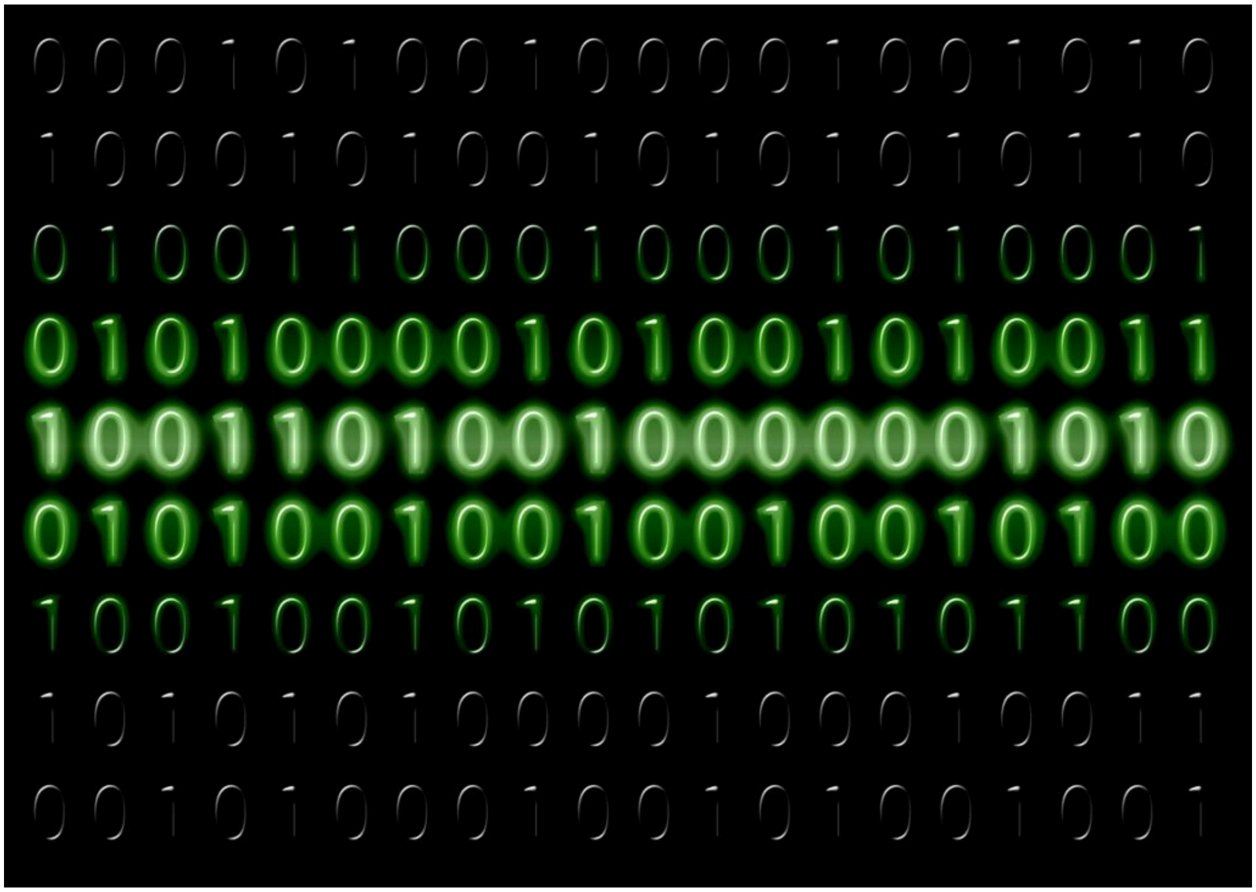
TRAFFIC SIGNAL CONTROLLER

CS2253 - Machine Level Programming | Faculty of Computer Science, UNB



Utsav Upadhyay | Kencho Namgyle

June 2025

Group ~ P

INTRODUCTION

This project was created as part of the **Machine-Level Programming** course. Our goal was to build a basic **traffic signal controller** using **MIPS assembly language**.

The program simulates how traffic lights change in real time and how users can interact with the system by pressing keys. We used MIPS features like loops, conditions, system calls, and memory-mapped I/O (MMIO) to control the timing and handle input.

