

Designing an Automatic Traffic Light Controller for a T-Junction Road

Aaron Reason (2312456)

L5 Electronics 1 - ACDB5002

Embedded Electronics Degree Apprenticeship

Table of Contents

Table of Contents.....	2
Introduction.....	3
Traffic Lights Research.....	4
Circuit Design.....	6
Lights.....	6
Final design.....	8
Displays.....	9
Pseudocode.....	10
Software.....	12
Writing To The Displays.....	12
Traffic Lights Control.....	13
Entire program.....	14
Simulation Results.....	17
Stage 1.....	19
Stage 2.....	20
Stage 3.....	21
Stage 4.....	22
Conclusion.....	23
Future Recommendations.....	23
References.....	24

Introduction

This report details the circuit design and software implementation for a T-junction road with traffic lights. UK traffic signal laws have been followed where possible to produce as realistic a recreation as possible. The lights cycle to allow each lane of traffic to flow gradually over time in a controlled manner. Additionally, a phase for pedestrians to cross the road was implemented restricting all traffic flow to allow safe crossing.

For each traffic light there is a pair of 7-segment displays to indicate the time until all cycles are complete. This value starts at 40 and counts down until it reaches 0, with each second of time passed causing the value to decrement.

A PIC16F877A microcontroller was used to control external devices, such as the traffic lights or the 7-segment displays, while allowing the use of embedded software. This PIC was used for its extensive array of I/O ports and built-in features. The PIC16F877A features 44 total pins, with PORTA-D being full 8-bit ports. [1]

These pins can be configured to act as either inputs or outputs, however for this project there were no inputs so all pins were configured to act as outputs. Despite the large selection of pins, the amount of connections from the PIC were optimised as best they could be, using as few pins as possible. The PIC16F877A also has 8kb of program memory, which was more than necessary, but offered a comfortable safety buffer. [1]

The software was written using the C programming language and compiled with the CCS C compiler. All software was written so that it could be easily understood to anyone with a good level of understanding of programming in C.

All simulation was carried out using the Proteus Design Suite which enabled the design and simulation of a circuit. Additionally, Proteus is able to simulate the software, allowing for code development to be undertaken without requiring a physical PCB.

Traffic Lights Research

A key aspect to the simulation was understanding how traffic signal timings actually work and what each stage means. All required information can be extracted from the manual provided by the Department for Transport (DfT). Each traffic light state and what they mean can be seen in **Figure 1**: [2]

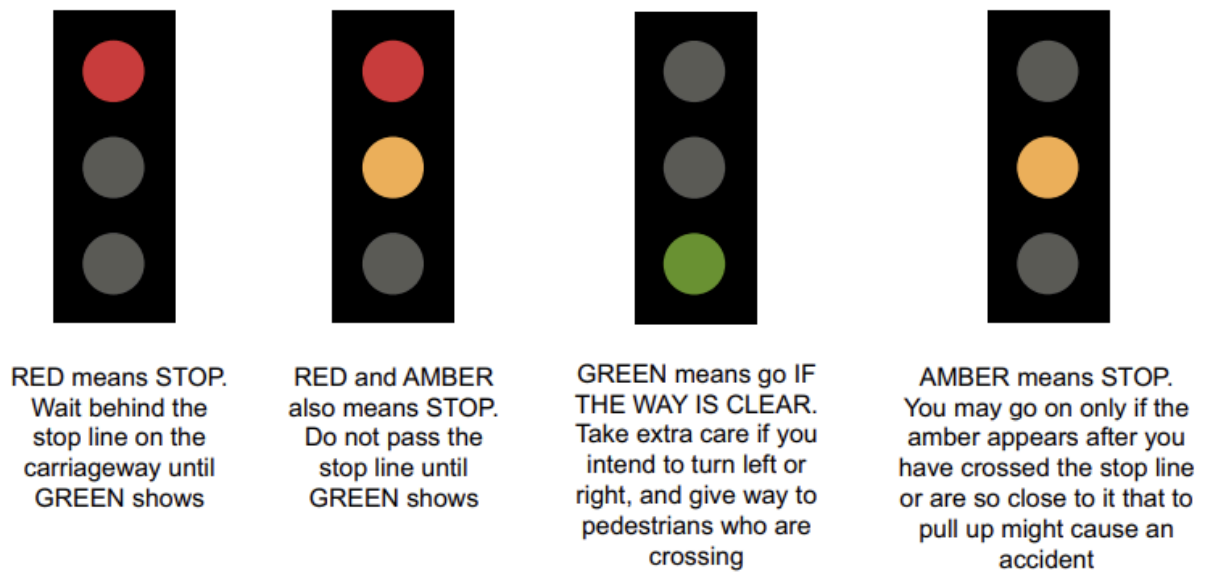


Figure 1: Traffic Light States [2, p35]

