor-based behavior using logic gates and flip-flops.

Once the simulation was validated, the design was transferred to the PCB layout tool within Proteus. A compact and efficient board layout was created, ensuring proper component placement and signal routing. The PCB was then fabricated and soldered, and the final hardware was tested to confirm that it functioned identically to the simulation, validating the overall design.

3.SYSTEM DESIGN

1.Circuit Overview



Component	Description
2 x Traffic Light Sets	Each set includes Red, Yellow, and Green LEDs for Main and Side roads
555 Timer	Generates a periodic clock pulse
4 x CD4013 ICs	Each contains 2 D Flip-Flops — total of 8 Flip-Flops used in the circuit
Push Button	Simulates a digital vehicle sensor on the side road
Logic Gates	AND, OR, NOT gates used to manage light control logic

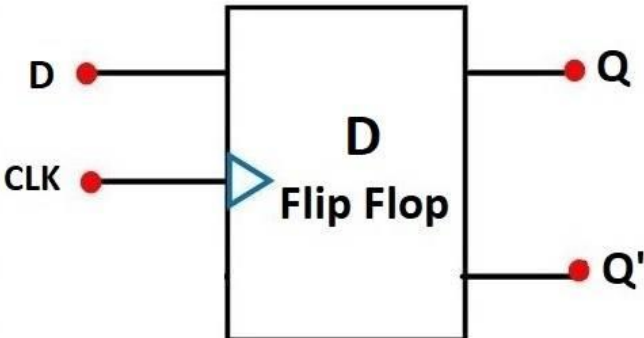
Tablo 1

2. Fundamental Component: D-Type Flip-Flop

The basic memory element used in this FSM is the D-type flip-flop. Its primary function is to store a single bit of data (0 or 1) present at its D (Data) input when triggered by a clock signal edge (typically the rising edge). The stored value appears at the Q output. The basic truth table for a rising-edge triggered D-type flip-flop is:

Truth Table of DFlip Flop

D	CLK	Q	Q'
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0



Tablo 2

3.Flip-Flop Utilization and FSM Design in the Traffic Light Controller:

In this traffic light controller system, we have implemented a **finite state machine (FSM)** that governs the behavior of the lights at a two-way intersection, controlled by the presence of a vehicle on the side road.

State	Description	Main Road Light	Side Road Light	Transition Condition
S0	Main road green	Green	Red	Button pressed → start FSM
S1	Main road yellow	Yellow	Red	Clock pulse
S2	Side road green	Red	Green	Clock pulse
S3	Side road yellow	Red	Yellow	Clock pulse
S0	Back to main green	Green	Red	Clock pulse

Tablo 3

- These states transition in a cyclic sequence only when a car is detected on the side road via a sensor (push button).
 - If no car is detected, the system remains in S0 (Main Green).
 - If a car is detected, the FSM transitions as follows:
 $S0 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S0$ (loop)

3.1 Flip-Flop Allocation in the FSM

The circuit utilizes 8 D-type Flip-Flops (IC: 4013), each identified and used for specific purposes. Below is a detailed description of why and how each flip-flop is used, with reference to the updated circuit:

<i>Flip-Flop</i>	<i>Function in the Circuit</i>
U1:A	Acts as the first stage in the state sequence/shift register. Its input is controlled by the sensor (SW1) logic (via U7:A, U9:A) and feedback (via U12:A), initiating the traffic light sequence when triggered.
U1:B	Second stage of the state sequence. Receives data from U1:A's Q output, holding the second bit of the FSM state.
U2:A	Third stage of the state sequence. Receives data from U1:B's Q output, holding the third bit of the FSM state.
U2:B	Fourth stage of the state sequence. Receives data from U2:A's Q output. Its output likely contributes to decoding the main road signal states via logic gates (e.g., U6:A).
U10:A	Fifth stage of the state sequence. Receives data from U2:B's Q output. Its output (Q and Q') is used in logic gates (U3:A, U5:A) to control side road lights (Red, Yellow, Green).
U10:B	Sixth stage of the state sequence. Receives data from U10:A's Q output. Its output likely contributes to defining later stages of the traffic light cycle.
U11:A	Seventh stage of the state sequence. Receives data from U10:B's Q output. Its output contributes to the logic controlling main road signals (via U4:A).
U11:B	Eighth and final stage observed in this chain. Receives data from U11:A's Q output. Its output feeds back into the logic (via U12:A) possibly to reset the sequence or hold the state

Tablo 4

3.2 FSM Structure and Operation


The FSM in this circuit utilizes eight D-type flip-flops (identified as U1A, U1B, U2A, U2B, U10A, U10B, U11A, U11B in the provided description). These flip-flops are connected serially: the Q output of one flip-flop connects to the D input of the next flip-flop in the chain. All flip-flops share a common clock signal, ensuring synchronous operation.

This configuration forms a shift register. With each active clock edge (assumed rising edge), the data bit stored in each flip-flop is shifted to the subsequent flip-flop in the chain. The state of the entire FSM, represented by the collective outputs of all flip-flops (QA through QH), advances one step.

The sequence of states is primarily determined by the data fed into the D input of the first flip-flop (DA for U1A). This input value (DAX before clock edge X) is controlled by external logic, including the sensor input (SW1) and feedback from the last flip-flop in the chain (QH from U11B), processed through logic gates (U7A, U9A, U12A). This control logic dictates whether the sequence continues, resets, or holds a specific pattern.

