Project 3.4 AVR Assembly: Traffic Light

Purpose

The purpose of the traffic light project is to teach a foundation for AVR assembly language. By coding an assembly program to control the sequence of a traffic light, basic assembly language techniques are introduced.

Reference

Project description: http://darcy.rsgc.on.ca/ACES/TEI4M/Tasks.html#TrafficLight

Project GitHub: https://github.com/rohan-development/TrafficLightAssemblyIntroduction

Theory

In Project 3.2 CHUMP, the concept of assembly language programming was introduced. CHUMP was a 4-bit processor, which restricted it to a maximum of 16 instructions. In assembly language, rather than use easy-to-understand syntax, such as `if()` statements and `for()` loops, the user is able to interact directly with the hardware, specifying a CPU instruction. Where 1 high-level instruction can translate to several [hundred] clock cycles, a line of assembly, or instruction, generally translates to 1 executed clock cycle. There are exceptions to this; some complex instructions can use up to 5 clock cycles. When working with a limited-processing-power microcontroller, such as the ATmega328P, this allows for extreme optimization of the hardware. This means that the same microcontroller can be used for more complex tasks.

It is important to note the internal memory structure of the ATmega328P before gaining an understanding of AVR assembly language. Figure 1 depicts the internal RAM structure of the ATmega328P. There are a total of 2048 8-bit registers (bytes) of SRAM, and a total of 256 other registers.

Figure 1. ATmega328P RAM Structure

Reg #	Description	Address Range
32	GP Registers	0x0000-0x001F
64	I/O Registers	0x0020-0x005F
160	Extended Registers	0x0060-0x0FF
2048	General SRAM	0x0100-0x08FF

The first 32 registers are the general-purpose registers. These are known as *working registers*. The working registers are used to hold temporary values, and to compute them, before they are either stored to memory using the `sts` instruction, or written to an I/O or extended I/O register. For example, there is no way to load an immediate (constant) directly into an I/O register, so the constant must first be put into a working register, and then the working register can be stored to the I/O register.

The next 64 registers are the I/O registers. The main difference between the standard and extended I/O registers are the way they are addressed. The standard registers occupy a lower set of addresses, which means that they can be accessed in a single clock cycle. The extended I/O registers take an extra byte to address, which means that they take a different set of instructions to access. The more commonly used registers, such as the PORT, PIN, DDR, and interrupt registers are in the standard register space, whereas configuration registers, such as the SPDR, which controls the SPI bus, is in the extended I/O space.

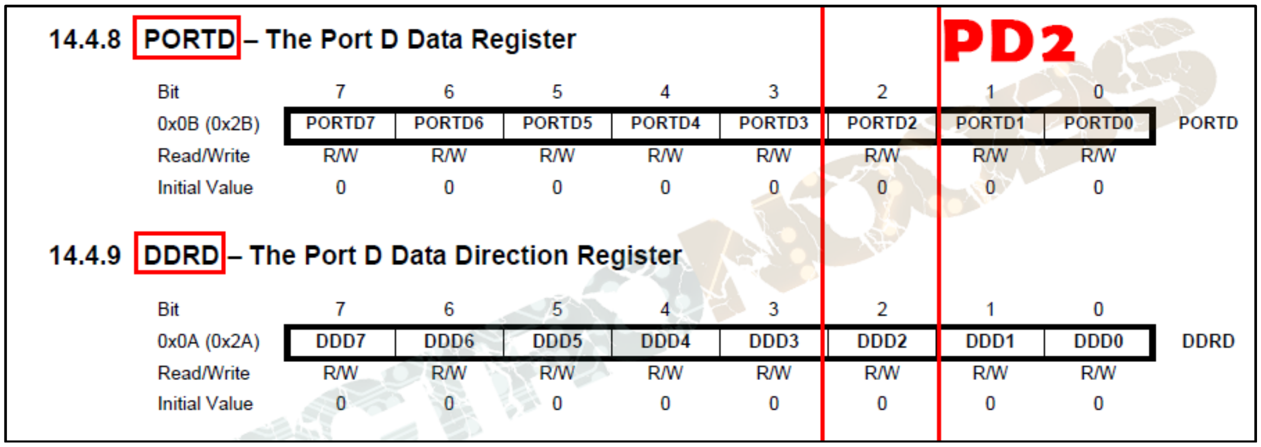
In AVR assembly, there are around 131 instructions. There are several instructions that perform similar functions (such as loading a register with a value). Each instruction has a specific type of accepted operand, which can be a certain type of register (see Figure 1), or an immediate (constant).

