Traffic Light Control System
MIPS Implementation Report

2021050300 Computer Science
Jaemin Kim

1. Project Overview and Purpose

The goal of this project is to simulate a real-world intersection traffic light control system using MIPS assembly language. Since assembly operates at the hardware level, this project directly handles the fundamentals of system operations, such as memory access, register control, conditional branching, loops, and function calls. By applying the core syntax and logic of assembly to a “state-based system,” the project aims to provide hands-on experience of how control flow, state transitions, timing control, and output interact in practice.

The main requirement was to implement state transition logic based on the MIPS architecture. The traffic lights transition in sequence from Green → Yellow → Red, and the north-south and east-west directions are designed to operate exclusively. A precise time delay function, using loops, simulates the duration of each state (e.g., green light for 5 seconds, yellow light for 2 seconds). The current state is visually displayed on the console in the form [NS: GREEN | EW: RED] using syscalls for real-time feedback.

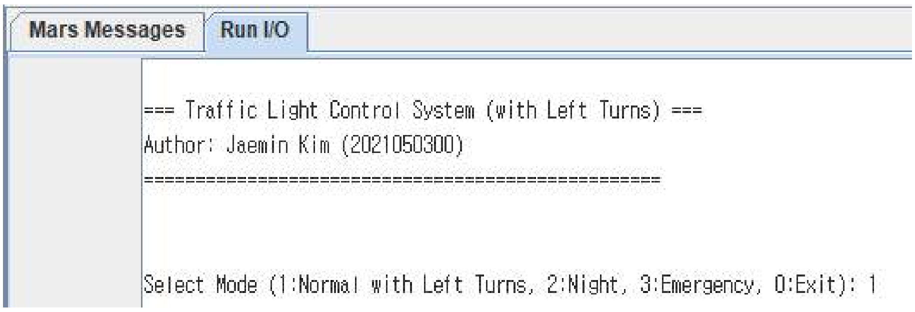
2. Code Structure and Operating Principle

The project consists of three main parts: system initialization, user mode selection, and execution of the selected mode (normal, night, emergency).

2.1. Overall System Flow

- System Initialization (initialize_system):
  Initializes necessary registers (e.g., $s2: total signal state changes) and variables (total_seconds: cumulative simulation time) to 0.

- User Mode Selection (get_user_mode):



Guides the user through the console prompt to select one of four modes (normal mode with left turn, night flashing mode, emergency mode, system exit). User input is received via syscall 5, with error handling for invalid input.

- Traffic Light Operation by Mode:

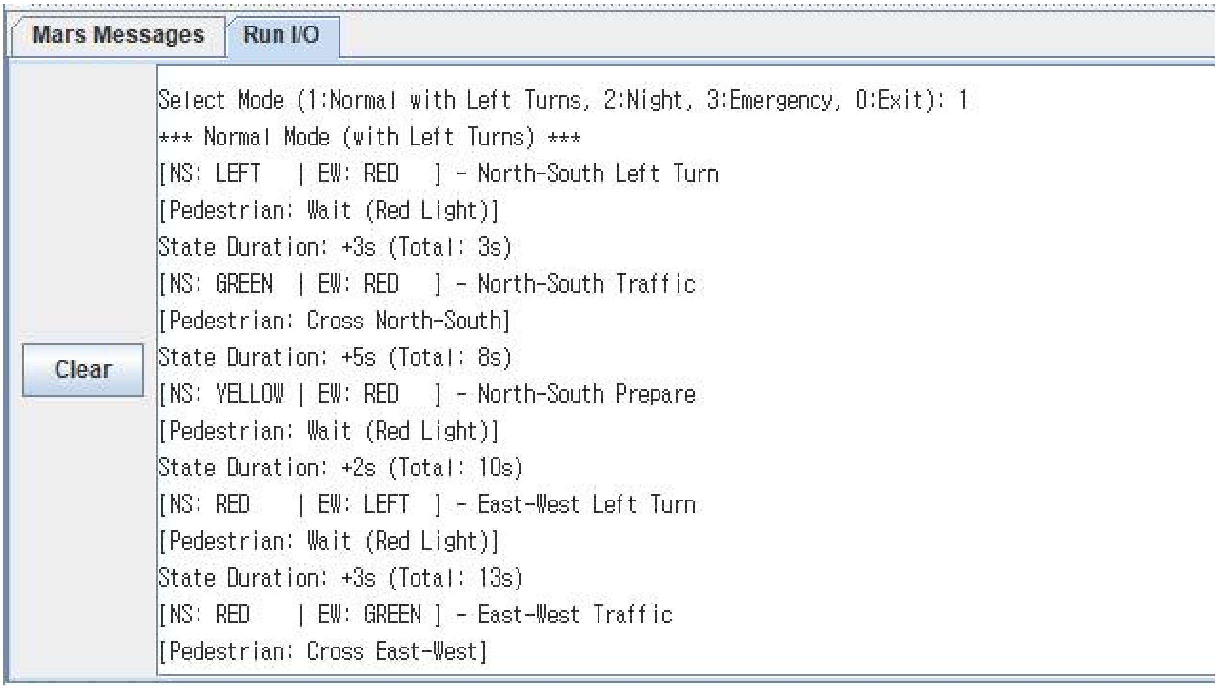
  Each mode is implemented as a separate function (execute_normal_mode, execute_night_mode, execute_emergency_mode), and the selected function is executed based on user input.