The following table conceptually illustrates the state transition based on the shift register

3.3 FSM-Based LED State Transitions for Traffic Light Controller (with Clock Pulse Trigger)

Clock Pulse	Current State (Q7...Q0)	D0 Input	Next State (Q7...Q0)	Main Road LEDs	Side Road LEDs	State Description
1 (Triggered)	00000001 (Q0)	1	00000010 (Q1)	 Green	 Red	Main road traffic flows
1 (Triggered)	00000010 (Q1)	1	00000100 (Q2)	 Green	 Red	Main road continues
1 (Triggered)	00000100 (Q2)	1	00001000 (Q3)	 Green	 Red	Main road continues
1 (Triggered)	00001000 (Q3)	1	00010000 (Q4)	 Green	 Red	Main road continues
1 (Triggered)	00010000 (Q4)	1	00100000 (Q5)	 Yellow	 Red	Prepare to stop main road
1 (Triggered)	00100000 (Q5)	1	01000000 (Q6)	 Red	 Red	Transition phase
1 (Triggered)	01000000 (Q6)	1	10000000 (Q7)	 Red	 Green	Side road traffic flows
1 (Triggered)	10000000 (Q7)	0	00000000 (Reset)	 Red	 Green	Side road continues
1 (Triggered)	00000000 (Reset)	1	00000001 (Q0)	 Red	 Yellow	Prepare to stop side road

Tablo 5

3.3 State Decoding and Control Logic

While the table above shows the progression of state bits within the FSM, it does not directly indicate which traffic lights are active. The specific combination of Q outputs (QA through QH) representing each traffic light phase (e.g., Main Road Green) is determined by the combinational output logic gates (identified as U3, U4, U5, U6, etc. in the schematic). These gates decode the current state vector from the flip-flops to activate the appropriate LEDs (D1-D6).

Furthermore, the input control logic (involving SW1 sensor, U7, U9, U12) plays a crucial role in managing the FSM sequence. It determines the value shifted into the first flip-flop (DA) based on external conditions (sensor activation) and internal state (feedback), thereby controlling the initiation, progression, and potentially the termination or looping of the traffic light cycle.

4. PHYSICAL IMPLEMENTATION: BREADBOARD AND PCB

2. BREADBOARD IMPLEMENTATION

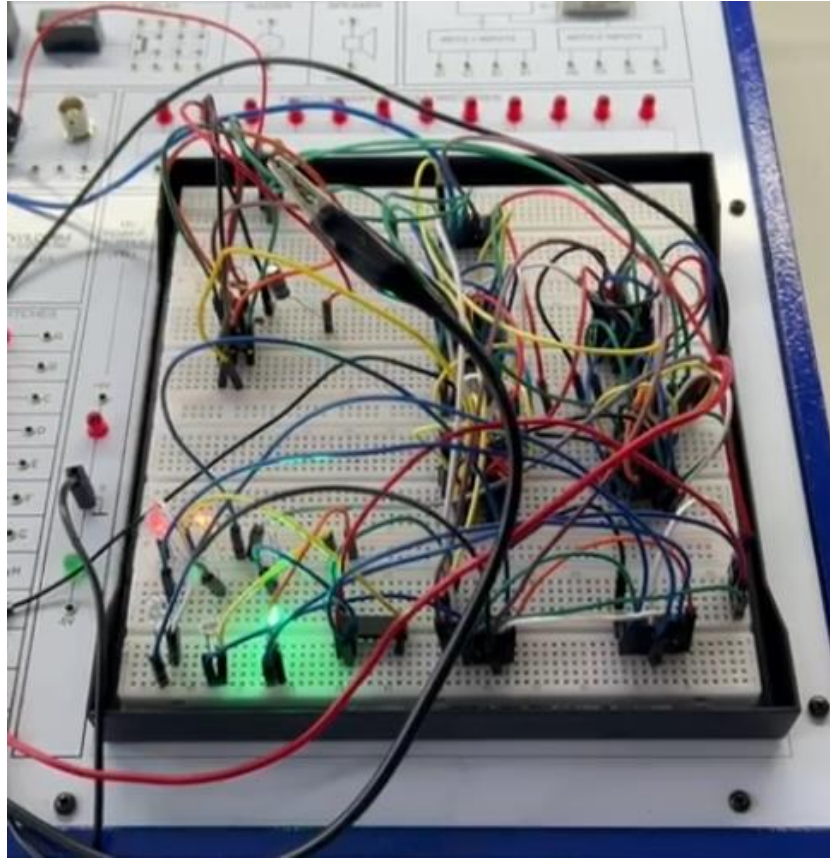
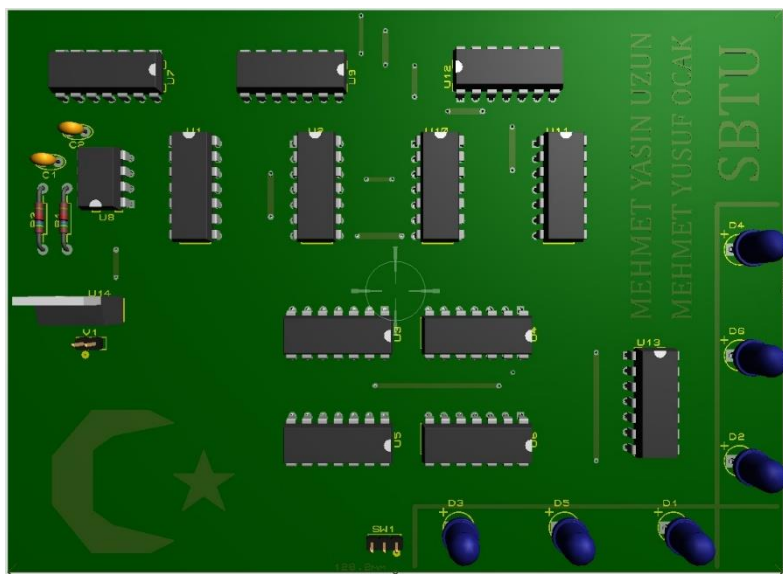
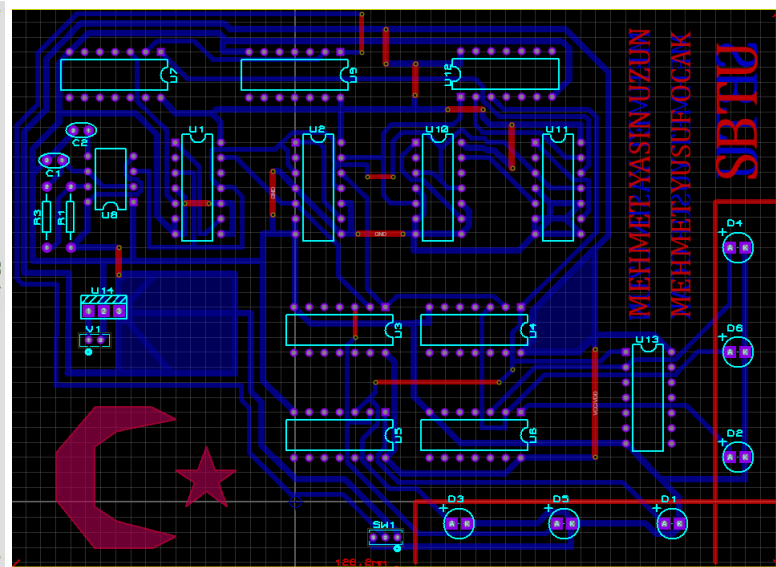


