

**EECS 3100, 043**

**Embedded Systems**

Dr. Devinder Kaur

TA: Mehzabien Iqbal

*Lab 7 - Traffic Light Controller*

*Aiden Rader, Alex Munn*

*07/25/2025*

## Table of Contents:

1. Introduction
2. Procedure
3. Main Source Code
4. Screenshots of FSM States
5. Simulator Screenshots
6. Hardware Implementation
7. Team Member Contributions
8. Conclusion

## Introduction:

The purpose of this lab was to design, implement, and test a **Finite State Machine (FSM)** to control traffic intersections using our microcontrollers and Keil simulation. The intersection includes north/southbound and east/westbound traffic with pedestrian crossing support. The lab focused on using real-time embedded control, implementing state transitions based on input sensors and a pedestrian button, and using a linked data structure to define the FSM logic.

Through this lab, we gained experience with hardware-level interfacing, including configuring GPIO pins, LEDs, and switches on the TM4C123 microcontroller. Additionally, we applied software engineering techniques to implement deterministic control without conditional branching, instead relying on a strict state-based structure. Some of the key concepts such as SysTick timer delays, hardware interfacing, and Moore machine design were reinforced during this project.

## Procedure:

The procedure began by selecting appropriate GPIO ports to interface traffic lights and input sensors. The pedestrian "walk" and "don't walk" lights were mapped to PF3 (green) and PF1 (red), respectively. Traffic signals for southbound and westbound directions were connected to either Port A or Port E depending on available hardware.

We designed an FSM with multiple states that account for traffic flow, pedestrian crossing, and wait times. Each state included:

- An 8-bit output to represent light signals
- A time delay using the SysTick\_Wait10ms routine
- Transitions to the next state based on an 8-way input condition (from 3 binary inputs)

After finalizing the FSM design, we implemented the controller in C using a structured typedef with output, delay, and next state pointers. The code was first tested using the TExaS simulator that is included in the code files, capturing simulation output via logic analyzers to validate transitions and timing behavior.

Finally, we built and tested the hardware circuit on the TM4C123 LaunchPad with physical LEDs and switches, debugging input signal responses and verifying the complete traffic and pedestrian cycle.

## Main Source Code:

Below is the entire TableTrafficLight.c file that was given AND edited to work with what this lab had intended us to make.

```
/*  
*****  
* TableTrafficLight.c  
* Instructor: Divinder Kaur  
* Runs on LM4F120/TM4C123  
*/
```

\* Index implementation of a Moore finite state machine to operate a traffic light.

\* Authors: Daniel Valvano,

\*

Jonathan Valvano,

\*

Thomas Royko

\* Student: Aiden Rader, Alex Munn

\* Section: 042

\* Date: 07/17/2025

\*

\* east/west red light connected to PB5

\* east/west yellow light connected to PB4

\* east/west green light connected to PB3

\* north/south facing red light connected to PB2

\* north/south facing yellow light connected to PB1

\* north/south facing green light connected to PB0

\* pedestrian detector connected to PE2 (1=pedestrian present)

\* north/south car detector connected to PE1 (1=car present)

\* east/west car detector connected to PE0 (1=car present)

\* "walk" light connected to PF3 (built-in green LED)

\* "don't walk" light connected to PF1 (built-in red LED)