Procedure

The ACES traffic light is a PCB with a green, a yellow, and a red LED onboard, arranged in a traffic light-like pattern. Each LED is connected to a pin, which allows each LED to be turned on or off.

Parts Table	
Quantity	**Description**
1	Arduino Uno
1	RSGC ACES Traffic Light PCB

In a high-level program, programming a simple sequence (such as green for 2 seconds, yellow for 1, and red for 2), would require each light to be explicitly set, as well as its own delay. In addition to being inefficient in terms of program space, it is inefficient in terms of resources used. A single `digitalWrite()` statement uses around 50 clock cycles, because it performs several redundant checks under the hood.



14.4.8 PORTD – The Port D Data Register

Bit	7	6	5	4	3	2	1	0	
0x0B (0x2B)	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

14.4.9 DDRD – The Port D Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x0A (0x2A)	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 1. PORTD Pin Associated Registers

In assembly, `digitalWrite()` is replaced with 1-2 instructions, depending on the method used: in the first method, a bit mask corresponding to the bit location is placed into a working register, using the `ldi` instruction. Then, the working register is written to the desired port, using the `out` instruction. While this method uses 2 instructions, in can manipulate the entire port in 2 cycles. The second method is to set or clear an individual bit. To replace `digitalWrite(2, X)`, where *X* is either high or low, the statement `sbi PORTD,2` or `cbi PORTD,2` is used, respectively. This is because PD2 corresponds to digital pin 2 (see Figure 1).

The second high-level function that must be replaced in assembly for the traffic light project is the delay function. By definition, the delay function simply "wastes" clock cycles, so there is no real optimization to be had by performing this function in assembly. To delay for 1 second, 16 million clock cycles must go by, which means that 16 million instructions must be executed. For this, a register is simply decremented 16 million times in a loop. In an 8-bit CPU, since a loop can only have 256 iterations, multiple loops must run. For this, some computation must be run. There are 3 loops: one with 256 iterations, one with 43, and one with 82. Since the 256-iteration loop takes 2 cycles to execute, the total number of clock cycles is equal to: $256 \times 2 \times 43 \times 82 = 1805312$. At a clock speed of 16 MHz, this translates to approximately 1.13 seconds. The extra delay is due to the overhead of the loops.

The following is a breakdown of how the traffic light sequence is programmed in assembly, efficiently (see Reference for GitHub, or Code section): The first 6 lines are compiler directives that map each colour to a pin, and include a set of predefines that correspond specifically to the ATmega328P. The program starts on line 7, with an `ldi` instruction. This instruction is "load immediate," and it puts a constant into a working register (see Theory section). The stored constant is a bitmask that sets the pins attached to the red, yellow, and green led high. This bitmask is then stored to DDRB on the next line using the `out` instruction. Line 9 is simply a label for line 10, as it is a "branching point" which means that the program counter can be set to this point later on using a relative jump (or a regular jump). Line 10 puts the bitmask corresponding to the green LED into working register 16. It is the reset point of the LED sequence.