Figure 1

1. PCB PART



PCB 3D Visualizer



PCB Layout

3.PRODUCTION STAGE OF PCB

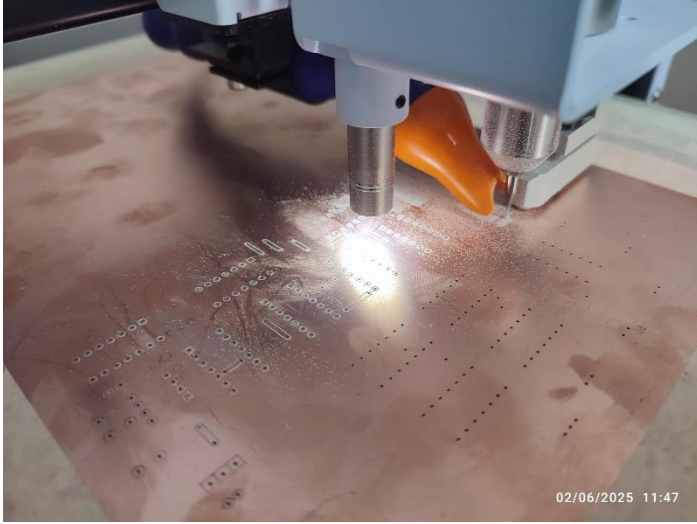


Figure 3

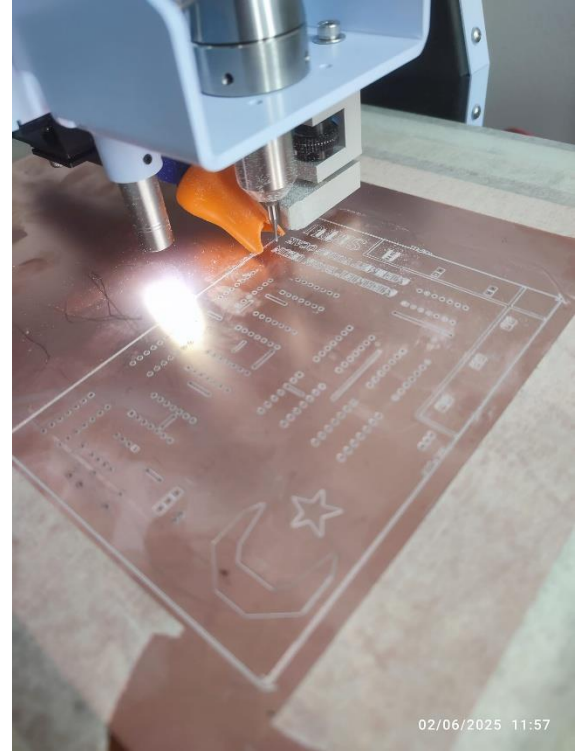


Figure 2

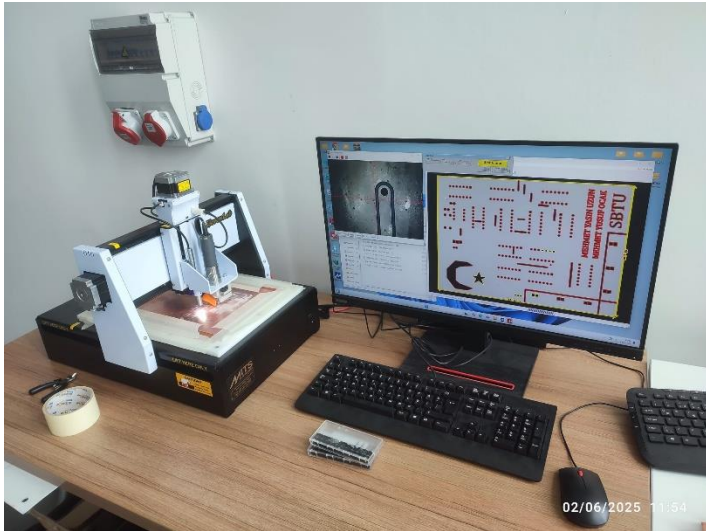


Figure 5

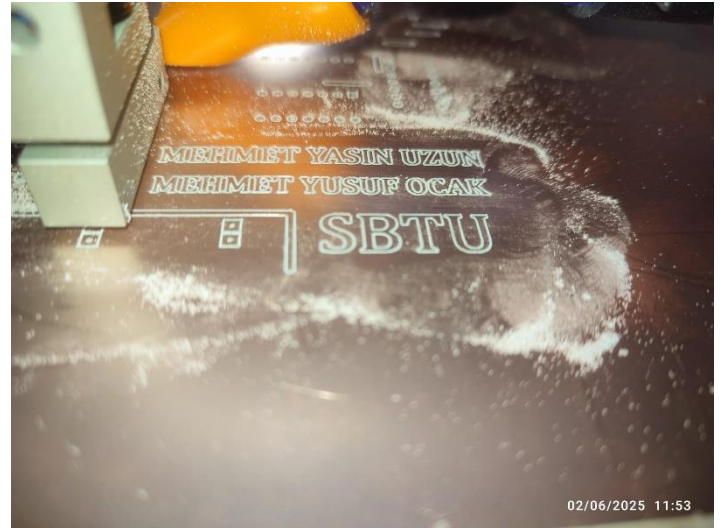


Figure 4

4.PCB IMPLEMENTATION

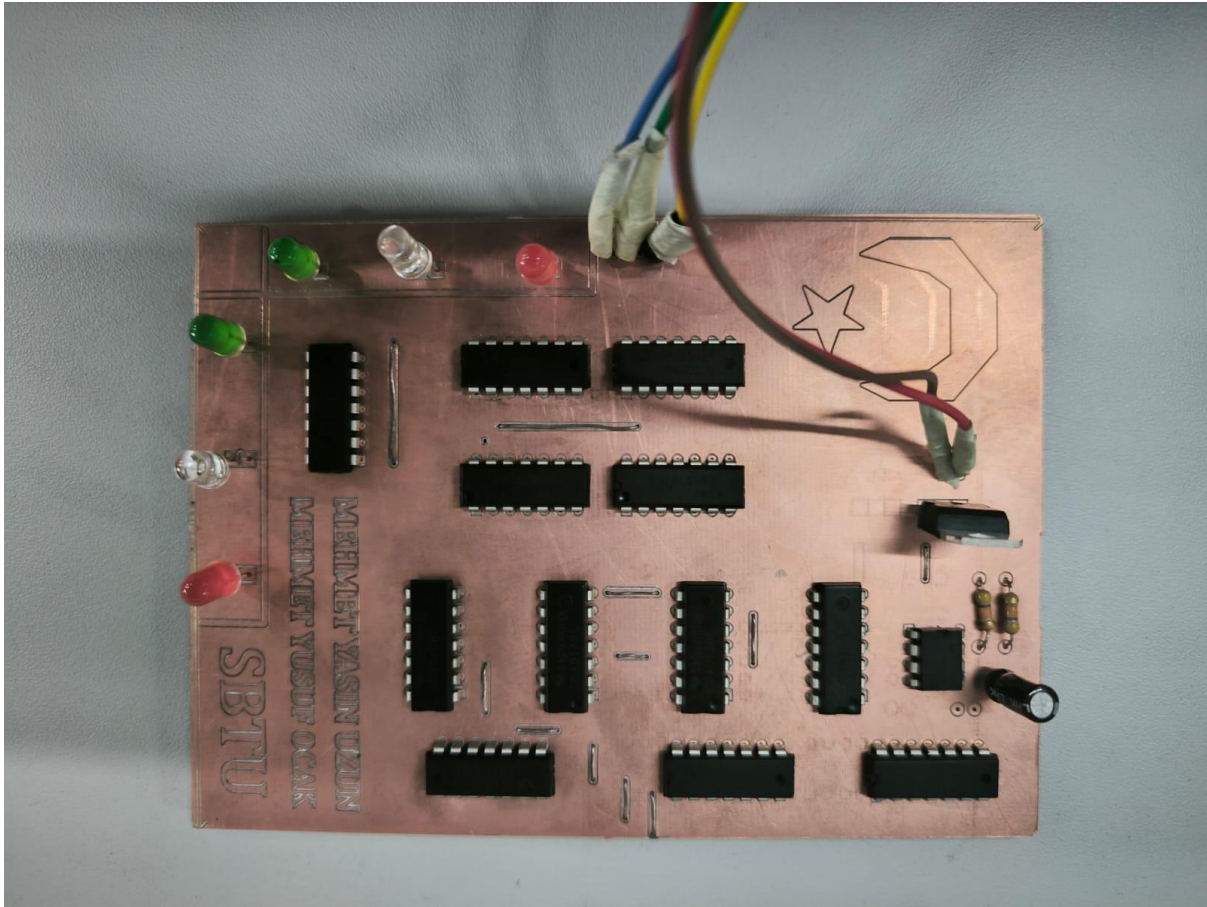


Figure 6

The images above illustrate the full process of printed circuit board (PCB) manufacturing and implementation. Initially, the PCB layout was created and engraved onto a copper-clad board using a CNC milling machine, as shown in Figures 1 through 4. This process included the precise drilling of component holes and the isolation of copper traces according to the digital design file. A computer-controlled system was used to manage the milling machine and ensure high accuracy. In the final stage, depicted in Figure 5, electronic components were mounted and soldered onto the milled PCB. These components include integrated circuits, resistors, LEDs, and connectors. This step completes the functional implementation of the designed circuit on the PCB.

CONTROLLING TRAFFIC LIGHTS WITH AN IR SENSOR AND ARDUINO MICROPROCESSORS

1. INTRODUCTION

The primary goal of this project is to control the lights at a four-way traffic intersection in two distinct modes: automatic (cyclic) operation and “override” (priority) operation via an IR (infrared) remote. In particular, the system is designed to handle scenarios such as:

- Emergency vehicles (fire trucks, ambulances, police cars, etc.) approaching the intersection, requiring the traffic flow to be directed so that the emergency vehicle always receives a green light,
- Holiday or special-event traffic (for example, during long queues or unusually heavy traffic in one particular direction, such as on a holiday), where it may be desirable to give extended green-light priority to a single direction.

