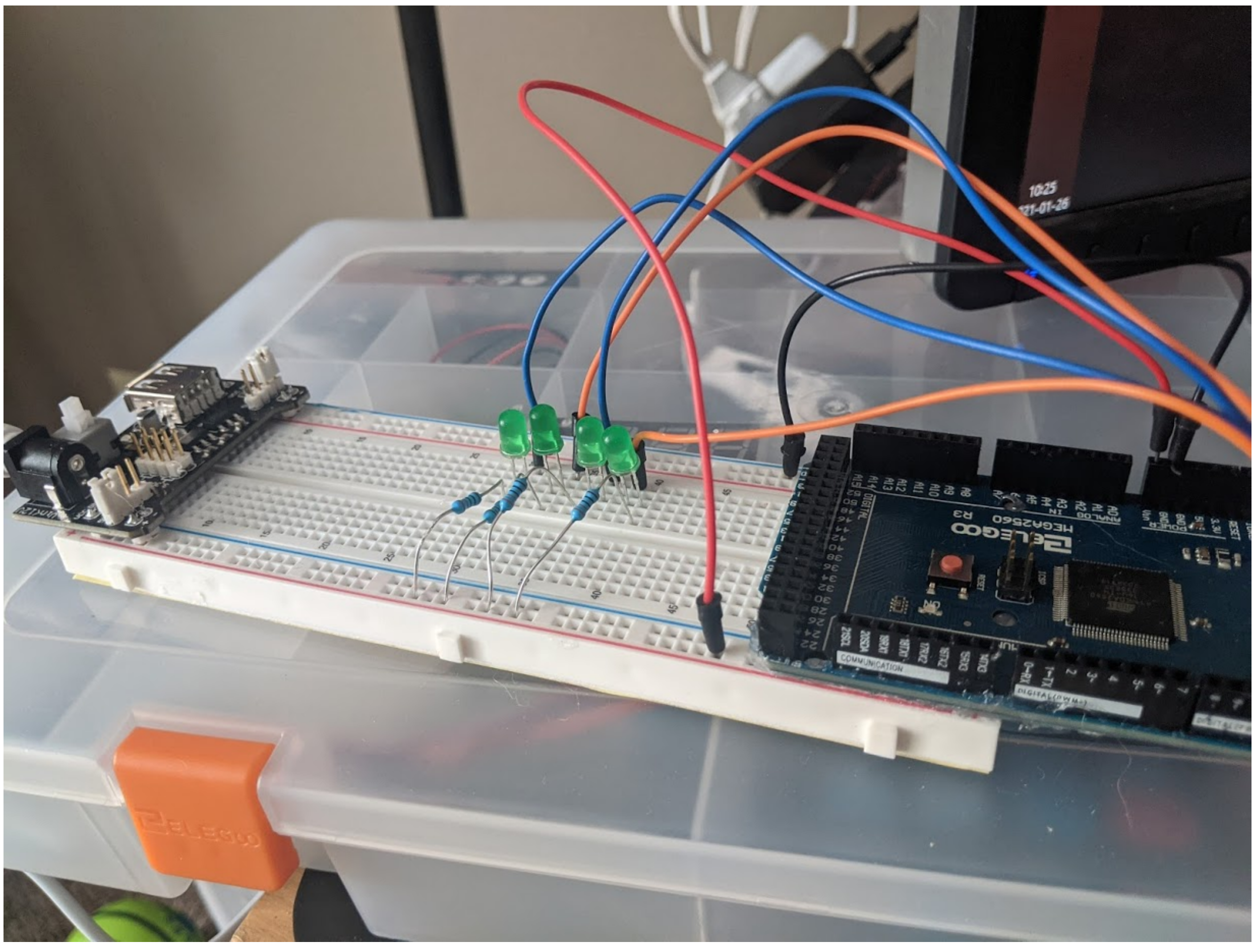
Lab 3 Report

4-bit Binary Counter in Assembly

CENG 347 Embedded Intelligent Systems



Samuel Donovan & Samantha Pfeiffer

2021-02-11

OVERVIEW

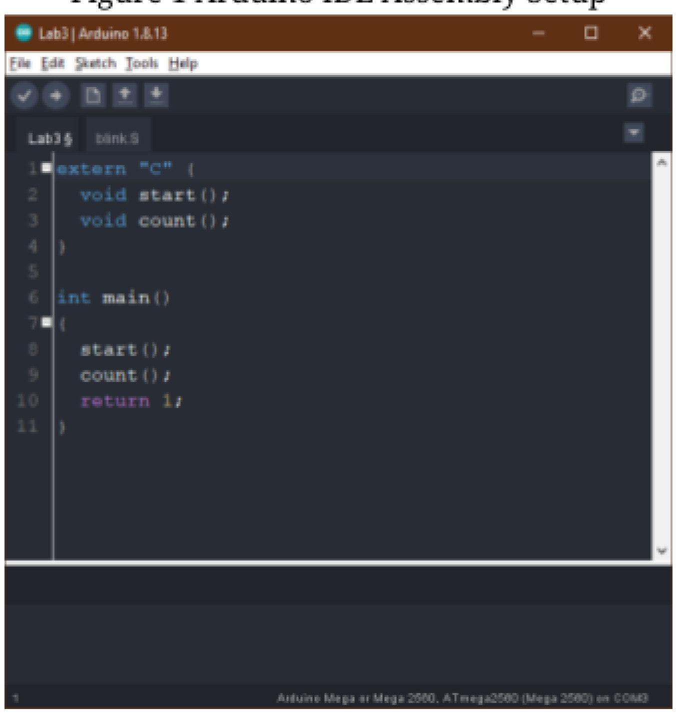
The purpose of this lab was to program up a simple 4-bit binary counter in assembly.