The project helped us understand how low-level code works and how real-time systems can be built without using high-level tools.

OBJECTIVES

- To create a working simulation of a traffic light system using **MIPS assembly**
- To control light phases using **loops and timers**
- To handle **real-time user input** using **MMIO** (memory-mapped I/O)
- To practice using key MIPS concepts like:
    - **System calls**
    - **Branching and conditions**
    - **Subroutines**
    - **Memory access (load/store)**

This project was aimed at helping us apply what we learned in class to build a simple, real-time system using only low-level instructions.

SYSTEM DESIGN AND ARCHITECTURE

Our program is made up of several parts that work together to simulate a traffic light system.

1) The program starts with a **menu**, where the user can:
    a) Start the simulation
    b) Change green/yellow light duration
    c) Set a speed limit
    d) Exit the program
2) When the simulation starts, the system cycles through four light phases:
    a) North-South Green
    b) North-South Yellow
    c) East-West Green
    d) East-West Yellow

   Each phase uses a **timer loop** to count down seconds and update the display.
3) While the lights are running, the program checks for **keyboard input in real time** using **MMIO**. This lets the user:
    a) Press 'p' to request a pedestrian crossing
    b) Press 'm' to return to the menu
    c) Press 'q' to quit the program
4) If a **pedestrian request** is made, the system adds a special **all-red phase** for crossing.

This structure allows the simulation to run smoothly while staying responsive to user actions.

IMPLEMENTATION DETAILS

We divided the project into two main parts based on our roles:

Utsav (Timing & Light Control)

1) Handled the **main traffic cycle**: green → yellow → next direction
2) Created the **timer loop**, which:
    a) Counts seconds for each light phase
    b) Breaks each second into **10 × 100ms** intervals
3) Added **real-time responsiveness** by checking for input every 100ms

Kencho (MMIO & Input Handling)

1) Implemented **keyboard input** using **memory-mapped I/O**
2) Used addresses:
    a) 0xFFFF0000 to check if a key was pressed
    b) 0xFFFF0004 to read the character
3) Built the **check_keyboard_input** subroutine to handle:
    a) 'p' → pedestrian request
    b) 'm' → return to menu
    c) 'q' → quit

ADDITIONAL COMPONENTS

- Menu system for changing green/yellow times and speed limit
- **Pedestrian phase logic** triggered after yellow lights if requested
- All logic kept modular using MIPS subroutines with jal, jr, and $ra

MIPS CONCEPTS USED

System Calls

Used syscall for:

- Printing text ($v0 = 4)
- Reading integers ($v0 = 5)
- Sleeping ($v0 = 32)
- Exiting the program ($v0 = 10)

Branching and Conditions

Used instructions like beq, bnez, and blez to control program flow

Managed decision-making for menu choices, input handling, and timer countdowns

Loops

Created loops for:

- Countdown timers for each light phase
- Dividing each second into 10 intervals (inner loop for 100ms delays)

Subroutines

Used jal to call procedures like *display_state_with_timer* and *check_keyboard_input*

Managed return flow with jr $ra.

Memory Access

Used lw and sw to store and load:

- Timing values (green/yellow time)
- Speed limit
- Pedestrian request flag

Memory-Mapped I/O

Accessed keyboard input directly using MMIO addresses:

- 0xFFFF0000 for receiver control
- 0xFFFF0004 for receiver data

CHALLENGES

1. **Handling Real-Time Input Without Interrupts**

Challenge: MIPS (in MARS) doesn’t support hardware interrupts

Solution: We used a **polling method** that checks for input every 100ms inside a loop

2. **Avoiding Input Lag**

Challenge: Sleeping for 1 second made the program unresponsive to key presses

Solution: We broke each second into **10 smaller delays** (100ms each), checking for input between them

3. **Clean Code Structure in Assembly**

Challenge: MIPS has no functions or high-level structure

Solution: We used **subroutines** (jal, jr) and saved return addresses properly to keep code modular