Using an Arduino Uno platform, an IR receiver module, standard LEDs, and resistors, we have developed a setup that operates in automatic cycle mode (in which each direction’s green and yellow intervals follow a fixed sequence) and also in override mode (in which an IR command immediately forces one chosen direction to green and suspends the automatic cycle). This design allows the intersection to function normally under typical traffic conditions, while also giving us the flexibility to prioritize emergency vehicles or relieve heavy one-way congestion when needed. The result is both an educational simulation of traffic-light control and a functional prototype that can be tested in real hardware for urgent or atypical traffic scenarios.

2.SYSTEM DISIGN

2.1. Automatic (Cyclic) Operation

In automatic mode (MODE_AUTO), the traffic lights change in the following fixed sequence and timing:

1. North-Green (6 seconds) → North-Yellow (3 seconds)
2. South-Green (6 seconds) → South-Yellow (3 seconds)
3. East-Green (6 seconds) → East-Yellow (3 seconds)
4. West-Green (6 seconds) → West-Yellow (3 seconds)

After West-Yellow, the cycle returns to North-Green, and the process repeats indefinitely.

The chosen durations (6 s for green, 3 s for yellow) are left in the code as example values and are not analyzed further in this report.

2.2. IR Remote “Override” Mode

When an IR command is received, the system switches to override mode (MODE_OVERRIDE). In override mode:

1. All four directions' lights immediately turn red (setAllRed()).
2. Only the direction corresponding to the received IR button becomes green.
3. Pressing the “5” button on the IR remote returns the system to automatic cyclic operation.

Thus, an IR command will interrupt the normal cycle and force a chosen direction to green (for instance, for an emergency vehicle or to relieve a single-lane traffic jam), and pressing “5” restores the normal automatic sequence.