--- ASSEMBLY ---

To program in assembly using the Arduino IDE, you must first create a normal sketch file that contains the `main` function, where the program will begin execution. In the same `.ini` file, include an `extern “C”{}` section and define the signatures for the functions that will be written in assembly. An example for two functions that take no arguments can be seen below in **Figure 1**.

After this, you can create a file in the same directory with a `.S` extension, and write your AVR assembly in there.

Figure 1 Arduino IDE Assembly Setup



--- 4-BIT BINARY COUNTER ---

A 4-bit binary counter is used to express a base-2 number between 0000_2 and 1111_2 (0_{10} to 15_{10}). Table 1 shows each bit value as it corresponds to its decimal counterpart. The table also shows the inverse logic values of those bits; the need for these values will be explained in the next section.

Table 1 4-Bit Binary Counter Truth Table

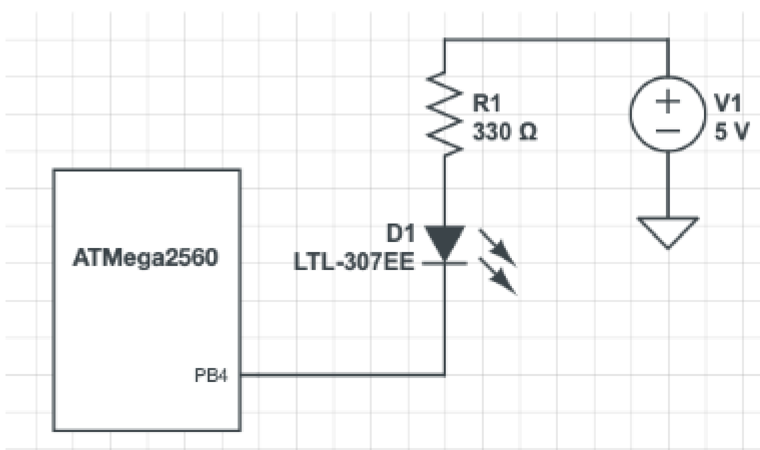
Decimal Value	B_3	B_2	B_1	B_0	$\sim B_3$	$\sim B_2$	$\sim B_1$	$\sim B_0$
0	0	0	0	0	1	1	1	1
1	0	0	0	1	1	1	1	0
2	0	0	1	0	1	1	0	1
3	0	0	1	1	1	1	0	0
4	0	1	0	0	1	0	1	1
5	0	1	0	1	1	0	1	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	0	0
8	1	0	0	0	0	1	1	1
9	1	0	0	1	0	1	1	0
10	1	0	1	0	0	1	0	1
11	1	0	1	1	0	1	0	0
12	1	1	0	0	0	0	1	1
13	1	1	0	1	0	0	1	0
14	1	1	1	0	0	0	0	1
15	1	1	1	1	0	0	0	0

METHODS

--- CIRCUITRY ---

The 4-bit binary counter was created using an ATMega2560 controller board, an 830 tie-point breadboard, a power supply module, four LEDs, and four 330 Ohm resistors. Each LED represented a binary bit and would light up to indicate a bit value of 1.

Since microcontrollers are not meant to directly drive LEDs, each anode was connected to a positive 5v source and each cathode to PORTB of the microcontroller. The counter was created using the high nybble of PORTB, so pins PB7, PB6, PB5, and PB4 were used. In order to current limit, a resistor was connected in series between each LED and the power supply. An example of this configuration was provided in the lab write up and can be seen below.



This configuration creates inverse logic. When a pin is set to 0, the LED is forward biased by the +5V source, and therefore lit. Setting that pin to 1 removes the voltage drop, no longer forward biasing the LED, therefore turning it off. Thus, the ~B values of Table 1, were used to create the correct blinking pattern.

--- CODE ---

Since this lab was completed separately, following are descriptions of each team member's code.

PFEIFFER

In the approach shown in **Appendix A**, two assembly functions are used to count from 0 to 15. The function `count` uses `sbi` and `cbi` instructions to set the four counting bits to display the values 0 to 15. After setting the bits, `count` calls `delay_n_ms`. This function was modified from the provided delay function by adding a secondary loop that decrements the value passed in r21, allowing for up to 510ms of delay.