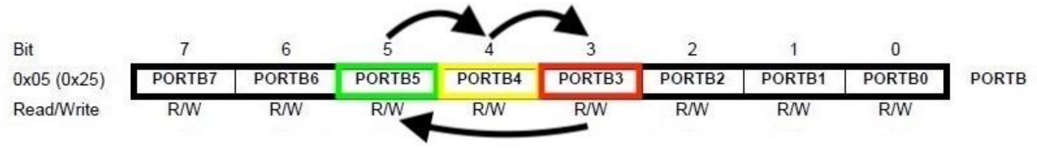


Figure 2. Green, Yellow, Red Pin Map and Bit Sequence

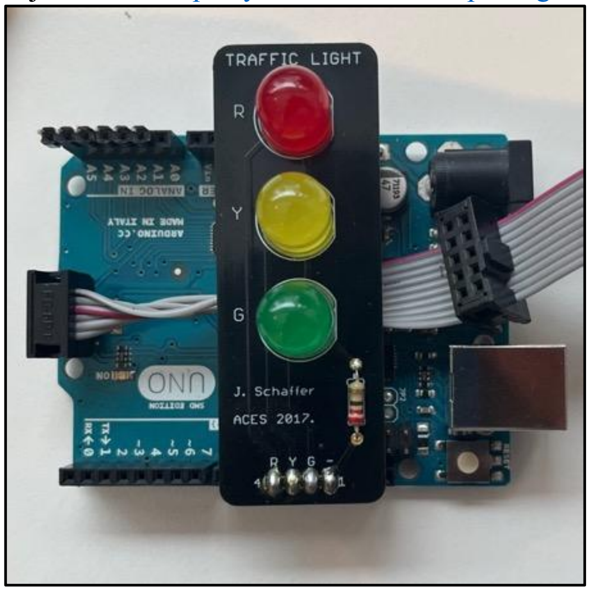
Since the yellow and red LEDs are simply the green LED shifted right once and twice respectively (see Figure 2), the program simply shifts working register 16 right, delays 1 second twice, by using the `rcall` instruction in conjunction with the delay1s function. Then, on each iteration, it uses the `cpi`, or compare instruction to check if it has been shifted beyond the red LED. If it has, the zero flag of the status register is set. This means that when a `breq` instruction is executed, it will branch. This branches to the reset point, and starts the sequence again. With the same logic when yellow, it only delays once.