2.3. Operating Modes in the Project

- MODE_AUTO: The traffic lights run in a predetermined cycle, with fixed green and yellow intervals for each direction.
- MODE_OVERRIDE: As soon as a valid IR code arrives, the cycle is suspended and the specified direction's green light is turned on. Pressing the designated “return to auto” button on the IR remote sets the system back to automatic mode.

3. Hardware Overview

3.1. Main Components

1. Arduino Uno (ATmega328P-based microcontroller board)
2. IR Receiver Module (e.g., VS1838B or a similar IR demodulator):
 - The module's VCC, GND, and OUT pins are connected to the Arduino.
 - The OUT pin is wired to Arduino's analog pin A3 (which is addressed as digital 17).
3. LEDs and Resistors:
 - Three LEDs (Red, Yellow, Green) for each of the four directions (12 LEDs total).
 - Each LED is connected in series with a 220 Ω resistor to limit current to ~ 15 mA.
4. Breadboard and Jumper Wires:
 - The Arduino's 5 V and GND rails are distributed across the breadboard, and each LED/resistor pair is wired to the appropriate digital pin.
5. Mobile Phone IR Remote App:
 - A smartphone app (e.g., "IR Remote") was used to generate IR signals corresponding to the "2, 4, 5, 6, 8" buttons.

3.2. Arduino Pin Assignments for Each Direction's LEDs

The table below summarizes how each LED (red, yellow, green) is connected to Arduino digital pins:

<i>Direction</i>	<i>Red LED Pin</i>	<i>Yellow LED Pin</i>	<i>Green LED Pin</i>
<i>North</i>	PIN_NORTH_RED = 2	PIN_NORTH_YELLOW = 3	PIN_NORTH_GREEN = 4
<i>South</i>	PIN_SOUTH_RED = 5	PIN_SOUTH_YELLOW = 6	PIN_SOUTH_GREEN = 7
<i>East</i>	PIN_EAST_RED = 8	PIN_EAST_YELLOW = 9	PIN_EAST_GREEN = 10
<i>West</i>	PIN_WEST_RED = 11	PIN_WEST_YELLOW = 12	PIN_WEST_GREEN = 13

Tablo 6

Note: On an Arduino Uno, analog pin A3 can be referred to as digital pin 17. Therefore, we connect the IR receiver's output to digital 17 (A3):

```
const int RECV_PIN = 17; // Arduino Uno: A3 = digital 17
```

4. Circuit Diagrams

4.1. Proteus Simulation Schematic

Figure 7: In Proteus, the four-way intersection is drawn, showing how the Arduino Uno's digital pins connect to each of the twelve LEDs (each with its $220\ \Omega$ resistor). The IR receiver's VCC, GND, and OUT pins are also wired to Arduino 5 V, GND, and pin 17 (A3), respectively.

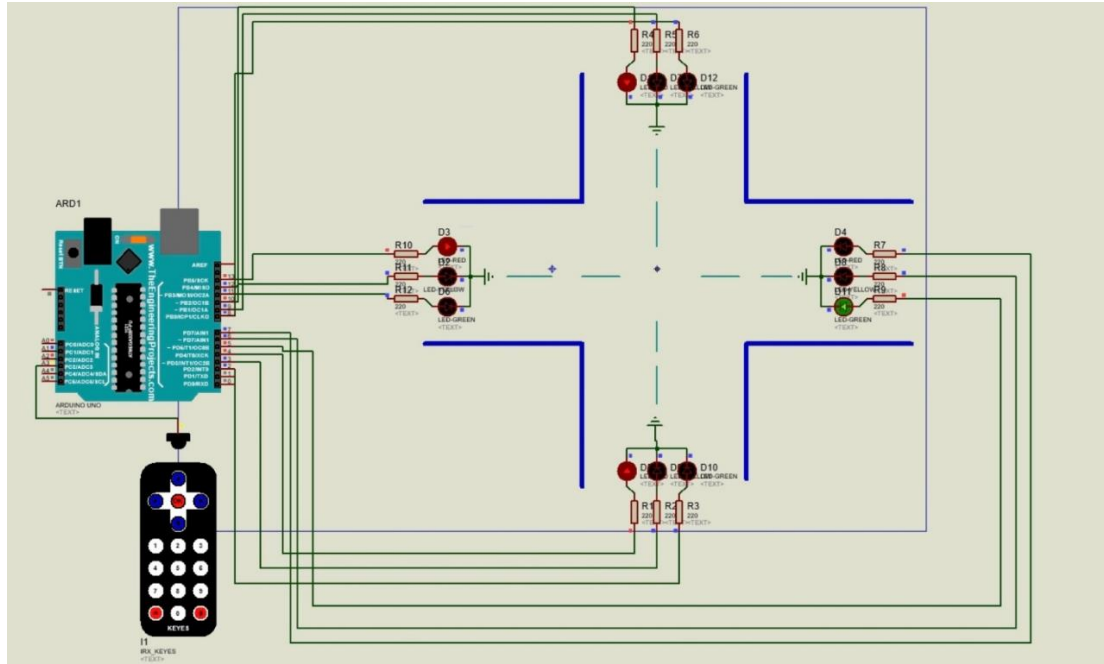


Figure 7

4.2. Physical Breadboard Layout (Photograph)

Figure 8: This photograph shows the real hardware setup on a breadboard:

- The Arduino Uno is seated next to the breadboard, with its 5 V and GND rails powering the breadboard's power buses.
- Twelve LEDs (four groups of Red/Yellow/Green) are mounted on the breadboard, each paired with a $220\ \Omega$ resistor in series.
- The IR receiver module is placed near the top of the breadboard, and its output pin is wired to Arduino A3 (digital 17).