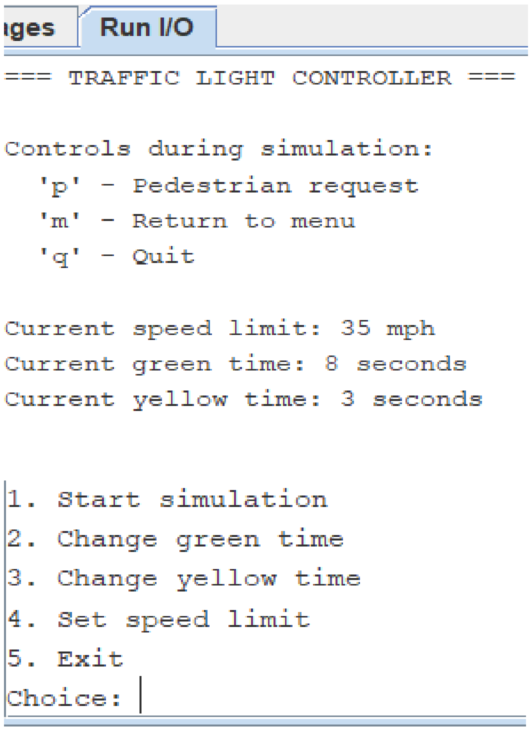
4. **Calculating Yellow Time Dynamically**

Challenge: Yellow light duration had to change based on speed

Solution: We used basic arithmetic to compute: yellow_time = 3 + (speed - 25) / 10

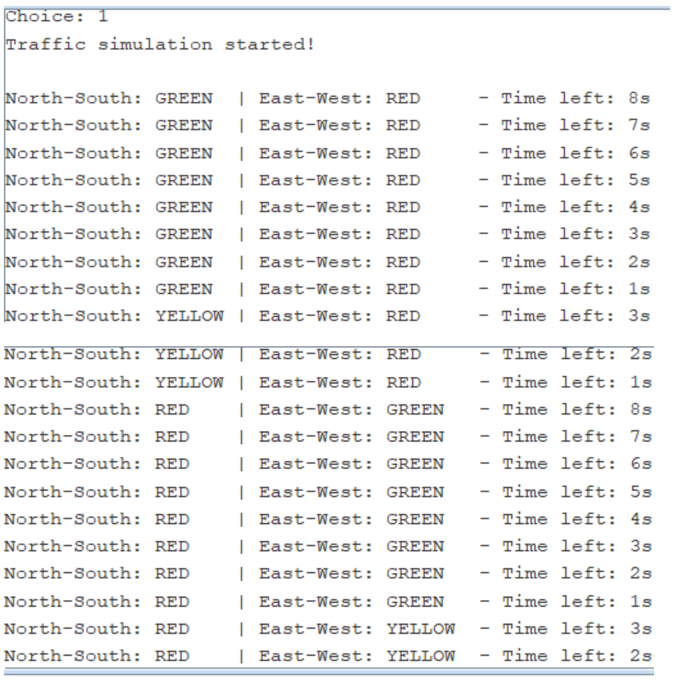
OUTPUT AND SCREENSHOTS

1. Menu Screen



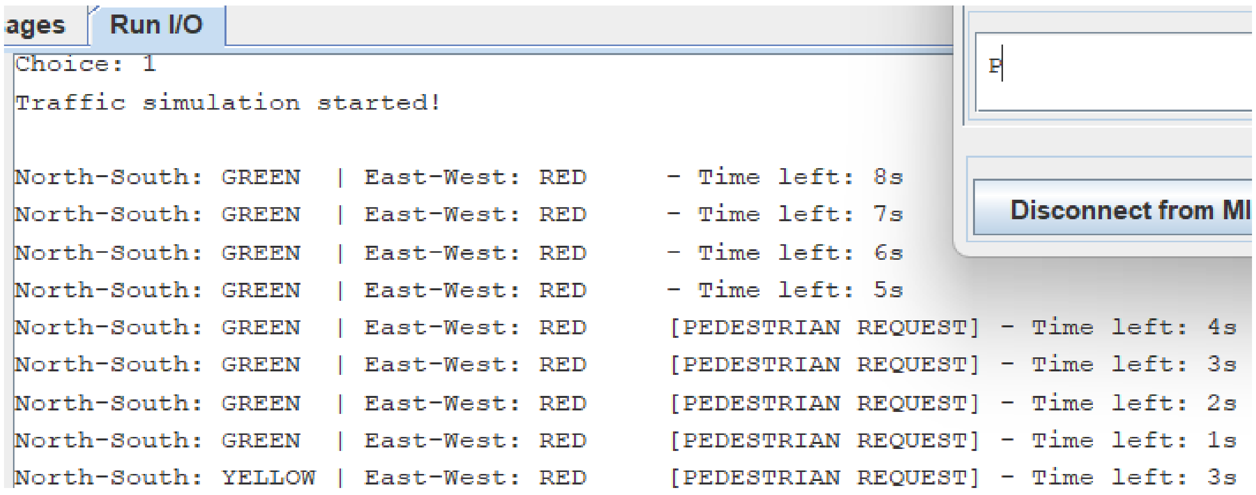
```
=== TRAFFIC LIGHT CONTROLLER ===

Controls during simulation:
  'p' - Pedestrian request
  'm' - Return to menu
  'q' - Quit

Current speed limit: 35 mph
Current green time: 8 seconds
Current yellow time: 3 seconds


1. Start simulation
2. Change green time
3. Change yellow time
4. Set speed limit
5. Exit
Choice:
```

