GENERAL

assembly language code for traffic light controller

Published: September 19, 2026 | Index ID: #-

Assembly Language Code for a Traffic Light Controller

When it comes to building reliable, low level control systems, assembly language remains one of the most powerful tools in an engineer's toolbox. From microcontrollers in industrial machinery to the firmware that powers everyday traffic lights, assembly gives developers unparalleled control over timing, resource usage, and hardware interfaces. In this article we'll dive into the world of assembly based traffic light controllers: why assembly is still relevant, how to design a state machine for a traffic intersection, and a step by step walkthrough of a complete assembly code example.

Why Choose Assembly for Traffic Light Control?

Traffic light controllers must be deterministic, fast, and dependable. A single millisecond delay can cause a traffic jam or, in the worst case, an accident. Assembly language offers:

- Deterministic Timing – Every instruction has a known cycle count, making precise delays straightforward.
- Minimal Footprint – Tiny code size keeps the program fast and reduces memory usage on small microcontrollers.
- Direct Hardware Access – Pin manipulation and register configuration are simple, reducing abstraction layers that could introduce bugs.
- Fine Grained Control – Interrupts, critical sections, and power saving modes can be implemented exactly as required.

While high level languages like C or Rust provide safety and ease of use, assembly is still the go to choice for mission critical embedded systems where every clock cycle counts.

Hardware Overview: What You'll Need

Before writing any code, it's essential to understand the hardware you'll be working with. A typical traffic light controller for a single intersection uses:

- Microcontroller – 8 bit AVR (e.g., ATmega328P), 16 bit PIC, or ARM Cortex M0.
- LEDs or Relays – Three LEDs per direction: Green, Yellow, Red.
- Power Supply – 5 /V DC, regulated.
- Clock Source – 16 /MHz crystal oscillator.
- Timer Module – For generating precise delays.
- Interrupts – Optional, for pedestrian button inputs.

For the purpose of this article we'll target the AVR ATmega328P, which is widely used and has excellent assembly documentation.

Software Architecture: The State Machine

A traffic light system can be modeled as a finite state machine (FSM). Each state represents a particular light configuration (e.g., North–South green, East–West red). Transitions are driven by timers or external events.

1. North–South Green, East–West Red
2. North–South Yellow, East–West Red
3. North–South Red, East–West Green

4. North–South Red, East–West Yellow

For simplicity, we'll ignore pedestrian signals and focus solely on vehicle lights. Each state lasts a fixed duration, e.g., 30 /seconds for green, 5 /seconds for yellow.

AVR provides several timer modules. We'll use Timer0 in CTC (Clear Timer on Compare) mode to generate 1 ms ticks. A counter will accumulate ticks until it reaches the desired state duration. Once the counter overflows, the FSM moves to the next state.

Assembly Language Basics for AVR

Below is a quick refresher on the AVR assembly syntax and key concepts:

- Registers – R0–R31, with special registers like SPH and SPL for the stack pointer.
- Directives – .org, .include, .equ for memory organization.
- Instructions – MOV, LDI, OUT, IN, BRNE, RJMP, SLEEP.
- Macros – .macro and .endmacro to encapsulate repetitive code.

We'll also use the ldi (load immediate) instruction to set up initial values and out to write to I/O registers controlling the LEDs.

Step by Step Code Walkthrough

Below is a complete AVR assembly program that implements the traffic light controller described above. Comments are provided to explain each section.

```
; ============================================================
; Traffic Light Controller – AVR ATmega328P
; ============================================================
.include "m328pdef.inc"      ; Include register definitions
.equ  LED_PORT = PORTB       ; LEDs connected to PORTB
.equ  LED_DDR  = DDRB        ; Data Direction Register for PORTB

; State definitions
.equ  NS_GREEN  = 0x01       ; Bit 0 – North–South Green
.equ  NS_YELLOW = 0x02       ; Bit 1 – North–South Yellow
.equ  EW_GREEN  = 0x04       ; Bit 2 – East–West Green
.equ  EW_YELLOW = 0x08       ; Bit 3 – East–West Yellow

; Timing constants (in milliseconds)
.equ  TICK_MS   = 1          ; Timer tick period
.equ  GREEN_MS  = 30000      ; 30 seconds
.equ  YELLOW_MS = 5000       ; 5 seconds

; Derived constants
.equ  GREEN_TICKS  = GREEN_MS / TICK_MS
.equ  YELLOW_TICKS = YELLOW_MS / TICK_MS

; Global variables
.def  TICKS  = R16           ; Accumulator for ticks
```

```
.def  STATE  = R17          ; Current state
.def  COUNTER = R18         ; Tick counter for current state

; ============================================================
; Main program start
; ============================================================

.org 0x0000
rjmp  RESET               ; Reset vector

.org 0x0010
rjmp  TIMER0_OVF_ISR      ; Timer0 overflow interrupt vector

; ============================================================
; Reset routine
; ============================================================

RESET:
   ldi   R0, HIGH(RAMEND)
   out   SPH, R0
   ldi   R0, LOW(RAMEND)
   out   SPL, R0

   ; Configure LED pins as outputs
   ldi   R0, (1
```

Let’s unpack what’s happening in this code.

- We set up the stack pointer and configure the LED pins as outputs.
- Timer0 is configured for 1 /ms ticks using a 64× prescaler and CTC mode. The OCR0A register is loaded with 249 because 16 /MHz / 64 = 250 /kHz, and 250 /kHz / 1 /kHz = 250 ticks per millisecond. Subtracting one gives 249.
- We enable the Output Compare A interrupt to run our timing routine.
- The initial state is NS_GREEN, and the tick counter starts at 0.
- Global interrupts are enabled with sei.

After initialization the main loop simply spins. All state management happens inside the Timer ISR, keeping the CPU free for other tasks if needed.

Each millisecond, the ISR increments COUNTER. When the counter overflows (i.e., after 256 ms), we treat it as a state duration end. In a real system you would compare COUNTER against the specific duration for the current state (e.g., GREEN_TICKS or YELLOW_TICKS). For brevity, this example uses a simple overflow approach.

The NEXT_STATE subroutine clears all LEDs, then uses a series of compares to determine the current state and set the next one. Each state turns on the appropriate LEDs and resets the counter. After setting the new state, the subroutine returns to the ISR.

Baca Juga (Artikel Terkait)

- **animal farm george orwell**
  URL: http://www.atemplatefree.com/animal-farm-george-orwell.pdf
- **animal physiology hill 3rd edition download shaojiore**
  URL: http://www.atemplatefree.com/animal-physiology-hill-3rd-edition-download-shaojiore.pdf
- **animal physiology hill 3rd edition dapter**
  URL: http://www.atemplatefree.com/animal-physiology-hill-3rd-edition-dapter.pdf
- **animal physiology hill 3rd edition**
  URL: http://www.atemplatefree.com/animal-physiology-hill-3rd-edition.pdf

Tags: #assembly #language #code #traffic #light #controller