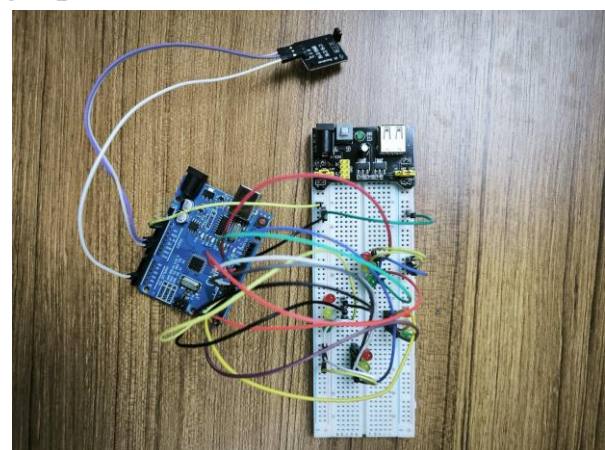


Figure 8

5. Software and Code Explanation

Below is a detailed walkthrough of the Arduino sketch used to implement both the automatic cycle and the IR override functionality. The code relies on the IRremote library to decode incoming infrared signals and uses a finite-state machine approach to manage traffic light transitions.

```
#include <IRremote.h>
/*
  The IRremote library (v4.x) provides functions such as:
  - IrReceiver.begin()
  - IrReceiver.decode()
  - IrReceiver.decodedIRData.decodedRawData, etc.
*/
```

5.1. Pin and IR-Button Definitions

At the top of the code, we define which pins drive each LED and we store the 32-bit IR codes for specific buttons:

```
// IR receiver is connected to Arduino Uno's A3 (digital 17).
const int RECV_PIN = 17;

// LED pin assignments for each direction:

// North direction LEDs
const int PIN_NORTH_RED    = 2;  // North Red LED
const int PIN_NORTH_YELLOW = 3;  // North Yellow LED
const int PIN_NORTH_GREEN  = 4;  // North Green LED

// South direction LEDs
const int PIN_SOUTH_RED    = 5;  // South Red LED
const int PIN_SOUTH_YELLOW = 6;  // South Yellow LED
const int PIN_SOUTH_GREEN  = 7;  // South Green LED

// East direction LEDs
const int PIN_EAST_RED     = 8;   // East Red LED
const int PIN_EAST_YELLOW  = 9;   // East Yellow LED
const int PIN_EAST_GREEN   = 10;  // East Green LED

// West direction LEDs
const int PIN_WEST_RED     = 11;  // West Red LED
const int PIN_WEST_YELLOW  = 12;  // West Yellow LED
const int PIN_WEST_GREEN   = 13;  // West Green LED

// IR button codes (NEC protocol, 32-bit values):
#define BUTTON_2 0xEE115000  // Pressing "2" → North priority
#define BUTTON_4 0xED125000  // Pressing "4" → West priority
#define BUTTON_5 0xEC135000  // Pressing "5" → Return to auto mode
#define BUTTON_6 0xEB145000  // Pressing "6" → East priority
#define BUTTON_8 0xEA155000  // Pressing "8" → South priority
```

- **RECV_PIN:** Indicates that the IR receiver's output is wired to Arduino digital pin 17 (i.e., A3).

- **PIN_<DIRECTION>_<COLOR>**: Assigns each of the twelve LEDs to a specific Arduino digital pin.
- **BUTTON_X**: Defines each IR remote button's unique 32-bit code so we can compare incoming codes against these values.

IR Button Codes Summary Table

<i>Button</i>	<i>IR Code (Hex)</i>	<i>Macro Name</i>	<i>Action / Green Direction</i>
2	0xEE115000	BUTTON_2	North lane becomes green (STATE_N_GREEN or override)
4	0xED125000	BUTTON_4	West lane becomes green (STATE_W_GREEN or override)
5	0xEC135000	BUTTON_5	Return to automatic mode (MODE_AUTO)
6	0xEB145000	BUTTON_6	East lane becomes green (STATE_E_GREEN or override)
8	0xEA155000	BUTTON_8	South lane becomes green (STATE_S_GREEN or override)

Tablo 7

Note: In override mode, pressing any of the above direction buttons immediately forces the chosen direction to green, and all other directions turn red.

5.2. Enumeration of Modes and States

We implement two enumerations to handle the system's logic:

```
// "Mode" indicates whether we are in automatic cycling or override:
enum Mode {
    MODE_AUTO,          // Traffic lights cycle automatically
    MODE_OVERRIDE       // An IR command has overridden the cycle
};
Mode currentMode = MODE_AUTO; // Start in automatic mode

// "AutoState" enumerates the eight possible automatic states:
enum AutoState {
    STATE_N_GREEN,      // North: Green
    STATE_N_YELLOW,     // North: Yellow
    STATE_S_GREEN,      // South: Green
    STATE_S_YELLOW,     // South: Yellow
    STATE_E_GREEN,      // East: Green
    STATE_E_YELLOW,     // East: Yellow
    STATE_W_GREEN,      // West: Green
}
```



```
STATE_W_YELLOW    // West: Yellow
};
AutoState currentState = STATE_N_GREEN;    // Initial state: North-Green

// Timer variable to track how long we've been in the current state:
unsigned long stateStartTime = 0;

// Duration constants (example values):
const unsigned long GREEN_TIME  = 6000UL; // 6 seconds in milliseconds
const unsigned long YELLOW_TIME = 3000UL; // 3 seconds in milliseconds
```

- **currentMode:** Tracks whether the system is in `MODE_AUTO` (automatic cycling) or `MODE_OVERRIDE` (an IR button has taken priority).
- **currentState:** Indicates which of the eight automatic sequence states is active.
- **stateStartTime:** Stores the value of `millis()` when a new state begins.
- **GREEN_TIME, YELLOW_TIME:** Define how long each green or yellow interval lasts in automatic mode.

5.3. Helper Functions

5.3.1. *void setAllRed()*

Turns all directions' LEDs to red (setting each red LED HIGH, and all yellow/green LEDs LOW).

```
void setAllRed() {
    // North direction: only the red LED remains on
    digitalWrite(PIN_NORTH_RED,    HIGH);
    digitalWrite(PIN_NORTH_YELLOW, LOW);
    digitalWrite(PIN_NORTH_GREEN,  LOW);

    // South direction: only the red LED remains on
    digitalWrite(PIN_SOUTH_RED,    HIGH);
    digitalWrite(PIN_SOUTH_YELLOW, LOW);
    digitalWrite(PIN_SOUTH_GREEN,  LOW);

    // East direction: only the red LED remains on
    digitalWrite(PIN_EAST_RED,     HIGH);
    digitalWrite(PIN_EAST_YELLOW,  LOW);
    digitalWrite(PIN_EAST_GREEN,   LOW);

    // West direction: only the red LED remains on
    digitalWrite(PIN_WEST_RED,     HIGH);
    digitalWrite(PIN_WEST_YELLOW,  LOW);
    digitalWrite(PIN_WEST_GREEN,   LOW);
}
```

Function:

- Whenever we need to reset all lights to red (either before starting an override or as the first step in changing states), we call `setAllRed()`.
- This ensures a known baseline before activating a single green or yellow LED.