Each state has specific timings related to how long they should last. The duration of green and red lights can vary depending on factors such as road geometry, traffic conditions, weather conditions or how many pedestrians there may be. [2, p35]

For the other two states, they have a fixed duration that must be within the given bounds. The amber state must last for 3 ± 0.25 seconds, whereas the red and amber state must last for 2 ± 0.25 seconds. [2, p35]

Additionally, there is an intergreen period, this is the time between the end of one green signal and the start of the next green signal. This is used as a way to allow traffic to clear the junction before traffic should flow again. The duration for the intergreen is a minimum of 5 seconds, but can be extended by external factors such as the road geometry or the pedestrian walking speeds. [2, p38]

Intergreens need to be balanced. Too short of an intergreen can be potentially dangerous, whereas too long of an intergreen can lead to drivers becoming impatient and annoyed. This could lead to drivers ignoring the red light if they are waiting for abnormal or extended periods of time. [2, p38]

Additionally, it is important any sensors used for variable intergreens are fully functionally and non-faulty. If a faulty sensor is used it could lead to too short or too long of an intergreen. [2, p38]

An example of a sensor system for variable intergreens is the Microprocessor Optimised Vehicle Actuation (MOVA) system. This system was developed by the Transport Research Laboratory (TRL). The purpose of this system is to provide optimised and responsive traffic control in place of more traditional, static time based systems. [2, p56]

Circuit Design

This project required 3 traffic lights and six pairs of 7-segment displays (two per light). Each display counted down from 40 to 0. For this project, as few pins as possible were used to create a neat and optimised circuit layout. A complete component list can be seen in **Table 1**:

Table 1: Component List

Component	Amount
PIC16F877A	1
7-segment display	6
Traffic light	3
74HCT4511	2
NOR gate	3
Resistor	1

Lights

Each traffic light had three lights: red, amber and green. This meant 3 pins would be required to program each traffic light. The downside to this is that it would require 9 pins in total, whereas a port on the PIC16F877A only has 8 pins. This would make programming the lights more convoluted in the later stages.

To simplify this, boolean logic was used. As there are only four states the traffic light can be in, this means only two bits need to be used to control the traffic lights (**Table 2**):

Table 2: Required outputs

Bit Values	Output
00	Green
01	Red
10	Amber
11	Red-Amber

To achieve this using only two bits, a NOR gate can be used. A NOR gate allows a high output only when both inputs are low. This achieves the required logic as seen in **Table 2**. The truth table for a NOR gate can be seen in **Table 3**:

Table 3: NOR Gate

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

This is implemented in the circuit using the NOR component found in Proteus. The red and amber values are used as inputs, with the output being used as the input for the green LED on the traffic light. This can be seen in **Figure 2**:

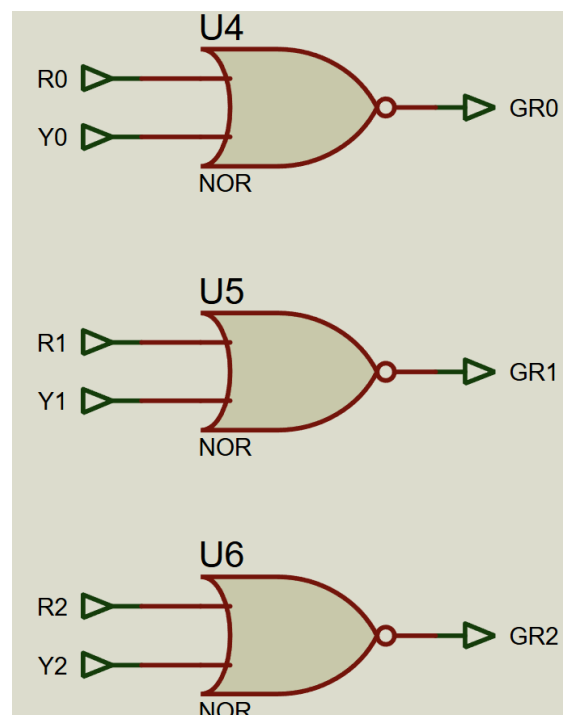


Figure 2: NOR Gates

The final circuit design can be seen in **Figure 3**:

Displays

For each traffic light, two 7-segment displays accompanied it showing a countdown from 40 to 0. For this circuit common cathode 7-segment displays were used. Each display shows the same value, so they can be addressed using the same pins.