\*\*\*\*\*/

```
#include "TEaS.h"
```

```
#include "tm4c123gh6pm.h"
```

```
#include <stdint.h>
```

```
#include "SysTick.h"
```

```
// Label input/output ports
```

```
#define LIGHT      (*((volatile uint32_t *)0x400050FC))
```

```
#define SENSOR      (*((volatile uint32_t *)0x4002401C))
```

```
// Define digital input/output ports
```

```
#define GPIO_PORTB_DIR_R      (*((volatile uint32_t *)0x40005400))
```

```
#define GPIO_PORTB_AFSEL_R    (*((volatile uint32_t *)0x40005420))
```

```
#define GPIO_PORTB_DEN_R      (*((volatile uint32_t *)0x4000551C))
```

```
#define GPIO_PORTB_AMSEL_R    (*((volatile uint32_t *)0x40005528))
```

```
#define GPIO_PORTB_PCTL_R     (*((volatile uint32_t *)0x4000552C))
```

```
#define GPIO_PORTE_DIR_R      (*((volatile uint32_t *)0x40024400))
```

```
#define GPIO_PORTE_AFSEL_R    (*((volatile uint32_t *)0x40024420))
```

```
#define GPIO_PORTE_DEN_R      (*((volatile uint32_t *)0x4002451C))
```

```
#define GPIO_PORTE_AMSEL_R    (*((volatile uint32_t *)0x40024528))
```

```
#define GPIO_PORTE_PCTL_R     (*((volatile uint32_t *)0x4002452C))
```

```
#define SYSCTL_RCGCGPIO_R     (*((volatile uint32_t *)0x400FE608))
```

```
#define SYSCTL_PRGPIO_R      (*((volatile uint32_t *)0x400FEA08))
```

```
struct State {
```

```
    uint32_t Out;    // 6-bit output
```

```
    uint32_t Walk_Out; // Lowest 3 bits
```

```
    uint32_t Time;    // 10 ms
```

```
    uint8_t Next[9];
```

```
}; // depends on 2-bit input
```

```
typedef const struct State STyp;
```

```
#define goN    0
```

```
#define waitN   1
```

```
#define goE     2
```

```
#define waitE   3
```

```
#define walk    4
```

```
#define wait_walk 5

// Define FSM as global (LOWERED VALS)
STyp FSM[6]={
{0x21, 0x02, 300,{goN,waitN,goN,waitN,waitN,waitN,waitN}}, // goN
{0x22, 0x02, 50,{goE,goE,goE,goE,walk,walk,walk,walk}}, // WaitN
{0x0C, 0x02, 300,{goE,goE,waitE,waitE,waitE,waitE,waitE,waitE}}, // goE
{0x14, 0x02, 50,{goN,goN,goN,goN,walk,walk,walk,walk}}, // WaitE
{0x24, 0x0E, 150,{wait_walk,wait_walk,wait_walk,wait_walk,wait_walk,wait_walk,wait_walk,wait_walk}}, // Walk
{0x24, 0x0A, 50,{goN,goE,goN,goN,goN,goE,goN,goN}} //Wait_Walk
};

// Local function prototypes
void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void); // Enable interrupts
void Init_GPIO_PortsEB(void);
void PortF_Init(void);

int main(void){
    uint8_t n; // state number
    uint32_t Input;
        EnableInterrupts();

        // Call various initialization functions
    TExaS_Init(SW_PIN_PE210, LED_PIN_PB543210, ScopeOff); // initialize 80 MHz system clock
    SysTick_Init(); // initialize SysTick timer
    Init_GPIO_PortsEB();
    PortF_Init();

    // Establish initial state of traffic lights
    n = goN; // initial state: Green north; Red east

    while(1){
        LIGHT = FSM[n].Out; // set lights to current state's Out value
        GPIO_PORTF_DATA_R = FSM[n].Walk_Out;
        SysTick_Wait10ms(FSM[n].Time); // wait 10 ms * current state's Time value
        Input = SENSOR; // get new input from car detectors
        n = FSM[n].Next[Input]; // transition to next state
    }
}

void Init_GPIO_PortsEB(void){
    volatile uint32_t delay;
    SYSCTL_RCGC2_R |= 0x00000032; // activate clock for Port E
    delay = SYSCTL_RCGC2_R; // allow time for clock to start

    GPIO_PORTB_DIR_R |= 0x3F; // make PB5-0 out
    GPIO_PORTB_AFSEL_R &= ~0x3F; // disable alt funct on PB5-0
    GPIO_PORTB_DEN_R = 0x3F; // enable digital I/O on PB5-0
        // configure PB5-0 as GPIO
    GPIO_PORTB_PCTL_R = (GPIO_PORTB_PCTL_R & 0xFF000000) + 0x00000000;
    GPIO_PORTB_AMSEL_R &= ~0x3F; // disable analog functionality on PB5-0

    GPIO_PORTE_DIR_R &= ~0x07; // make PE2-0 in
    GPIO_PORTE_AFSEL_R &= ~0x07; // disable alt funct on PE2-0
    GPIO_PORTE_DEN_R |= 0x07; // enable digital I/O on PE2-0
}
```

```

// configure PE2-0 as GPIO
GPIO_PORTE_PCTL_R = (GPIO_PORTE_PCTL_R&0xFFFFF000)+0x00000000;
GPIO_PORTE_AMSEL_R &= ~0x07; // disable analog functionality on PE1-0
}

void PortF_Init(void){ // NOT SURE IF THIS IS NEEDED DUE TO TExaS_Init function!
    volatile unsigned long delay;

    GPIO_PORTF_LOCK_R = 0x4C4F434B; // 2) unlock GPIO Port F
    GPIO_PORTF_CR_R = 0x1F; // allow changes to PF4-0

    // only PF0 needs to be unlocked, other bits can't be locked
    GPIO_PORTF_AMSEL_R = 0x00; // 3) disable analog on PF
    GPIO_PORTF_PCTL_R = 0x00000000; // 4) PCTL GPIO on PF4-0
    GPIO_PORTF_DIR_R = 0x0E; // 5) PF4,PF0 in, PF3-1 out (important)
    GPIO_PORTF_AFSEL_R = 0x00; // 6) disable alt funct on PF7-0
    GPIO_PORTF_DEN_R = 0x1F; // 7) enable digital I/O on PF4-0
}

