

신호등 제어 시스템

MIPS 구현 보고서

2021050300 컴퓨터학부
김재민

1. 프로젝트 개요 및 목적

본 프로젝트는 MIPS 어셈블리어를 활용하여 실제 교차로의 신호등 제어 시스템을 시뮬레이션하는 것을 목표로 한다. 어셈블리어는 하드웨어 수준에서 동작하기 때문에, 메모리 접근, 레지스터 제어, 조건 분기, 반복문, 함수 호출 등 시스템의 근본적인 작동 방식을 직접 다루게 된다. 이 과제를 통해 어셈블리어의 핵심 문법과 논리를 실제 “상태 기반 시스템”에 적용함으로써 제어 흐름, 상태 전이, 시간 제어, 출력 등 다양한 요소가 어떻게 상호작용하는지를 체험하고자 한다.

이를 위해 MIPS 아키텍처 기반의 상태 전이 로직 구현이 핵심적으로 요구되었다. 신호등은 초록(Green) → 노랑(Yellow) → 빨강(Red)의 순차적 상태 전환을 기반으로 하며, 동서방향과 남북방향이 상호 배타적으로 작동하는 구조로 설계되었다. 반복 루프를 활용한 정밀한 시간 지연 기능이 구현되어 각 상태(예: 녹색등 5초, 황색등 2초)의 지속 시간을 모사하도록 하였으며, syscall을 통한 콘솔 기반 출력 시스템은 현재 신호 상태를 [NS: GREEN | EW: RED] 형태로 실시간 표시하여 시각적 피드백을 제공한다.

2. 코드 구조 및 작동 원리

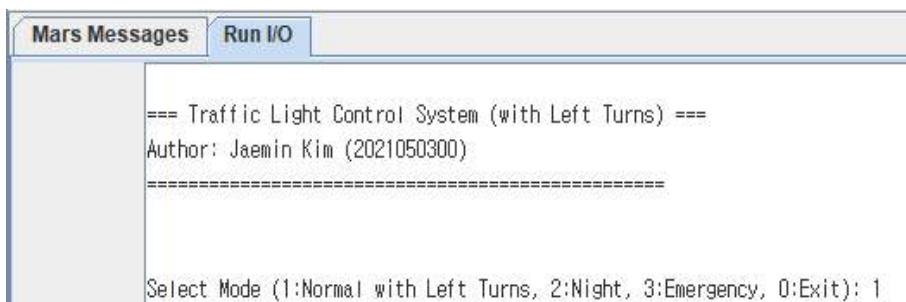
프로젝트는 크게 시스템 초기화, 사용자 모드 선택, 그리고 선택된 모드(일반, 야간, 긴급)의 실행 부분으로 구성된다.

2.1. 전체 시스템 흐름

- 시스템 초기화 (initialize_system):

시스템 구동에 필요한 레지스터(\$s2: 총 신호 상태 변경 횟수)와 변수(total_seconds: 총 시뮬레이션 누적 시간)를 0으로 초기화한다.

- 사용자 모드 선택 (get_user_mode):



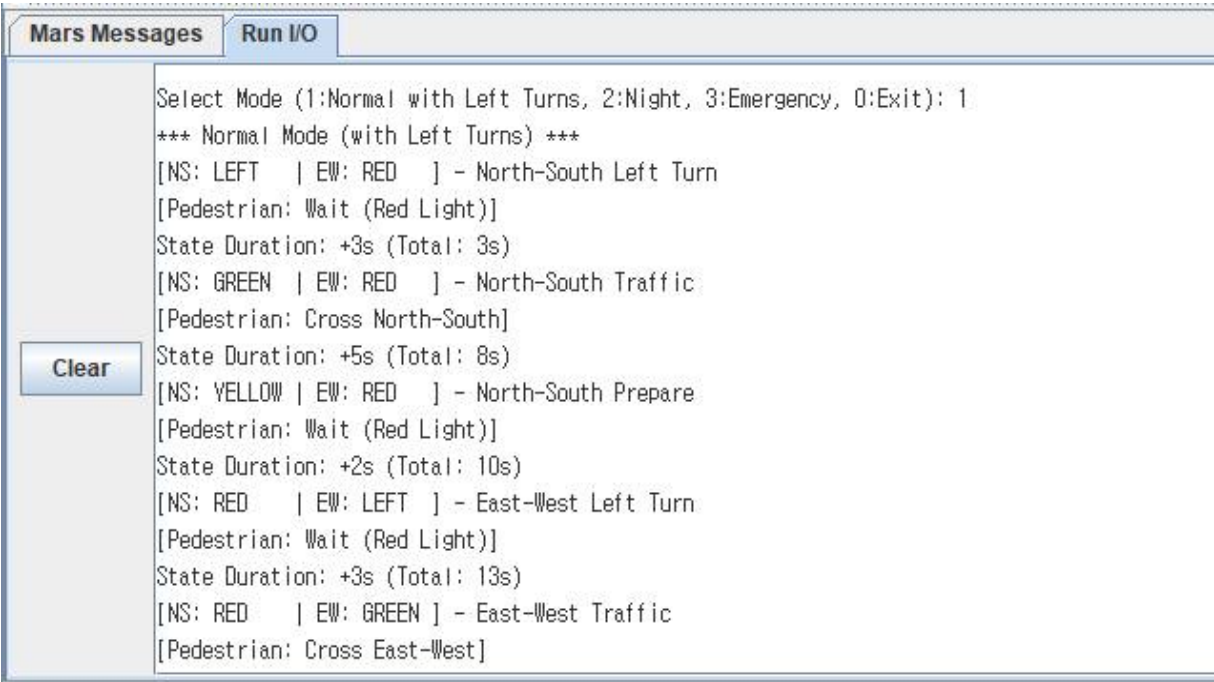
```
Mars Messages Run I/O
=== Traffic Light Control System (with Left Turns) ===
Author: Jaemin Kim (2021050300)
=====
Select Mode (1:Normal with Left Turns, 2:Night, 3:Emergency, 0:Exit): 1
```