As each display uses 7 pins to be programmed, 14 pins would be required to program them. However, this can be optimised by using two 74HCT4511 binary coded decimal (BCD) decoders. These decoders require 5 input pins (4 data, 1 latch enable) and can automatically convert a 4 bit input to the required output for the displays. This reduces the required pins from 14 to 10. [3]

The amount of pins used could still be reduced however. This was done by utilising the latching functionality of the 74HCT4511 which stores the last used value. This means both displays can be programmed with the same data input pins, but the display being changed is chosen based on the latching status. This achieved the desired result and allowed the amount of pins required for the displays to be reduced to 6.

The implementation can be seen in **Figure 4**:

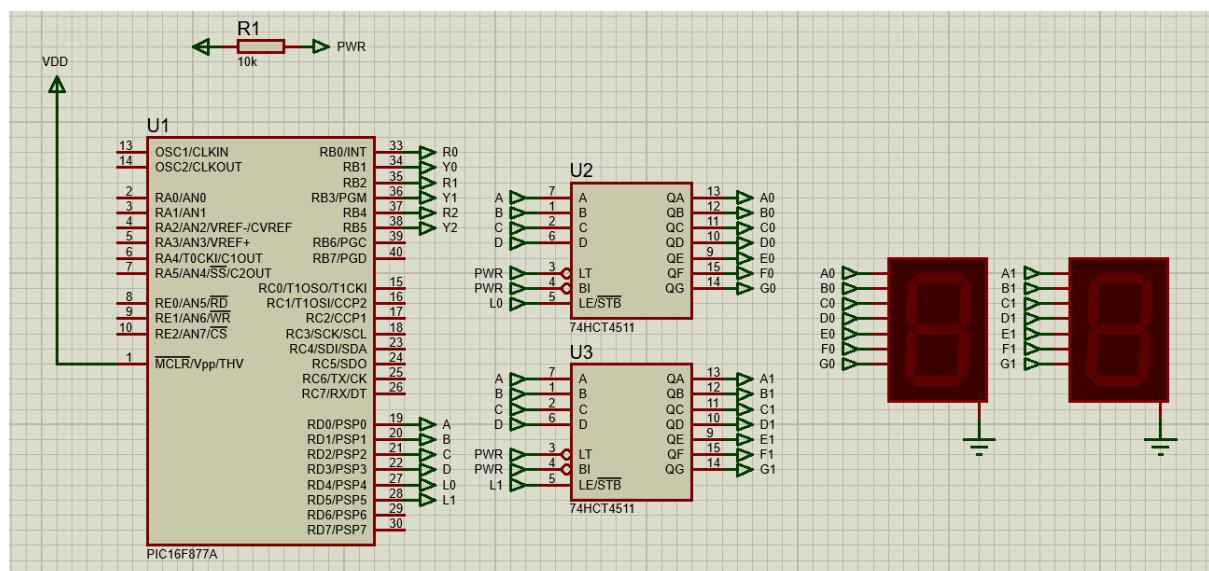


Figure 4: 7-segment Displays

Pseudocode

Before properly implementing the software for the circuit, pseudocode was used to get an idea for how the program should flow. The program was designed to create local variables which store the light stages and duration, the current time to display to the 7-segment displays, the duration left for the light stage and which light stage is currently being used. Before beginning the main program loop, the 7-segment display and traffic lights should display their initial values to ensure proper startup of the system.

During the main loop, the program should continuously update the display and update the duration left on the current light stage. This should occur every second. Additionally, when the light stage duration reaches 0, it should update the traffic light stage to use the next one and reset the light stage duration counter.

Lastly, when the current time on the 7-segment displays reaches 0, it should essentially re-initialise the circuit as was done before entering the main loop. This allows the program to continue indefinitely.

This pseudocode developed can be seen below in **Figure 5**:

```
Light_config[][4] = {
    {Light_1, Light_2, Light_3, Duration},
    {Light_1, Light_2, Light_3, Duration},
    ...
}

current_time = 40
light_duration = 0
current_light = 0

write_to_display(current_time)
update_traffic_lights(light_config[0])

while(true):
    delay_ms(1000)

    current_time--
    write_to_display(current_time)

    light_duration--
    if light_duration == 0:
        update_traffic_lights(++current_light)
```

```
if current_time == 0:  
    current_time = 40  
    current_light = 0  
    write_to_display(current_time)  
    update_traffic_lights(light_config[0])
```

Figure 5: Pseudocode

Software

The software for the project was written in the C programming language and compiled with the CCS C compiler. C was used due to pre-existing knowledge and experience.