- Program Termination and Statistics Output:

  When the user selects "0" or the repetition in a mode ends, the system outputs the total number of signal state changes and the accumulated runtime through print_statistics. Then, the program ends with a print_goodbye message.

2.2. Detailed Operation by Mode

(1) Normal Mode (execute_normal_mode) – Includes Left Turn Signals

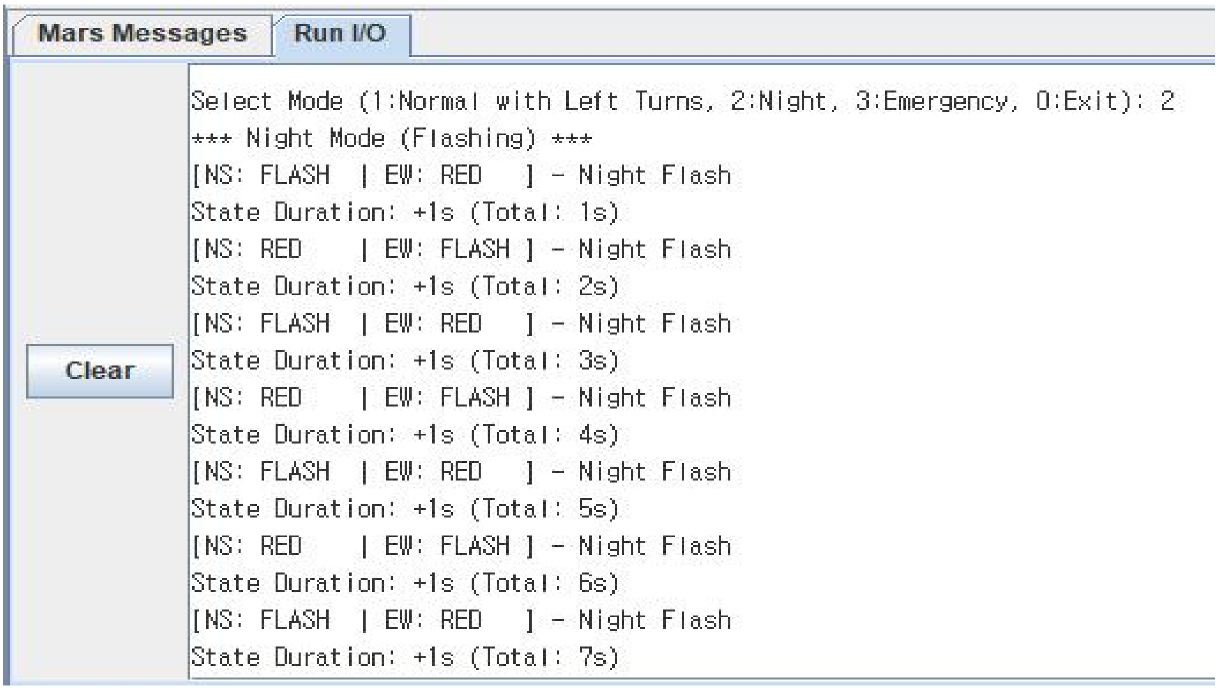


Simulates six signal states, repeating the cycle three times (total 18 states), reflecting a real intersection.