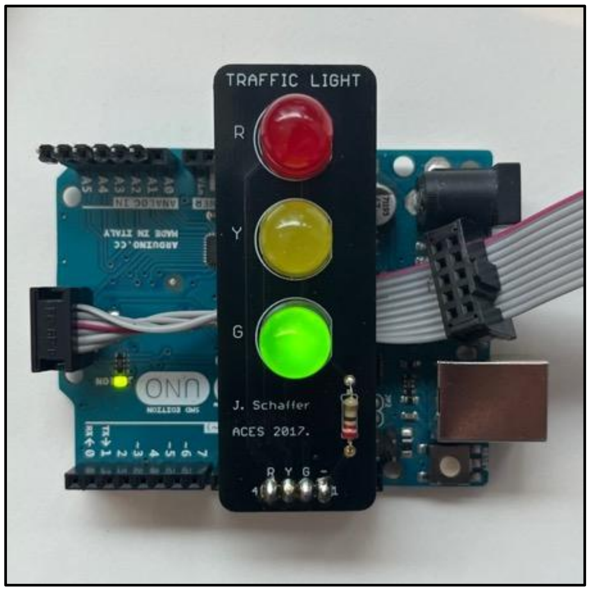
Code

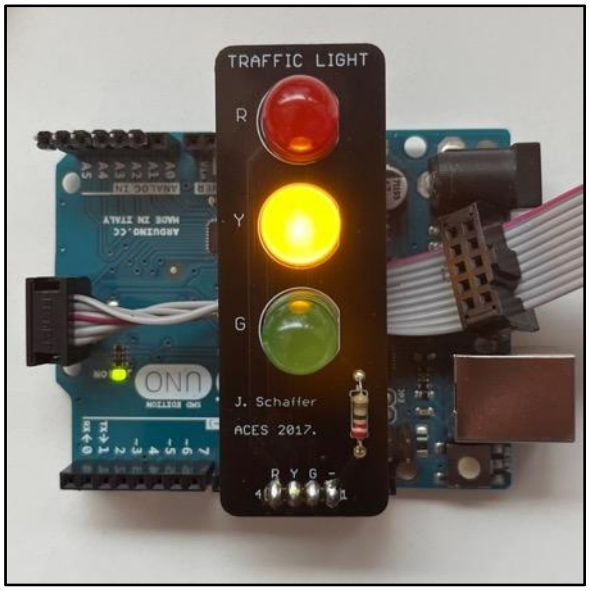
```
.include "m328pdef.inc"    ;Include 328P predefines
.equ GREEN = (1<<PB5)      ;Define green LED pin
.equ YELL = (1<<PB4)       ;Define yellow LED pin
.equ RED = (1<<PB3)        ;Define red LED pin
.equ PORT = PORTB          ;Define used port (DDR)
.equ DDR = DDRB            ;Define used port
ldi r16,GREEN|YELL|RED     ;Bitmask of 3 pins in r16
out DDR, r16               ;Put bitmask into DDR
reset:                     ;Label for reset point
ldi r16,GREEN              ;Put green into the bitmask
run:                       ;"Loop" label
out PORT,r16               ;Write out current colour
rcall delay1s              ;Wait 1 second
cpi r16,YELL               ;Check if YELL is in r16
breq yellow                ;Branch if YELL was in r16
rcall delay1s              ;Wait another second if not yellow
yellow:                    ;Label for when colour=yellow
lsr r16                    ;Shift colour down 1 bit
cpi r16,(1<<PB2)           ;Check if colours overflowed
breq reset                 ;If overflowed, reset colour
rjmp run                   ;If no overflow, loop again
delay1s:                   ;Delay 1 second function
  ldi  r18, 82             ;Load 82 into r18
  ldi  r19, 43             ;Load 43 into r19
  ldi  r20, 0              ;Load 0 into r20
L1:                        ;Loop 1 label
  dec  r20                 ;Subtract 1 from r20
  brne L1                  ;If r20 has a 0, go to L1
  dec  r19                 ;If not, decrement r19
  brne L1                  ;If r19 has a 0, go to L1
  dec  r18                 ;If not, decrement r18
  brne L1                  ;If r18 has a 0, go to L1
  ret                      ;Return (exit function)
```

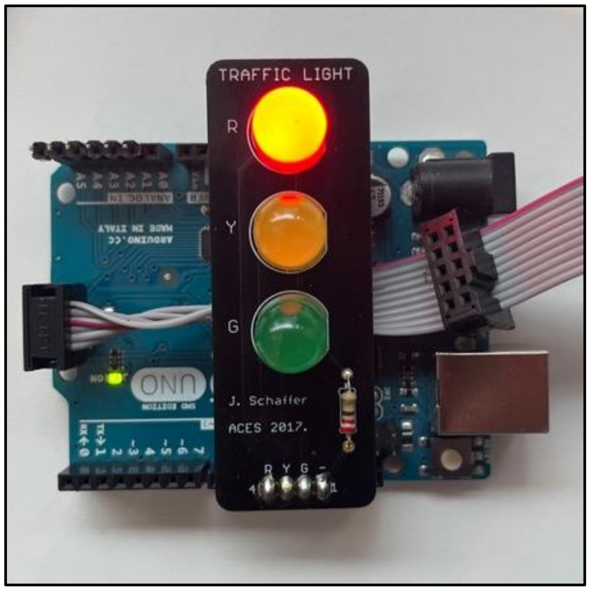
Media

Project video: https://youtu.be/VmWGHp7-iGg


Traffic Light Build


Traffic Light Green (2 Seconds)


Traffic Light Yellow (1 Second)


Traffic Light Red (2 Seconds)

ACES Traffic Light PCB Side View

Reflection

I think that this was a great introduction to assembly coding. I learned some new techniques (like using a working register and shifting through it to affect certain pins) that I have no doubt will be helpful in the future. As I think of new ISPs, I am hoping that I do something involving assembly, although I'm not sure what that looks like yet, and I still have to experiment with it to determine my own skill level with assembly.

I will say that most of my time on this project was not spent building the circuit, writing the code, or even writing my DER and making the video. It was without a doubt, getting Microchip Studio to cooperate. And that is certainly saying something, because I had to record my video several times. The first time, it didn't record sound. The second time, it only recorded the back window, not the processor status and I/O status windows.

I spent a very long time trying to figure out why AVRDUDE wouldn't launch properly (eventually I figured out that it was because I was missing the AVRDUDE configuration file). I also spent a fairly long time trying to import my Microchip Studio code into word with the formatting. I never actually figured it out, so I just manually changed all the colours instead.

Overall, I learned a fair amount with this project, though I do still have much learning to do in terms of assembly, windows screen recording, and Microchip Studio. I look forward to the next assembly language project.