Writing To The Displays

Writing to the displays was done by interfacing with the 74HCT4511 BCD decoders. As this is two individual 7-segment displays, one of the displays had to use the latched value when writing to the other. This was achieved by setting the corresponding latch enable pin high.

The values to write to the displays were retrieved from the new, decremented time from the displays. Each display represented either the 10 or 1 part of the number. To get the 10 values a division of 10 was used, for the 1 values a modulo of 10 was used. After getting these values, they were outputted to the port. It was important to match the value with the display so the correct display was using the latched value when outputting on the pins. The function to write to the display was called by a function that also decrements the timer, reducing required code in the main function (**Figure 6**).

```
// Update the 7-segment displays
void update_displays(int value)
{
    // Dividing always removes the decimal, so 3.7 becomes 3
    // Modulo gives the remainder, so 37 % 10 = 7
    // Using this we can get the 10 and 1 values for our countdown
    int tens = value / 10;
    int ones = value % 10;

    // Turn off the corresponding pin to write to the display
    // while keeping the other pin high to use the latched value
    output_d(0b00100000 + tens);
    output_d(0b00010000 + ones);
}

// Update the value on the displays and show it
void update_countdown(int* time)
{
    *time -= 1;
    update_displays(*time);
}
```

Figure 6: Display Code

Traffic Lights Control

Controlling the traffic lights was just a case of outputting the required values to the port depending on what state the lights should be. The value to output was stored in a variable which was manipulated by the *set_light* function, then outputted at the end of the *update_lights* function.

The *set_light* function changed the value for the red and amber light which achieved all 4 states (**Table 2**). This was done by manipulating the bits individually depending on which light was being changed and updating them with the new value.

The code for this can be seen below in **Figure 7**:

```
// Set the light based on the index and state given
void set_light(enum State light_state, int light_index, int8* lights_var)
{
    // The mask is required as otherwise when a bit is set high, it won't be able
    // to become low again with the OR statement
    int mask = 0b11 << 2 * light_index;

    *lights_var = *lights_var & ~mask | light_state << (2 * light_index);
}

// Handle light updates
void update_lights(int* light_states, int8* lights_var, int* light_duration)
{
    set_light(light_states[0], 0, lights_var);
    set_light(light_states[1], 1, lights_var);
    set_light(light_states[2], 2, lights_var);

    *light_duration = light_states[3];

    output_b(*lights_var);
}
```

Figure 7: Traffic Light Code

Entire program

The entire program achieved the same flow as described by the pseudo code (**Figure 5**). The main function allowed the software to keep track of what light signal is currently being used, as well as correctly displaying the time left on the 7-segment displays.

The whole program can be seen below in **Figure 8**:

```
#include "main.h"

// Enum for readability in the 2d array
typedef enum State
{
    GREEN = 0,
    RED,
    AMBER,
    RED_AMBER
};

// Update the 7-segment displays
void update_displays(int value)
{
    // Dividing always removes the decimal, so 3.7 becomes 3
    // Modulo gives the remainder, so 37 % 10 = 7
    // Using this we can get the 10 and 1 values for our countdown
    int tens = value / 10;
    int ones = value % 10;

    // Turn off the corresponding pin to write to the display
    // while keeping the other pin high to use the latched value
    output_d(0b00100000 + tens);
    output_d(0b00010000 + ones);
}

// Update the value on the displays and show it
void update_countdown(int* time)
{
    *time -= 1;
    update_displays(*time);
}

// Set the light based on the index and state given
void set_light(enum State light_state, int light_index, int8* lights_var)
{
    // The mask is required as otherwise when a bit is set high, it won't be able
    // to become low again with the OR statement
    int mask = 0b11 << 2 * light_index;

    *lights_var = *lights_var & ~mask | light_state << (2 * light_index);
}
```

```

// Handle light updates
void update_lights(int* light_states, int8* lights_var, int* light_duration)
{
    set_light(light_states[0], 0, lights_var);
    set_light(light_states[1], 1, lights_var);
    set_light(light_states[2], 2, lights_var);

    *light_duration = light_states[3];

    output_b(*lights_var);
}

int main (void)
{
    // Set all ports to output
    set_tris_a(0);
    set_tris_b(0);
    set_tris_c(0);
    set_tris_d(0);

    // 2d array storing the states of the lights and how long each state should
last
    int lights_order[][4] = {

        // Starting lights
        {GREEN, GREEN, RED, 7},

        // Transition period 1
        {AMBER, AMBER, RED_AMBER, 3},

        // Green for 7 seconds
        {RED, RED, GREEN, 7},

        // Transition period 2
        {RED_AMBER, RED_AMBER, AMBER, 3},

        // Green for 7 seconds
        {GREEN, GREEN, RED, 7},

        // Transition period for pedestrians
        {AMBER, AMBER, RED, 3},

        // All lights red for pedestrian crossing
        {RED, RED, RED, 8},

        // Final transition before looping
        {RED_AMBER, RED_AMBER, RED, 3}

    };

    // Current time for the countdown
    int current_time = 41;