Step	Vehicle Signal	Pedestrian Signal	Duration
1	NS Left, EW Red	All wait	3 sec
2	NS Go, EW Red	NS Cross	5 sec
3	NS Yellow, EW Red	All wait	2 sec
4	EW Left, NS Red	All wait	3 sec
5	EW Go, NS Red	EW Cross	5 sec
6	EW Yellow, NS Red	All wait	2 sec

Each state entry prints a signal and pedestrian message to the console. The print_time_info function outputs the current state's duration and the system's cumulative elapsed time. The update_cycle_counter function updates the count of signal changes and total time, and delay_ms simulates actual delay.

(2) Night Flashing Mode (execute_night_mode)



Simulates a quiet intersection at night, with NS and EW signals flashing alternately for 1 second, repeating 10 times. Each flash uses print_night_flash_ns or print_night_flash_ew. Time and count are managed by print_time_info and update_cycle_counter.

(3) Emergency Mode (execute_emergency_mode)

```
Mars Messages | Run I/O

==================================================


Select Mode (1:Normal with Left Turns, 2:Night, 3:Emergency, 0:Exit): 3
*** Emergency Mode ***
[NS: RED    | EW: RED   ] - Emergency Situation
State Duration: +5s (Total: 5s)
[NS: RED    | EW: RED   ] - Emergency Situation
State Duration: +5s (Total: 10s)
[NS: RED    | EW: RED   ] - Emergency Situation
State Duration: +5s (Total: 15s)
[NS: RED    | EW: RED   ] - Emergency Situation
State Duration: +5s (Total: 20s)
[NS: RED    | EW: RED   ] - Emergency Situation
State Duration: +5s (Total: 25s)
```

Simulates an emergency scenario (such as for passing emergency vehicles), maintaining all signals red for 5 seconds, repeated 5 times.

2.3. Main Functions and Their Purposes

- initialize_system:

```
# System Initialization Function
initialize_system:
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    li $s0, 0         # Current State (less used now, sequence is fixed in normal mode)
    li $s1, 0         # (Unused)
    li $s2, 0         # Total signal states processed
    li $s3, 1         # System mode (default Normal)
    li $t0, 0         # Local cycle counter (for full sequences)
    sw $zero, total_seconds # Clears total_seconds
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

Ensures stable program start by initializing global variables and registers.

- get_user_mode:

```
# User Mode Selection Function
get_user_mode:
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    li $v0, 4
    la $a0, mode_prompt
    syscall
    li $v0, 5
    syscall
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

Decides the system mode based on user input, with exception handling for invalid input.

- Signal Output Functions (print_...):

```
normal_full_cycle_loop: # Renamed for clarity
    # Phase 1: NS_LEFT, EW_RED (3s)
    jal print_ns_left_ew_red
    jal print_pedestrian_wait       # Pedestrians wait
    lw $a2, total_seconds
    li $a1, 3
    jal print_time_info
    li $a1, 3
    jal update_cycle_counter
    lw $a0, LEFT_TURN_TIME
    li $a1, 3
    jal delay_ms

    # Phase 2: NS_GREEN, EW_RED (5s)
    jal print_ns_green_ew_red
    jal print_pedestrian_ns         # NS Pedestrians cross
    lw $a2, total_seconds
    li $a1, 5
    jal print_time_info
    li $a1, 5
    jal update_cycle_counter
    lw $a0, GREEN_TIME
    li $a1, 5
    jal delay_ms

    # Phase 3: NS_YELLOW, EW_RED (2s)
    jal print_ns_yellow_ew_red
    jal print_pedestrian_wait       # Pedestrians wait
    lw $a2, total_seconds
    li $a1, 2
    jal print_time_info
    li $a1, 2
    jal update_cycle_counter
    lw $a0, YELLOW_TIME
    li $a1, 2
    jal delay_ms
```

Each signal state, pedestrian guide, and system message is printed to the console by a dedicated function for better readability and maintainability.