2. Simulation Running

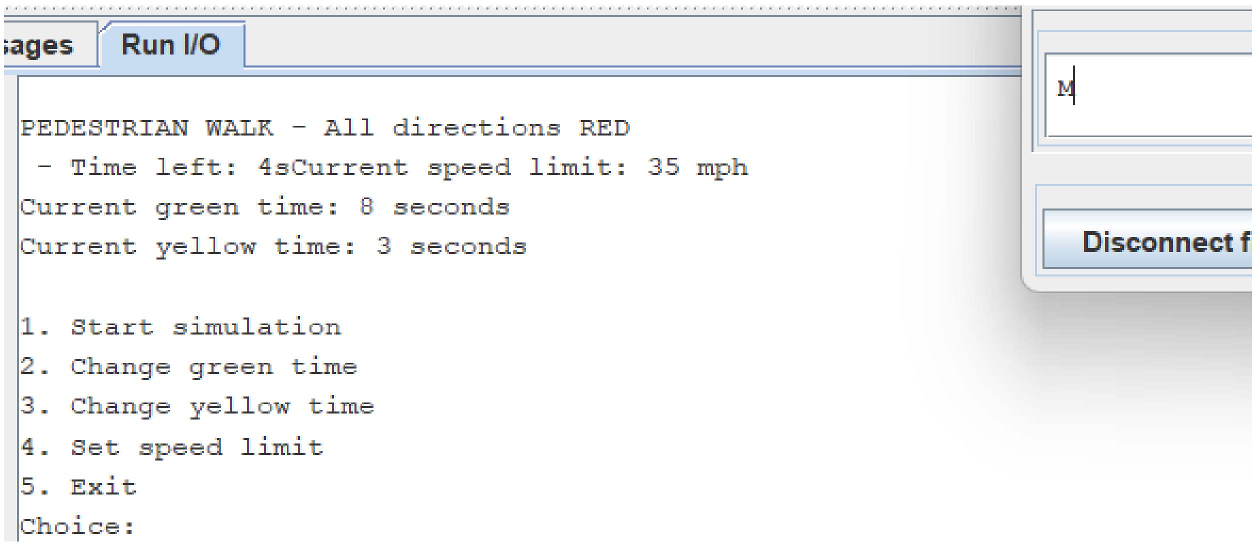


```
Choice: 1
Traffic simulation started!

North-South: GREEN  | East-West: RED     - Time left: 8s
North-South: GREEN  | East-West: RED     - Time left: 7s
North-South: GREEN  | East-West: RED     - Time left: 6s
North-South: GREEN  | East-West: RED     - Time left: 5s
North-South: GREEN  | East-West: RED     - Time left: 4s
North-South: GREEN  | East-West: RED     - Time left: 3s
North-South: GREEN  | East-West: RED     - Time left: 2s
North-South: GREEN  | East-West: RED     - Time left: 1s
North-South: YELLOW | East-West: RED     - Time left: 3s
North-South: YELLOW | East-West: RED     - Time left: 2s
North-South: YELLOW | East-West: RED     - Time left: 1s
North-South: RED    | East-West: GREEN   - Time left: 8s
North-South: RED    | East-West: GREEN   - Time left: 7s
North-South: RED    | East-West: GREEN   - Time left: 6s
North-South: RED    | East-West: GREEN   - Time left: 5s
North-South: RED    | East-West: GREEN   - Time left: 4s
North-South: RED    | East-West: GREEN   - Time left: 3s
North-South: RED    | East-West: GREEN   - Time left: 2s
North-South: RED    | East-West: GREEN   - Time left: 1s
North-South: RED    | East-West: YELLOW  - Time left: 3s
North-South: RED    | East-West: YELLOW  - Time left: 2s
```