```

```

int current_light = 0;
int light_duration;
int8 lights_var;

// Initial display value
update_countdown(&current_time);
update_lights(lights_order[0], &lights_var, &light_duration);

while (1)
{
    delay_ms(1000);

    light_duration--;

    if(light_duration == 0)
    {
        update_lights(lights_order[++current_light], &lights_var,
&light_duration);
    }

    update_countdown(&current_time);

    if(current_time == -1)
    {
        current_time = 41;
        current_light = 0;

        // Initial display value
        update_countdown(&current_time);
        update_lights(lights_order[0], &lights_var, &light_duration);
    }
}
return 0;
}

```

Figure 8: Entire Program Code

Simulation Results

For this project there were 8 stages for the lights which get looped over as the displays count down from 40 to 0. These stages describe what colours the traffic lights should be showing as well as how long that state should last.

For the sake of keeping the program within the 10 second phase duration restriction, realistic traffic light functionality could not be achieved to a sensible standard. This was due to the lack of time for an intergreen, which would normally cause traffic lights to start changing at different times, allowing any possible extra traffic to exit the junction. However, this period is at minimum 5 seconds which would not allow for a reasonable duration for the next green.

The configurations for each stage were stored in a 2d array composed of the colour to be shown on each light and the duration that state should last for. Additionally, an enumeration was used in place of raw numbers, each value adhered to the truth table in **Table 2**. This made it easier to understand and program each state in the array. The code for this can be seen in **Figure 9**

```
// Enum for readability in the 2d array
typedef enum State
{
    GREEN = 0,
    RED,
    AMBER,
    RED_AMBER
};
...

...
// 2d array storing the states of the lights and how long each state should
// last
int lights_order[][4] = {

    // Starting lights
    {GREEN, GREEN, RED, 7},

    // Transition period 1
    {AMBER, AMBER, RED_AMBER, 3},

    // Green for 7 seconds
    {RED, RED, GREEN, 7},
```

```
// Transition period 2
{RED_AMBER, RED_AMBER, AMBER, 3},

// Green for 7 seconds
{GREEN, GREEN, RED, 7},

// Transition period for pedestrians
{AMBER, AMBER, RED, 3},

// All lights red for pedestrian crossing
{RED, RED, RED, 8},

// Final transition before looping
{RED_AMBER, RED_AMBER, RED, 3}

};
```

Figure 9: Traffic Light Stages Code

Stage 1

Initially, two of the traffic lights are green allowing normal traffic flow while one is red. This state lasts for 7 seconds, then the lights start to change. The stage can be seen in **Figure 10** and **Figure 11**:

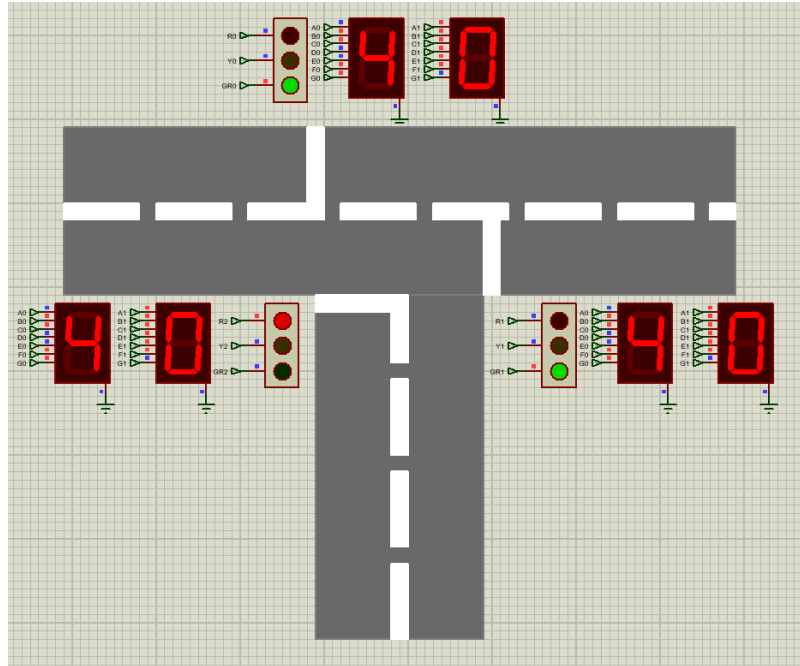


Figure 10: Stage 1-1

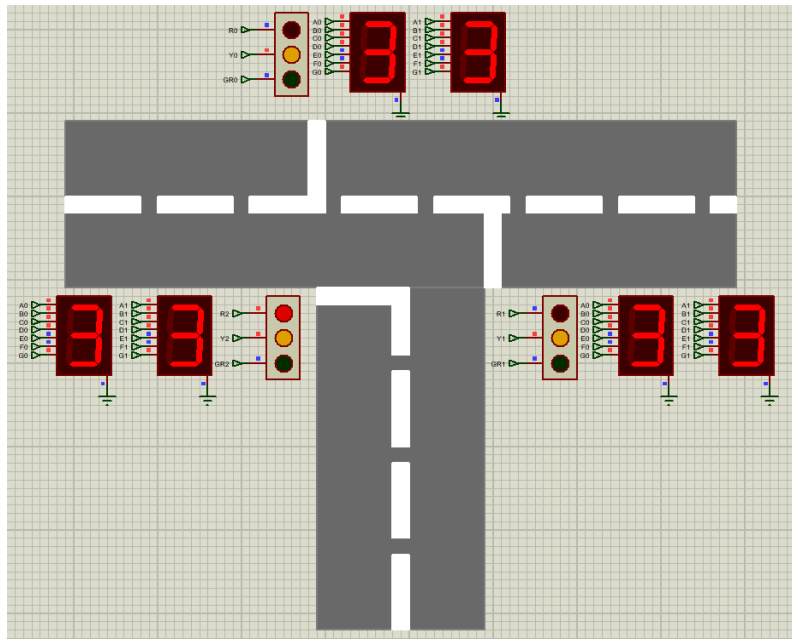


Figure 11: Stage 1-2

Stage 2

This stage occurs after the lights transition from amber or red-amber. In this stage only one light is green and two are red, allowing traffic to exit the junction. Similarly to stage 1, the lights remain in this state for 7 seconds until they begin to transition. **Figure 12** and **Figure 13** show this stage:

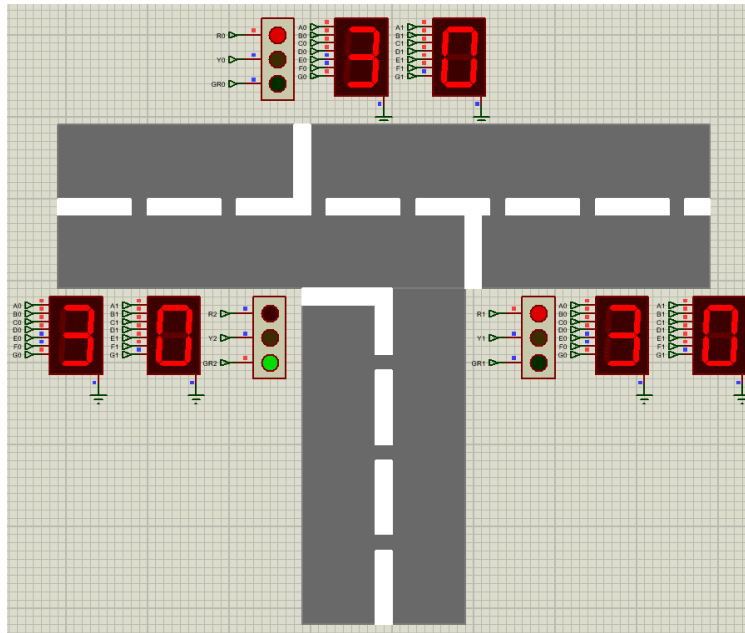


Figure 12: Stage 2-1

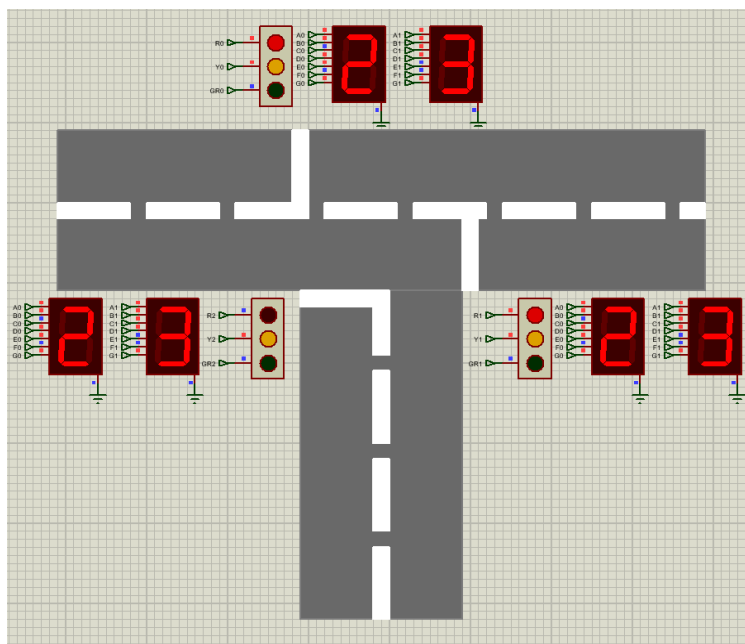


Figure 13: Stage 2-2

Stage 3

Stage 3 is essentially the same as stage 1. The only difference being that when the lights transition, only the two green lights turn to red. This is because the pedestrian crossing stage is directly after this one. **Figure 14** shows this change:

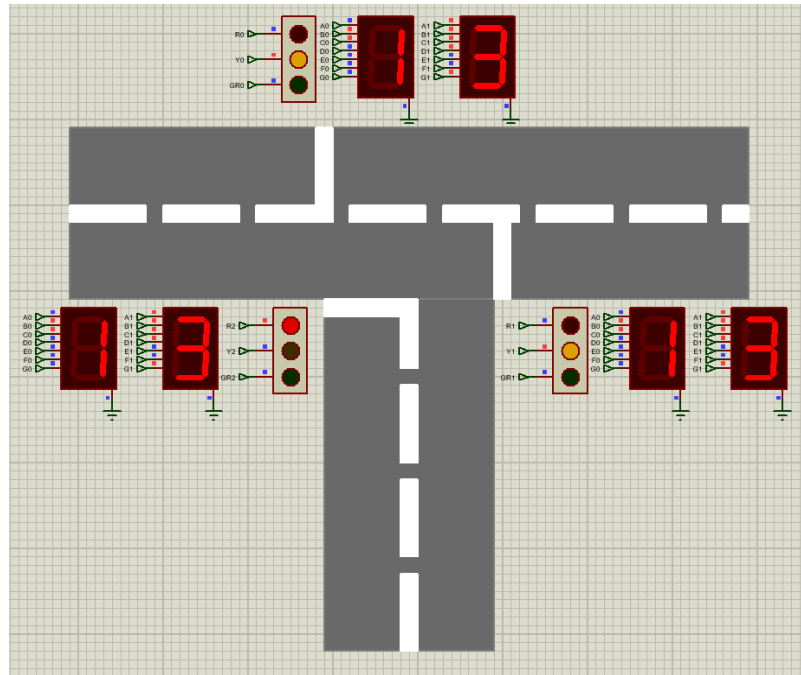


Figure 14: Stage 3-2

Stage 4

This stage is the pedestrian crossing stage. All lights are red during this stage to allow pedestrians to cross safely. This stage is the final stage until looping back to stage 1 and re-doing the cycle. The lights are all red for 8 seconds, then the lights begin to change for 3 seconds, as seen in **Figure 15** and **Figure 16**:

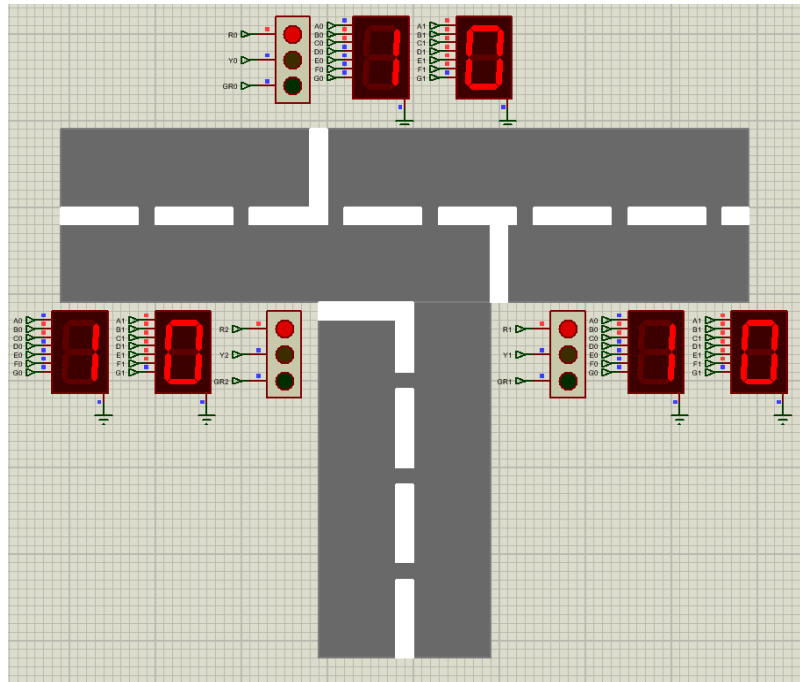


Figure 15: Stage 4-1

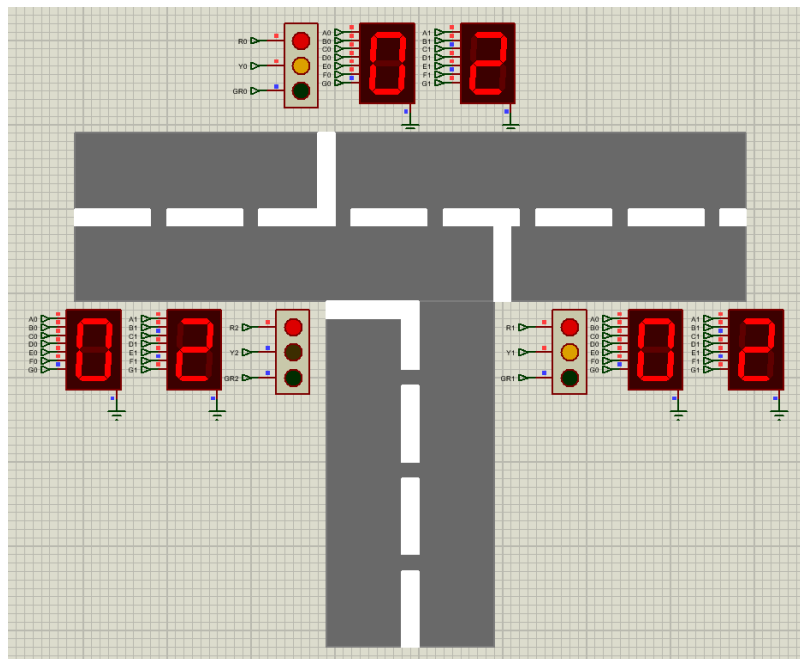


Figure 16: Stage 4-2

Conclusion

The development of this project required some circuit design knowledge and basic software development skills. Knowledge gained by researching traffic lights was unable to be used due to artificial constraints set by the brief. This meant the traffic lights implementation had to be adapted to attain a sensible result.

Future Recommendations

Further development of this project would be improved by creating a more realistic scenario, with more realistic time periods. The UK traffic light guidelines could not be followed in this project as the time constraint imposed meant realistic timings were not sensible to implement. Furthermore, instead of having a dedicated pedestrian crossing phase, a button could be added to allow for dynamic pedestrian crossing phases. This would improve traffic flow as otherwise traffic is brought to a halt when there may not even be any pedestrians to give way to.

Additionally, instead of powering the traffic lights directly from the PIC, a proper power supply could be implemented. This would reduce the power load on the PIC microcontroller allowing for a potentially more stable circuit.

References

- [1] Microchip, "PIC16F877A", [www.microchip.com](https://www.microchip.com/en-us/product/PIC16F877A).
<https://www.microchip.com/en-us/product/PIC16F877A> (accessed April 24, 2025)
- [2] Department for Transport, "Traffic Signs Manual | Chapter 6", [www.gov.uk/government/organisations/department-for-transport](https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/851465/dft-traffic-signs-manual-chapter-6.pdf).
https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/851465/dft-traffic-signs-manual-chapter-6.pdf (accessed April 24, 2025)
- [3] Nexperia, "74HCT4511D", www.nexperia.com.
<https://www.nexperia.com/product/74HCT4511D> (accessed April 24, 2025)