5.3.2. *void applyAutoState()*

Based on the current `currentState` value, this function sets exactly one direction's LED to green or yellow, while all others remain red.

```
void applyAutoState() {  
    // 1) First, turn all lights red  
    setAllRed();  
  
    // 2) Then, switch on the appropriate LED for the current state  
    switch (currentState) {  
        case STATE_N_GREEN:  
            digitalWrite(PIN_NORTH_RED, LOW);  
            digitalWrite(PIN_NORTH_GREEN, HIGH);  
            break;  
  
        case STATE_N_YELLOW:  
            digitalWrite(PIN_NORTH_RED, LOW);  
            digitalWrite(PIN_NORTH_YELLOW, HIGH);  
            break;  
  
        case STATE_S_GREEN:  
            digitalWrite(PIN_SOUTH_RED, LOW);  
            digitalWrite(PIN_SOUTH_GREEN, HIGH);  
            break;  
  
        case STATE_S_YELLOW:  
            digitalWrite(PIN_SOUTH_RED, LOW);  
            digitalWrite(PIN_SOUTH_YELLOW, HIGH);  
            break;  
  
        case STATE_E_GREEN:  
            digitalWrite(PIN_EAST_RED, LOW);  
            digitalWrite(PIN_EAST_GREEN, HIGH);  
            break;  
  
        case STATE_E_YELLOW:  
            digitalWrite(PIN_EAST_RED, LOW);  
            digitalWrite(PIN_EAST_YELLOW, HIGH);  
            break;  
  
        case STATE_W_GREEN:  
            digitalWrite(PIN_WEST_RED, LOW);  
            digitalWrite(PIN_WEST_GREEN, HIGH);  
            break;  
  
        case STATE_W_YELLOW:  
            digitalWrite(PIN_WEST_RED, LOW);  
            digitalWrite(PIN_WEST_YELLOW, HIGH);  
            break;  
    }  
}
```

Function:

- This function is called whenever `currentState` changes. It resets all lights to red, then turns on exactly one direction's green or yellow LED.
- This ensures there is never any conflict (e.g., two directions showing green simultaneously).

5.3.3. *void updateAutoState()*

In automatic mode, this function checks how long the system has been in the current state. If the elapsed time exceeds the designated green or yellow interval, it advances `currentState` to the next enumeration and calls `applyAutoState()` again.

```
void updateAutoState() {
    unsigned long now      = millis();
    unsigned long elapsed = now - stateStartTime;

    switch (currentState) {
        case STATE_N_GREEN:
            if (elapsed >= GREEN_TIME) {
                currentState = STATE_N_YELLOW;
                stateStartTime = now;
                applyAutoState();
            }
            break;

        case STATE_N_YELLOW:
            if (elapsed >= YELLOW_TIME) {
                currentState = STATE_S_GREEN;
                stateStartTime = now;
                applyAutoState();
            }
            break;

        case STATE_S_GREEN:
            if (elapsed >= GREEN_TIME) {
                currentState = STATE_S_YELLOW;
                stateStartTime = now;
                applyAutoState();
            }
            break;

        case STATE_S_YELLOW:
            if (elapsed >= YELLOW_TIME) {
                currentState = STATE_E_GREEN;
                stateStartTime = now;
                applyAutoState();
            }
            break;

        case STATE_E_GREEN:
            if (elapsed >= GREEN_TIME) {
                currentState = STATE_E_YELLOW;
                stateStartTime = now;
                applyAutoState();
            }
            break;

        case STATE_E_YELLOW:
            if (elapsed >= YELLOW_TIME) {
                currentState = STATE_W_GREEN;
                stateStartTime = now;
                applyAutoState();
            }
            break;

        case STATE_W_GREEN:
            if (elapsed >= GREEN_TIME) {
```


Explanation:

1. **Serial.begin(115200):** Starts serial communication so we can print debug messages (e.g., which IR code was received).
2. **IrReceiver.begin(RECV_PIN, DISABLE_LED_FEEDBACK):** Initializes the IrReceiver library on pin 17 (A3) and disables its internal LED indicator, letting only the hardware IR module's LED show IR activity.
3. **pinMode(..., OUTPUT):** Sets each of the twelve LED pins as outputs so we can drive them HIGH or LOW.
4. We assign **MODE_AUTO**, set **currentState** to **STATE_N_GREEN**, set **stateStartTime = millis()**, and call **applyAutoState()** so that as soon as the Arduino powers up, the North-Green LED turns on and the automatic cycle begins.

5.5. void loop()

This is the main continuous execution loop. It checks for incoming IR signals, processes override commands, and—if in automatic mode—continues cycling through traffic states.

```
void loop() {
  // 1) Check if a new IR code has arrived
  if (IrReceiver.decode()) {
    unsigned long irCode = IrReceiver.decodedIRData.decodedRawData;

    // 2) Compare the incoming code with our defined IR button values
    if (irCode == BUTTON_2) {
      // If "2" is pressed:
      Serial.println(F("BUTTON_2"));
      currentMode = MODE_OVERRIDE; // Switch to override mode
      setAllRed();                 // Turn all lights red first
      // Then allow North-Green only
      digitalWrite(PIN_NORTH_RED, LOW);
      digitalWrite(PIN_NORTH_GREEN, HIGH);
    }
    else if (irCode == BUTTON_4) {
      // If "4" is pressed:
      Serial.println(F("BUTTON_4"));
      currentMode = MODE_OVERRIDE;
      setAllRed();
      digitalWrite(PIN_WEST_RED, LOW);
      digitalWrite(PIN_WEST_GREEN, HIGH);
    }
    else if (irCode == BUTTON_5) {
      // If "5" is pressed:
      Serial.println(F("BUTTON_5"));
      setAllRed();
      delay(2000); // Keep all red for 2 seconds
      // Then return to automatic mode at North-Green
      currentMode = MODE_AUTO;
      currentState = STATE_N_GREEN;
      stateStartTime = millis(); // Reset timer
      applyAutoState();          // Activate North-Green
    }
    else if (irCode == BUTTON_6) {
      // If "6" is pressed:
      Serial.println(F("BUTTON_6"));
```

```
        currentMode = MODE_OVERRIDE;
        setAllRed();
        digitalWrite(PIN_EAST_RED, LOW);
        digitalWrite(PIN_EAST_GREEN, HIGH);
    }
    else if (irCode == BUTTON_8) {
        // If "8" is pressed:
        Serial.println(F("BUTTON_8"));
        currentMode = MODE_OVERRIDE;
        setAllRed();
        digitalWrite(PIN_SOUTH_RED, LOW);
        digitalWrite(PIN_SOUTH_GREEN, HIGH);
    }
    else {
        // If the code does not match any of our button definitions:
        Serial.print(F("Received IR Code (RAW): 0x"));
        Serial.println(irCode, HEX);
    }

    // 3) Prepare the IR receiver to decode the next signal
    IrReceiver.resume();
}

// 4) If we are still in automatic mode, run the state machine
if (currentMode == MODE_AUTO) {
    updateAutoState();
}
// If in MODE_OVERRIDE, do nothing here—the lights remain as forced
// until a new IR command arrives.
}
```