3. User Input During Simulation

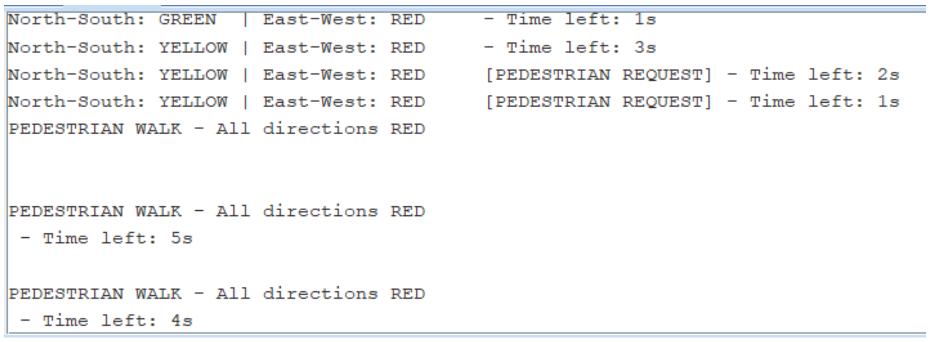


```
Choice: 1
Traffic simulation started!

North-South: GREEN  | East-West: RED     - Time left: 8s
North-South: GREEN  | East-West: RED     - Time left: 7s
North-South: GREEN  | East-West: RED     - Time left: 6s
North-South: GREEN  | East-West: RED     - Time left: 5s
North-South: GREEN  | East-West: RED     [PEDESTRIAN REQUEST] - Time left: 4s
North-South: GREEN  | East-West: RED     [PEDESTRIAN REQUEST] - Time left: 3s
North-South: GREEN  | East-West: RED     [PEDESTRIAN REQUEST] - Time left: 2s
North-South: GREEN  | East-West: RED     [PEDESTRIAN REQUEST] - Time left: 1s
North-South: YELLOW | East-West: RED     [PEDESTRIAN REQUEST] - Time left: 3s
```



```
PEDESTRIAN WALK - All directions RED
 - Time left: 4sCurrent speed limit: 35 mph
Current green time: 8 seconds
Current yellow time: 3 seconds

1. Start simulation
2. Change green time
3. Change yellow time
4. Set speed limit
5. Exit
Choice:
```

4. Pedestrian Phase



```
North-South: GREEN  | East-West: RED     - Time left: 1s
North-South: YELLOW | East-West: RED     - Time left: 3s
North-South: YELLOW | East-West: RED     [PEDESTRIAN REQUEST] - Time left: 2s
North-South: YELLOW | East-West: RED     [PEDESTRIAN REQUEST] - Time left: 1s
PEDESTRIAN WALK - All directions RED


PEDESTRIAN WALK - All directions RED
 - Time left: 5s

PEDESTRIAN WALK - All directions RED
 - Time left: 4s
```

CONCLUSION

This project helped us understand how to build a real-time system using **low-level MIPS assembly code**. We simulated a working traffic light controller that responds to user input, manages time-based light phases, and includes features like pedestrian requests and customizable settings.

We applied key MIPS concepts such as **loops**, **branching**, **system calls**, **subroutines**, and **memory-mapped I/O**. By dividing the responsibilities between timing logic and input handling, we kept the program modular and easy to manage.

Overall, this project gave us practical experience in low-level programming and showed how systems can be built from the ground up without using high-level tools.