DONOVAN

In the approach shown in **Appendix B**, the code written utilizes integer overflow to simplify the code. Setting a register to the max value and continuously subtracting mimics the functionality of counting upward and then inverting the register. Here, this fact is utilized in `count` and `start` by setting the upper four bits of `PORTB` to 15 (`1111`), and subtracting `00010000` from the register (which subtracts 1 from the upper nybble). When the register reaches zero an overflow occurs, causing the register to loop back to `11110000`, and the cycle repeats. `r16` is used as temporary storage when manipulating the bits for `PORTB` .

The necessary DDR bits are also set in `start`.

Outside of `count` and `start` functions, the `delay_n_ms` and `delay_lp` labels serve the purpose of providing a delay function, which takes an input in `r0` which represents an approximate delay in milliseconds.

RESULTS

Both methods explained above resulted in a 4-bit binary counter. Each counter started at 0 and counted to 15 before repeating, as expected. Though each team member accomplished this task in different ways, the results were identical. For more results, see the video file submitted with this report.

APPENDIX A: 4-bit Binary Counter Assembly Code (PFEIFFER)

```
//Filename: CENG347_Lab3.c
//Written By: Sam Pfeiffer
//Confid: AVR ATMega2560 @16MHz
//Description: A 4-bit binary counter using PB7, PB6, PB5, and
// PB4. Counting is done by assignment statements in ASM.

extern "C" {
  void start();
  void count();
}

void init() {
  start();
}

int main()
{
  init();
  while(1)
  {
    count();
  }
  return 1;
}

#define __SFR_OFFSET 0

#include "avr/io.h"

.global start
.global count

start:
  ldi R16, 0xF0
  in  R17, DDRB
  or  R16, R17
  out DDRB, R16
  ret
```

```
count:
  sbi   PORTB, 7
  sbi   PORTB, 6
  sbi   PORTB, 5
  sbi   PORTB, 4
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms

  cbi   PORTB, 4
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms

  sbi   PORTB, 4
  cbi   PORTB, 5
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms

  cbi   PORTB, 4
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms

  sbi   PORTB, 4
  sbi   PORTB, 5
  cbi   PORTB, 6
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms

  cbi   PORTB, 4
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms

  sbi   PORTB, 4
  cbi   PORTB, 5
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms
```

```
cbi   PORTB, 4
ldi   r20, 250
ldi   r21, 250
call  delay_n_ms

sbi   PORTB, 4
sbi   PORTB, 5
sbi   PORTB, 6
cbi   PORTB, 7
ldi   r20, 250
ldi   r21, 250
call  delay_n_ms

cbi   PORTB, 4
ldi   r20, 250
ldi   r21, 250
call  delay_n_ms

sbi   PORTB, 4
cbi   PORTB, 5
ldi   r20, 250
ldi   r21, 250
call  delay_n_ms

cbi   PORTB, 4
ldi   r20, 250
ldi   r21, 250
call  delay_n_ms

sbi   PORTB, 4
sbi   PORTB, 5
cbi   PORTB, 6
ldi   r20, 250
ldi   r21, 250
call  delay_n_ms

cbi   PORTB, 4
ldi   r20, 250
ldi   r21, 250
call  delay_n_ms

sbi   PORTB, 4
cbi   PORTB, 5
```

```
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms

  cbi   PORTB, 4
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms

  sbi   PORTB, 4
  sbi   PORTB, 5
  sbi   PORTB, 6
  sbi   PORTB, 7
  ldi   r20, 250
  ldi   r21, 250
  call  delay_n_ms
  ret

delay_n_ms:
  ldi   r31, 3000>>8
  ldi   r30, 3000&255
delaylp:
  sbiw  r30, 1
  brne  delaylp
  subi  r20, 1
  brne  delay_n_ms
lp2:
  ldi   r31, 3000>>8
  ldi   r30, 3000&255
delaylp2:
  sbiw  r30, 1
  brne  delaylp2
  subi  r21, 1
  brne  lp2
  ret
```

APPENDIX B: 4-bit Binary Counter Assembly Code (DONOVAN)

```
#define __SFR_OFFSET 0 //Something to do with register address
offsetting

#include "avr/io.h"

.global start
.global count

delay_n_ms: //For 16 MHz
    ldi     r31, 3000>>8
    ldi     r30, 3000&255
delay_lp:
    sbiw    r30, 1
    brne    delay_lp
    subi    r20, 1
    brne    delay_n_ms
    ret

start:
    sbi     DDRB,7
    sbi     DDRB,6
    sbi     DDRB,5
    sbi     DDRB,4
    ldi     r16, 0b11110000 //Logic is inverted, so we're
counting down
    ret

count: //NOTE: r16 value auto-loops thanks to integer overflow
    out     PORTB, r16  //Write r16 to PORTB

    //Delay
    ldi     r20, 250    //Max delay == 255
    call    delay_n_ms

    subi     r16, 0b00010000
    jmp     count //Loop
```