Code Flow Summary:

1. **IrReceiver.decode()**: Checks if a new IR packet is available. If true, it reads the 32-bit decodedRawData.
2. **Compare irCode to Predefined Values:**
 - If the code equals `BUTTON_2`, `BUTTON_4`, `BUTTON_6`, or `BUTTON_8`, we switch to override mode, call `setAllRed()` to turn everything red, and then set only the chosen direction's green LED to HIGH.
 - If the code equals `BUTTON_5`, we do `setAllRed()`, wait 2 s (`delay(2000)`), and then return to automatic mode (`MODE_AUTO`), resetting `currentState = STATE_N_GREEN` and `stateStartTime = millis()`, and calling `applyAutoState()` so North-Green comes back on.
 - If none of those codes match, we print the raw IR code to Serial Monitor for debugging.
3. **IrReceiver.resume()**: After processing a code, we tell the IR library we're ready for the next incoming signal.
4. **Automatic Mode Check:** If `currentMode` is still `MODE_AUTO`, we call `updateAutoState()` so the traffic cycle continues. If `currentMode` is `MODE_OVERRIDE`, the lights remain as set (i.e., the forced green direction) until another IR command arrives.

6. Experiment and Results

6.1. Experimental Setup

1. The entire breadboard assembly (Arduino Uno + IR receiver + twelve LEDs + resistors) was connected via USB to a PC. The Arduino IDE was used to upload the code.
2. A smartphone IR remote application generated the IR commands corresponding to the buttons “2,” “4,” “5,” “6,” and “8.”
3. During testing, we observed both the automatic traffic cycle (green/yellow intervals) and each override scenario triggered by the IR remote.

6.2. Observations in Automatic Mode

- The traffic lights passed through the sequence:

North Green → North Yellow → South Green → South Yellow → East Green → East Yellow → West Green → West Yellow → (back to North Green)

Each green interval lasted 6 s, followed by a 3 s yellow interval, as defined.

- Proteus simulation confirmed identical timing and LED behavior.
- At no point did two directions show green simultaneously. Each transition was clean: the previous green turned yellow, then switched to red before the next direction’s green illuminated.

6.3. Observations in IR “Override” Mode

- **Button 2 (North Priority):** Pressing “2” via the mobile app forced all lights red, then immediately illuminated North-Green. The automatic cycle paused until further IR input.
- **Button 4 (West Priority):** Pressing “4” turned all lights red, then West-Green lit up. Automatic cycling halted.
- **Button 6 (East Priority):** Pressing “6” forced East-Green after turning everything red.
- **Button 8 (South Priority):** Pressing “8” forced South-Green after all-red.
- **Button 5 (Return to Auto):** Pressing “5” made all lights red for 2 s (delay), then restarted the automatic cycle at North-Green.

Conclusion: Each IR button correctly triggered the intended override behavior. Pressing “5” reliably returned the system to automatic cycling. The IR-to-LED response was immediate, demonstrating the code’s correct decoding and override logic.

6.4. Potential Improvements

- **IR Receiver Sensitivity:** In bright lighting or at greater distances, the IR receiver sometimes failed to decode the signal. Using a higher-sensitivity IR module or narrowing the module's field of view could improve reliability.
- **LED Brightness and Energy Efficiency:** Adjusting resistor values (e.g., switching from 220 Ω to 330 Ω) could reduce LED brightness and power consumption if needed, though 220 Ω was satisfactory for visibility in testing.

CONCLUSION

This project has successfully demonstrated two distinct yet complementary approaches to traffic light control systems—one grounded in pure hardware design using finite state machine (FSM) logic, and the other based on microprocessor-driven software control. Through the hardware-based design, we gained a deeper understanding of sequential logic, timing circuits, and sensor-triggered transitions by implementing flip-flop logic, 555 timer-based clocks, and logic gates. The complete process, from simulation in Proteus to PCB fabrication, provided hands-on experience in system integration and hardware troubleshooting.

In contrast, the Arduino-based implementation showcased the flexibility and adaptability of software-driven control systems. By incorporating IR remote override functionality, we simulated real-world scenarios such as emergency vehicle prioritization. This phase strengthened our skills in microcontroller programming, interrupt handling, and real-time control strategies.

Comparing the two systems highlighted the trade-offs between hardware rigidity and software flexibility. While hardware design ensures reliability and predictability, software systems offer greater ease of modification and feature expansion. Ultimately, this project enriched our understanding of digital system design, control logic, and the practical application of both hardware and software methodologies in solving real-world engineering problems.

REFERENCES

- <https://archive.org/details/DigitalDesign5thEditionByManoCiletti>
- <https://archive.org/details/IC555Projects>
- https://www.electronics-tutorials.ws/waveforms/555_timer.html
- <https://www.instructables.com/Arduino-Traffic-Light/>
- <https://archive.org/details/getting-started-with-arduino>
- <https://www.electronicwings.com/note/traffic-light-control-system-using-proteus>