콘솔 프롬프트를 통해 사용자에게 다음 4가지 모드 중 하나를 선택하도록 안내한다. (일반 모드 - 좌회전 포함, 야간 점멸 모드, 긴급 모드, 시스템 종료) 사용자 입력은 syscall 5를 사용하여 받으며, 유효하지 않은 입력 시 오류 메시지를 출력하고 재입력을 요청한다.

- 모드별 신호등 작동:
각 모드는 별도의 함수(execute_normal_mode, execute_night_mode, execute_emergency_mode)로 구현되어 있으며, 선택된 모드에 따라 해당 함수가 호출된다.
- 프로그램 종료 및 통계 출력:
사용자가 "0"을 입력하여 종료를 선택하거나, 특정 모드의 반복이 끝나면 시스템은 print_statistics 함수를 통해 총 신호 상태 변경 횟수와 누적 런타임을 콘솔에 출력한다. 이후 print_goodbye 메시지와 함께 프로그램을 종료한다.

2.2. 모드별 신호등 상세 작동

(1) 일반 모드 (execute_normal_mode) - 좌회전 신호 포함



실제 교차로의 신호등 흐름을 반영하여 6단계의 신호 상태를 3회 반복 (총 18) 시뮬레이션한다.

단계	차량 신호	보행자 신호	지속시간
1	NS 좌회전, EW 적색	모두 대기	3초
2	NS 직진, EW 적색	NS 횡단 가능	5초
3	NS 황색, EW 적색	모두 대기	2초
4	EW 좌회전, NS 적색	모두 대기	3초
5	EW 직진, NS 적색	EW 횡단 가능	5초
6	EW 황색, NS 적색	모두 대기	2초

각 상태 진입 시, 해당 신호 메시지 및 보행자 안내(print_ns_left_ew_red, print_pedestrian_wait 등)를 콘솔에 출력한다. print_time_info 함수를 통해 현재 상태의 지속 시간과 시스템의 누적 경과 시간을 함께 출력하여 시뮬레이션의 시간 흐름을 직관적으로 파악할 수 있도록 했다. 또한 update_cycle_counter 함수를 호출하여 신호 상태 변경 횟수(\$s2)와 누적 시간(total_seconds)을 갱신하며, delay_ms 함수로 실제 시간 지연을 시뮬레이터 상에서 구현한다.

(2) 야간 점멸 모드 (execute_night_mode)

Mars Messages	Run I/O
Clear	Select Mode (1:Normal with Left Turns, 2:Night, 3:Emergency, 0:Exit): 2 *** Night Mode (Flashing) *** [NS: FLASH EW: RED] - Night Flash State Duration: +1s (Total: 1s) [NS: RED EW: FLASH] - Night Flash State Duration: +1s (Total: 2s) [NS: FLASH EW: RED] - Night Flash State Duration: +1s (Total: 3s) [NS: RED EW: FLASH] - Night Flash State Duration: +1s (Total: 4s) [NS: FLASH EW: RED] - Night Flash State Duration: +1s (Total: 5s) [NS: RED EW: FLASH] - Night Flash State Duration: +1s (Total: 6s) [NS: FLASH EW: RED] - Night Flash State Duration: +1s (Total: 7s)

교차로가 한산한 야간 상황을 시뮬레이션한다. NS와 EW 방향의 신호가 1초씩 번갈아 깜빡이는 점멸 신호를 10회 반복한다. 각 점멸 시마다 print_night_flash_ns 또는 print_night_flash_ew를 사용하며, print_time_info 및 update_cycle_counter를 통해 시간 및 횟수를 관리한다.