- update_cycle_counter:

```
# Cycle Counter Update (and accumulate seconds)
# $a1: 이번 상태의 초 (seconds for current state)
update_cycle_counter:
    addi $a2, $a2, 1            # Increment total states processed count
    lw $t1, total_seconds
    add $t1, $t1, $a1           # Add current state's seconds to total_seconds
    sw $t1, total_seconds
    jr $ra
```

Updates the total state change count and cumulative time on every signal state change.

- print_time_info:

```
# Time Info Print Function: per-state and cumulative seconds
# $a1: 이번 상태의 초(초단위, 예: 5 또는 2)
# $a2: 진입 전 누적 초 (total_seconds 값)
print_time_info:
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    # $a1 = current state duration in seconds
    # $a2 = total seconds before this state

    li $v0, 4
    la $a0, elapsed_msg     # "State Duration: +"
    syscall

    move $a0, $a1           # X (이번 상태 초)
    li $v0, 1
    syscall

    li $v0, 4
    la $a0, s_msg           # "s (Total: "
    syscall

    addu $t2, $a1, $a2      # $t2 = 이전 누적 + 이번 초 (새로운 총 누적 시간)
    move $a0, $t2
    li $v0, 1
    syscall

    li $v0, 4
    la $a0, close_msg       # "s)\n"
    syscall

    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

Displays the duration and cumulative time for each signal state.

- delay_ms:

```
# Delay Function (in milliseconds)
# $a0: milliseconds to delay
delay_ms:
    # $a0 already contains milliseconds
    li $v0, 32      # sleep syscall
    syscall
    jr $ra
```

Uses syscall 32 to implement actual timing delay, crucial for realistic simulation.

- ask_continue:

```
# Continue Check
ask_continue:
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    li $v0, 4
    la $a0, continue_prompt
    syscall
    li $v0, 5              # Read integer
    syscall                # $v0 contains the user's input (1 or 0)
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra

```

Prompts the user whether to repeat the simulation.

- print_statistics:

```
# Statistics Print
print_statistics:
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    li $v0, 4
    la $a0, divider
    syscall

    li $v0, 4
    la $a0, total_cycles   # "Total Signal States Processed: "
    syscall
    li $v0, 1
    move $a0, $s2          # Print $s2 (total states)
    syscall
    li $v0, 4
    la $a0, cycles_msg     # " states₩n"
    syscall

    li $v0, 4
    la $a0, runtime_msg    # "Total Runtime: "
    syscall
    lw $t1, total_seconds
    move $a0, $t1          # Print total accumulated seconds
    li $v0, 1
    syscall
    li $v0, 4
    la $a0, seconds_msg    # "sec₩n"
    syscall

    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
```

Outputs a summary of total signal changes and elapsed time before program termination.

3. Reflections and Learning Outcomes

Through this MIPS assembly traffic light control system project, I was able to gain a deeper understanding in several areas:

- Essence of Hardware Control:

By dealing directly with registers, memory access, and system calls, I was able to understand how computers actually work beyond high-level abstractions. Especially, implementing time delays using syscall 32 gave me insight into how software interacts with hardware timers.

- Complexity of State-Based Systems:

Traffic light control is more than just changing colors—it also requires consideration for vehicle and pedestrian flow, left turns, and special scenarios. Implementing these in assembly made me realize the importance of designing logical connections and transitions between states.

- Importance of Modularity and Function Design:

By manually implementing function calls and stack management, I realized that modularity is essential for code reusability and maintainability. Separating output functions by signal state significantly improved readability and manageability.

- Challenges of Accurate Timing Control:

While simulation delays may not perfectly match real hardware timing, directly managing state times through total_seconds and delay_ms highlighted the core aspects of real-time system design.