```

## Screenshots of FSM States:

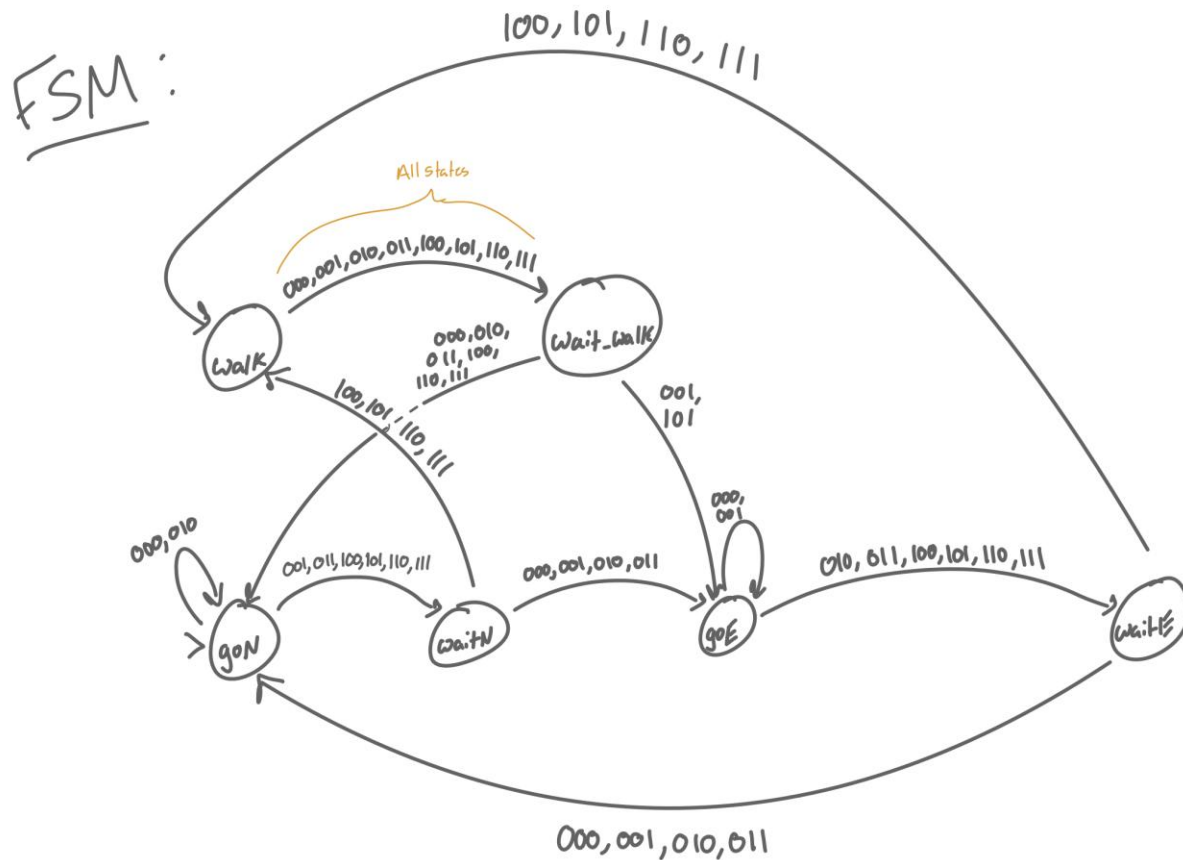
Our traffic light controller was designed using a **Moore finite state machine (FSM)** structure. Each state represents a unique traffic or pedestrian signal combination, with defined outputs, delay times, and eight possible transitions based on 3-bit inputs (South sensor, West sensor, Walk button). Some of the key states where:

State Name	Purpose
goN	Green light for southbound traffic
waitN	Yellow light for southbound traffic
goE	Green light for westbound traffic
waitE	Yellow light for westbound traffic
walk	Walk signal (PF3 ON), all traffic red
wait_walk	"Don't walk" blinking (PF1 blinking), warns pedestrians to finish

Below we can see the FSM **Table** we made before the actual FSM diagram was made, this shows the timing, the binary values, AND the transitions given inputs.

State	Name	Lights	Time	Inputs (In=0 to In=7)							
				In=0	In=1	In=2	In=3	In=4	In=5	In=6	In=7
Start State	goN	10001100	300	goN	waitN	goN	waitN	waitN	waitN	waitN	waitN
	waitN	10001010	50	goE	goE	goE	goE	walk	walk	walk	walk
	goE	00110010	300	goE	goE	waitE	waitE	waitE	waitE	waitE	waitE
	waitE	01010010	50	goN	goN	goN	goN	walk	walk	walk	walk
	walk	10010011	150	waitW	waitW	waitW	waitW	waitW	waitW	waitW	waitW
	wait_walk	10010011	50	goN	goE	goN	goN	goN	goE	goN	goN

After making the table we went on to create the FSM (diagram and in C structs), the diagram you can see below. It transitions between the states previously mentioned based on a 3-bit input representation (i.e. the orange binary above each input).

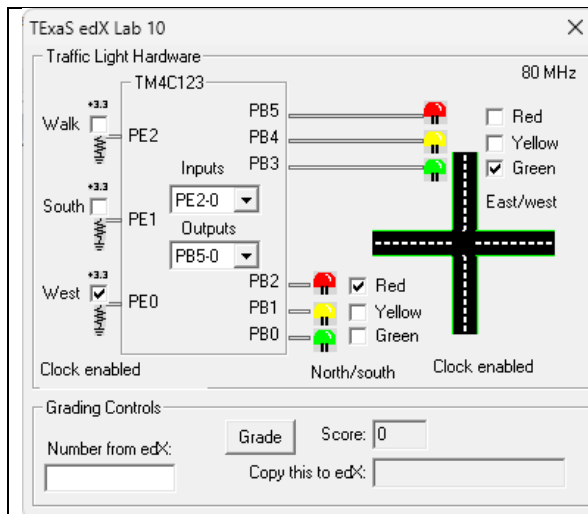


NOTE: I really tried to make this as clean as possible but given our transitions it did come out a little weird, but I hope this makes sense!