(3) 긴급 모드 (execute_emergency_mode)

Mars Messages	Run I/O
Clear	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)

모든 방향의 신호등이 빨간불(ns_red_ew_red) 상태를 5초간 유지하는 과정을 5회 반복한다. 이는 비상 차량 통과와 같은 긴급 교통 통제 상황을 반영한 시나리오이다.

2.3. 주요 함수와 작동 의도

– initialize_system:

```
124 # System Initialization Function
125 initialize_system:
126     addi $sp, $sp, -4
127     sw $ra, 0($sp)
128     li $a0, 0      # Current State (less used now, sequence is fixed in normal mode)
129     li $a1, 0      # (Unused)
130     li $a2, 0      # Total signal states processed
131     li $a3, 1      # System mode (default Normal)
132     li $t0, 0      # Local cycle counter (for full sequences)
133     sw $zero, total_seconds # Clears total_seconds
134     lw $ra, 0($sp)
135     addi $sp, $sp, 4
136     jr $ra
```

시스템 작동 전 전역 변수(누적 시간) 및 레지스터(상태 카운터 등)를 초기화하여 프로그램의 안정적인 시작을 보장한다.

– get_user_mode:

```
153 # User Mode Selection Function
154 get_user_mode:
155     addi $sp, $sp, -4
156     sw $ra, 0($sp)
157     li $v0, 4
158     la $a0, mode_prompt
159     syscall
160     li $v0, 5
161     syscall
162     lw $ra, 0($sp)
163     addi $sp, $sp, 4
164     jr $ra
```

사용자 입력을 받아 시스템의 작동 모드를 결정하며, 잘못된 입력에 대한 예외 처리를 통해 사용자 편의성을 높인다.

- 각 신호 출력 함수 (print...):

```
175 normal_full_cycle_loop: # Renamed for clarity
176     # Phase 1: NS_LEFT, EW_RED (3s)
177     jal print_ns_left_ew_red
178     jal print_pedestrian_wait    # Pedestrians wait
179     lw $a2, total_seconds
180     li $a1, 3
181     jal print_time_info
182     li $a1, 3
183     jal update_cycle_counter
184     lw $a0, LEFT_TURN_TIME
185     li $a1, 3
186     jal delay_ms
187
188     # Phase 2: NS_GREEN, EW_RED (5s)
189     jal print_ns_green_ew_red
190     jal print_pedestrian_ns    # NS Pedestrians cross
191     lw $a2, total_seconds
192     li $a1, 5
193     jal print_time_info
194     li $a1, 5
195     jal update_cycle_counter
196     lw $a0, GREEN_TIME
197     li $a1, 5
198     jal delay_ms
199
200     # Phase 3: NS_YELLOW, EW_RED (2s)
201     jal print_ns_yellow_ew_red
202     jal print_pedestrian_wait    # Pedestrians wait
203     lw $a2, total_seconds
204     li $a1, 2
205     jal print_time_info
206     li $a1, 2
207     jal update_cycle_counter
208     lw $a0, YELLOW_TIME
209     li $a1, 2
210     jal delay_ms
211
```

print_ns_green_ew_red, print_pedestrian_wait 등 각 신호 상태, 보행자 안내, 시스템 메시지 등을 콘솔에 출력한다. 함수별로 기능을 분리하여 코드의 가독성과 재사용성, 유지보수성을 향상시켰다.

– print_time_info:

```
414 # Time Info Print Function: per-state and cumulative seconds
415 # $a1: 이번 상태의 초(초단위, 예: 5 또는 2)
416 # $a2: 진입 전 누적 초 (total_seconds 값)
417 print_time_info:
418     addi $sp, $sp, -4
419     sw $ra, 0($sp)
420     # $a1 = current state duration in seconds
421     # $a2 = total seconds before this state
422
423     li $v0, 4
424     la $a0, elapsed_msg      # "State Duration: +"
425     syscall
426
427     move $a0, $a1            # X (이번 상태 초)
428     li $v0, 1
429     syscall
430
431     li $v0, 4
432     la $a0, s_msg           # "s (Total: "
433     syscall
434
435     addu $t2, $a1, $a2      # $t2 = 이전 누적 + 이번 초 (새로운 총 누적 시간)
436     move $a0, $t2
437     li $v0, 1
438     syscall
439
440     li $v0, 4
441     la $a0, close_msg       # "s)\n"
442     syscall
443
444     lw $ra, 0($sp)
445     addi $sp, $sp, 4
446     jr $ra
```

매 신호 상태 진입 시, "State Duration: +X초 (누적 Y초)" 형태로 출력하여 시뮬레이션의 시간 흐름을 직관적으로 파악할 수 있도록 한다.

– delay_ms:

```
448 # Delay Function (in milliseconds)
449 # $a0: milliseconds to delay
450 delay_ms:
451     # $a0 already contains milliseconds
452     li $v0, 32              # sleep syscall
453     syscall
454     jr $ra
```

syscall 32를 이용하여 지정된 밀리초만큼 실제 시간을 제어(시뮬레이션 지연)한다. 이는 실제 시간 경과를 모방하기 위한 핵심적인 기능이다.

- update_cycle_counter:

```
456 # Cycle Counter Update (and accumulate seconds)
457 # $a1: 이번 상태의 초 (seconds for current state)
458 update_cycle_counter:
459     addi $s2, $s2, 1          # Increment total states processed count
460     lw $t1, total_seconds
461     add $t1, $t1, $a1         # Add current state's seconds to total_seconds
462     sw $t1, total_seconds
463     jr $ra
```

신호 상태가 바뀔 때마다 \$s2 레지스터(총 상태 변경 횟수)와 total_seconds 변수(누적 시간)를 갱신한다.

- ask_continue:

```
465 # Continue Check
466 ask_continue:
467     addi $sp, $sp, -4
468     sw $ra, 0($sp)
469     li $v0, 4
470     la $a0, continue_prompt
471     syscall
472     li $v0, 5          # Read integer
473     syscall             # $v0 contains the user's input (1 or 0)
474     lw $ra, 0($sp)
475     addi $sp, $sp, 4
476     jr $ra
477
```

사용자에게 프로그램 반복 실행 여부를 선택(1: 계속, 0: 종료)할 수 있도록 프롬프트를 제공한다.

- print_statistics:

```
485 # Statistics Print
486 print_statistics:
487     addi $sp, $sp, -4
488     sw $ra, 0($sp)
489     li $v0, 4
490     la $a0, divider
491     syscall
492
493     li $v0, 4
494     la $a0, total_cycles # "Total Signal States Processed: "
495     syscall
496     li $v0, 1
497     move $a0, $a2 # Print $a2 (total states)
498     syscall
499     li $v0, 4
500     la $a0, cycles_msg # " states#n"
501     syscall
502
503     li $v0, 4
504     la $a0, runtime_msg # "Total Runtime: "
505     syscall
506     lw $t1, total_seconds
507     move $a0, $t1 # Print total accumulated seconds
508     li $v0, 1
509     syscall
510     li $v0, 4
511     la $a0, seconds_msg # "sec#n"
512     syscall
513
514     lw $ra, 0($sp)
515     addi $sp, $sp, 4
516     jr $ra
```

프로그램 종료 직전, 전체 시뮬레이션 동안의 총 신호 상태 변경 횟수와 누적 경과 시간을 요약하여 콘솔에 출력한다.

3. 프로젝트를 통해 느낀 점 및 학습 효과

이번 MIPS 어셈블리어 신호등 제어 시스템 프로젝트를 수행하면서 다음과 같은 점들을 깊이 있게 체감하고 학습할 수 있었다.

- 하드웨어 제어의 본질 체감:

고급 언어에서 추상화되어 알기 어려웠던 레지스터, 메모리 접근, 시스템 콜(syscall) 등을 직접 다루며, 컴퓨터가 실제로 어떻게 동작하는지에 대한 근본적인 이해를 높일 수 있었다. 특히 syscall 32를 이용한 시간 지연 구현은 소프트웨어가 하드웨어 시계를 활용하는 방식을 직접 체험하는 좋은 계기가 되었다.

- 상태 기반 시스템의 복잡성:

신호등 제어는 단순히 색을 바꾸는 것을 넘어, 차량 및 보행자 흐름, 좌회전, 야간/비상 시나리오까지 고려해야 하는 복잡한 상태 기반 시스템임을 다시금 깨달았다. 이를 어셈블리어로 직접 구현하면서 각 상태 간의 전이와 논리적 연결을 설계하는 중요성을 인지하게 되었다.

- 모듈화와 함수 설계의 중요성:

어셈블리어에서 함수 호출(jal) 및 스택 관리(\$sp, \$ra)를 직접 구현하며, 코드의 재사용성과 유지보수성을 높이는 데 "모듈화"가 필수적임을 실감했다. 신호 상태별 출력 함수를 별도로 분리함으로써 코드의 가독성과 관리 편의성을 크게 개선할 수 있었다.

- 정확한 시간 제어의 도전:

시뮬레이션 상의 delay와 실제 하드웨어의 시간은 완전히 일치하지 않을 수 있지만, total_seconds와 delay_ms를 통해 각 상태별 시간 누적을 직접 관리하는 것이 실시간 시스템 설계의 핵심임을 체감했다.