## Simulator Screenshots:

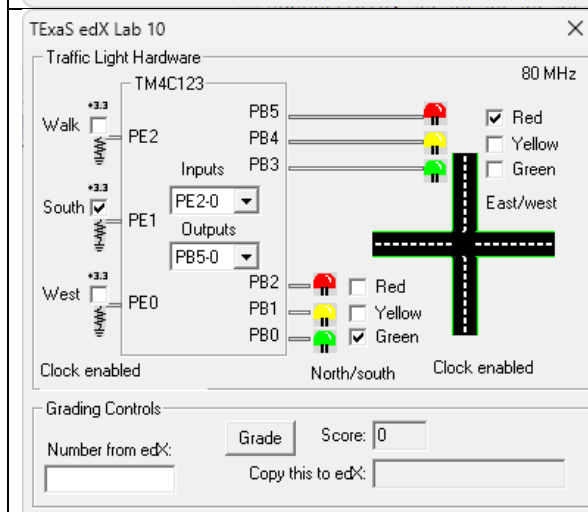
The following images were captured from the TExaS simulation environment and demonstrate key stages of our FSM in action. These snapshots showcase how the system responds to input changes, such as pedestrian button presses and car detections, and how it transitions through different traffic and walk states accordingly.

Photos	Details
--------	---------



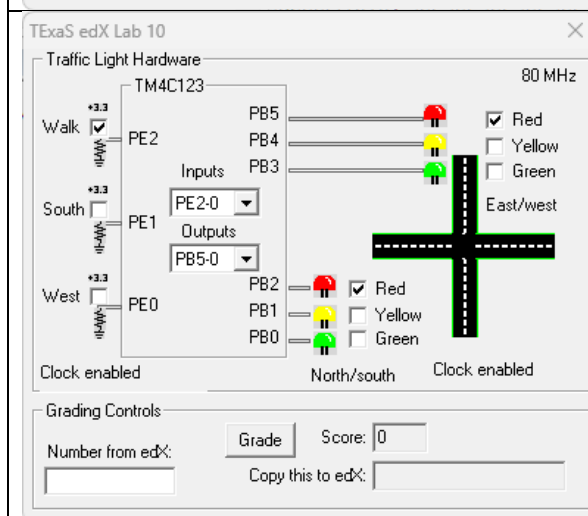
### West Button Pressed:

We can see in this first picture that when West is active then the North/Southbound light turns Red, and the East/Westbound light turns Green.



### South Button Pressed:

We can see in this second picture that when South is active, it is the opposite of when West is active where North/Southbound is Green and East/Westbound is Red.



### Walk Button Pressed:

Now for our last picture, we can see that the Walk button is active. Both North/South and East/West lights are Red. What we do not get to see is the cycling of the North/Southbound light through each color. We give wait delay and cycle to Green, let this wait a bit then cycle back to Red and repeat constantly.



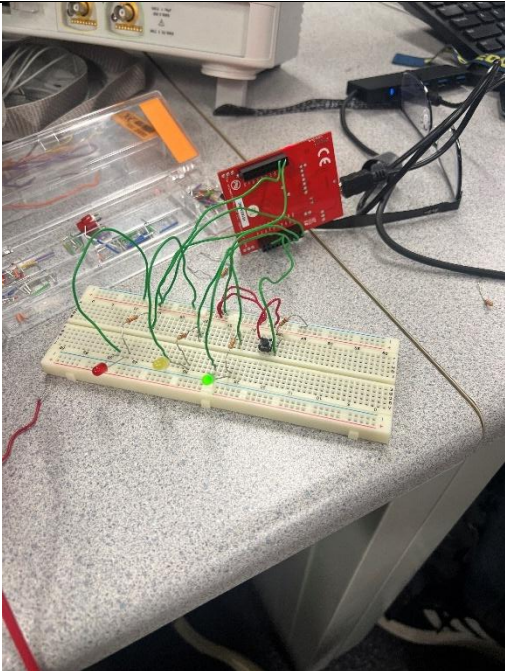
## Hardware Implementation:

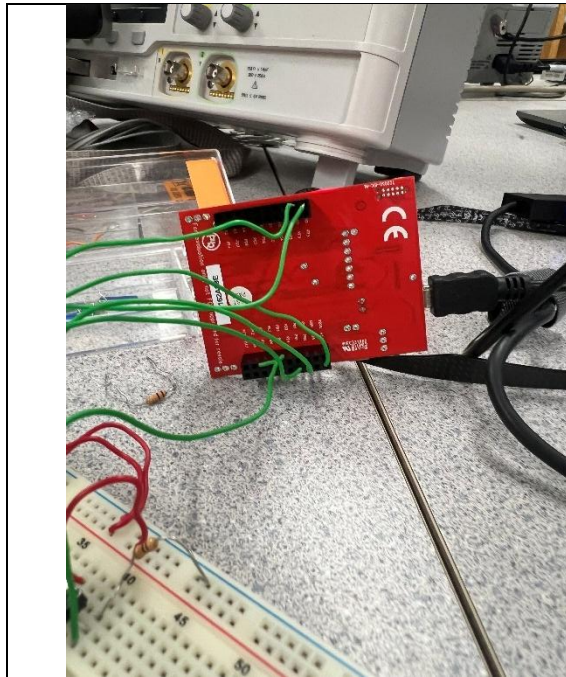
We implemented our system physically using a **TM4C123 LaunchPad**, a **breadboard**, and **external components** like LEDs, resistors, and push-button switches.

### Components (Supposed) to be Used:

- **6 LEDs** for south and west traffic lights (R/Y/G per direction)
- **2 on-board LEDs** for walk (PF3) and don't walk (PF1) signals
- **3 push-buttons** to simulate input sensors (South, West, Walk)

In our hardware test our goal was only to get one of these types of states working, that is in our case turning the traffic light from Green -> Red with the push of the singular button as you can see below in our pictures!

Pictures	Details
	<p>Overall view of the breadboard, this shows our wiring for just doing one button press (that being either South, West, or Walk). The three LEDs are just there to symbolize the Red, Yellow, Green of a traffic light. All of which are controlled primarily through the input of the button and the output to each LED.</p> <p>Either way we wanted to simulate what it would do if we wired together the components and ran this on our actual microcontroller's hardware!</p>



A quick picture showing which ports were being used in the hardware.

**Ports used:**

- PB2-0 were used as outputs to the LEDs and giving them power
- PE2-0 (Whichever we ended up using) controlled whether the lights signal is for Walking, South or West.
- Vcc and GND are of course Voltage and Ground connections, we used a handful of resistors for the 3 LED's and the push button.

Although our **software simulation ran as expected**, we encountered issues when transitioning to real hardware. Specifically, only the green light remained active regardless of the input button presses. We verified all physical connections and circuit components, including resistors, but the system failed to fully cycle through the FSM on hardware.

This could be due to limitations in the LaunchPad setup, pin misconfiguration, or insufficient debouncing or signal stability on the button inputs. Despite these hardware setbacks, we were able to successfully demonstrate the logic and functionality in **simulation mode**.

## Team Contributions:

This project was a collaborative effort between both team members:

- **Alex Munn** led much of the **software development**, including implementing the FSM in C, handling timing logic using the SysTick module, and ensuring the output matched the required light patterns and delays.
- **Aiden Rader (Me)** took the lead on the **hardware implementation**, building the physical circuit on the TM4C123 board, wiring the switches and LEDs, and testing the layout for accurate behavior during real-time demonstrations.
- **Both of us** worked together on **debugging**, simulation testing, and logic analyzer verification. We verified each input combination, resolved timing mismatches, and refined the FSM transitions for accurate and safe operation. We both made changes to the FSM table and logic along the way during this project!

This balanced division of work allowed for a solid grasp of both software structure and hardware control, ensuring a working and reliable traffic control system.

## Conclusion:

This lab provided practical insight into the design and implementation of real-time control systems using FSMs on embedded hardware. We successfully created a traffic light controller that responded to car and pedestrian input, executed walk sequences, and maintained safe operation using a deterministic Moore machine approach.

By avoiding conditional branches and instead structuring our program around a linked FSM, we demonstrated a clean and scalable embedded design. Testing both in simulation and on actual hardware confirmed our system's correctness. We also gained a deeper understanding of embedded software-hardware integration, GPIO control, and the importance of clean state transitions in safety-critical systems like traffic lights.