# EE2016: Microprocessers Lab - Assignment 5 Report

Aadarsh Ramachandiran (EE23B001), Aditya Kartik (EE23B003), Rohita Datta (EE23B065)

September 24, 2024

**Abstract**

This is a detailed report on an assignment based on AVR I/O Port Configuration and Interfacing using the ATmega8 microcontroller, a USBASP board and a breadboard. The report contains the details of the solution, observation and comments on the programming and debugging experience.

## 1 Introduction

This assignment involves designing and implementing circuits for: (i) blinking an LED (ii) blinking two LEDs alternately (iii) a standard counter using two LEDs (iv) a Johnson counter using three LEDs (v) a two-bit adder realised using three LEDs and (vi) a four-bit adder using five LEDs and (vii) realising the previous results using an alternate port on the microcontroller.

### 1.1 Configuration of Ports

The architecture of AVR microcontrollers is register-based: information in the microcontroller such as the program memory, state of input pins and state of output pins is stored in registers. There are a total of 32 8-bit registers. Atmega8 has 23 I/O pins. These pins are grouped under what are known as ports. A port can be visualized as a bidirectional buffer with a specific address between the CPU and the external world. The CPU uses these ports to read input from and write output to them. The Atmega8 microcontroller has 3 ports: PortB, PortC, and PortD. Each of these ports is associated with 3 registers - DDRx, PORTx and PINx which set, respectively, the direction, output and input functionality of the ports. Each bit in these registers configures a pin of a port.

#### 1.1.1 Register DDRx

DDR stands for Data Direction Register and 'x' indicates a port alphabet. As the name suggests, this register is used to set the direction of port pins to either input or output. For input, we set to 0 while for output, we set to 1. For instance, let us consider PortB. To set this port as input or output, we need to initialize DDRB. Each bit n DDRB corresponds to the respective pin in PortB. Suppose we write DDRB = 0xFF, then all bits in PortB are configured to 'Output'. Similarly, DDRB=0x00 configures the port to be 'Input'.

#### 1.1.2 Register PORTx

PORTx sets a value to the corresponding pin. DDRx can set a bit to be either input or output while PORTx changes its functionality based on it. For example, PORTC = 0x01 will enable (or light up) an LED connected to this port after the appropriate DDRC setting.

**Note:** In this experiment, we will primarily use LEDs for showing the output of various tasks. LEDs can, in principle, be connected to different ports. For example, if we use PORTD, then all 8 pins are available to connect an LED (via a series resistor). If we use PORTC, pins PC0 to PC5 are available.

### 1.2 ATmega8 to USBASP board connections

For the ciruit implementation, we use the USBASP, which itself is a programmable microcontroller, to provide an elegant arrangement to program the ATmega8 on the breadboard. To enable this, we make a few connections between them whose schematic is as shown in **Figure 1**.

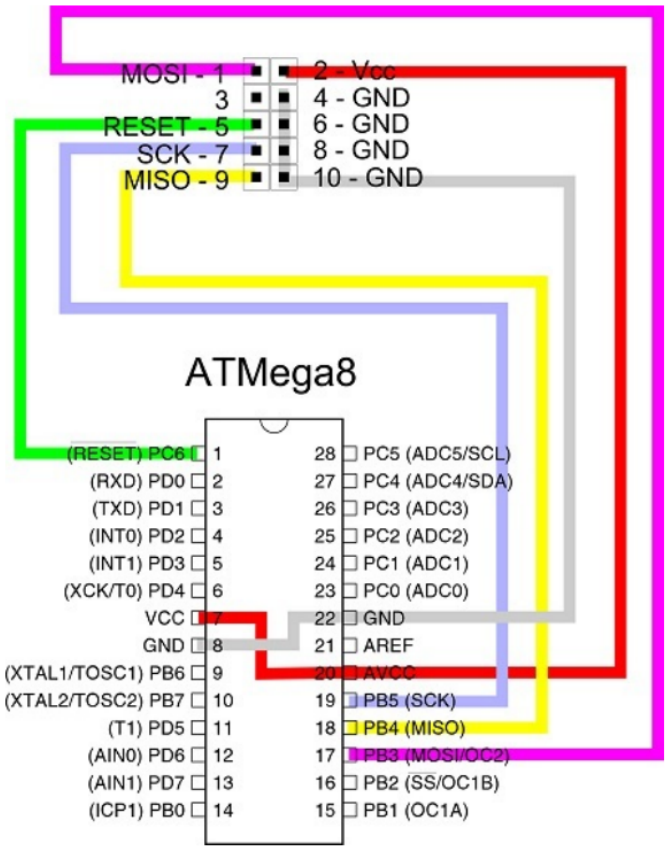


Figure 1: ATmega8 to USBASP board Connections

# 2 Design

## 2.1 Turning on LED permanently

### 2.1.1 Description

We desire to turn on the LED permanently. For this, the LED is connected to the Port D pin 0 (via a 330Ω resistor in series) of the Atmega8 microcontroller.

### 2.1.2 Code

**Turning on LED indefinitely**

```
#include <avr/io.h>

int main(void)
{
        DDRD = 0x01; // Direction
        PORTD = 0x01; // State
}
```

### 2.1.3 Circuit Connections

Refer to the base circuit implementation shown in 1

- Connect the anode of an LED to PD0 (pin 2 on ATmega8) and the cathode is grounded.
- Ensure to connect 330Ω resistors (or other suitable values) between LED's cathode and the ground to limit current through the LED.

## 2.2 Making LED Blink

### 2.2.1 Description

We extend the previous design to make the LED blink repeatedly with about a one second gap.

### 2.2.2 Code

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
        DDRD = 0x01;
        while(1){
                PORTD = 0x01;
                _delay_ms(1000);
                PORTD = 0x00;
                _delay_ms(1000);
        }
}
```

### 2.2.3 Circuit Connections

Refer to the base circuit implementation shown in 1

- Connect the anode of an LED to PD0 (pin 2 on ATmega8) and the cathode is grounded.
- Ensure to connect 330Ω resistors (or other suitable values) between LED's cathode and the ground to limit current through the LED.

## 2.3 Making LEDs blink alternately

### 2.3.1 Description

Further, we connect one more LED to port D (via a 300Ω resistor) and attempt to make the two LEDs blink alternately.

### 2.3.2 Code

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
        DDRD = 0x03;
        while(1){
                PORTD = 0x02;
                _delay_ms(1000);
                PORTD = 0x01;
                _delay_ms(1000);
        }
}
```

### 2.3.3 Circuit Connections

Refer to the base circuit implementation shown in 1

- Connect the anodes of two LEDs to PD0 and PD1 (pins 2 and 3 on ATmega8) and the cathodes are grounded.
- Ensure to connect 300Ω resistors (or other suitable values) between LED cathodes and the ground to limit current through the LEDs.

## 2.4 Two-bit Standard Counter

### 2.4.1 Description

We desire to output a sequence of 00, 01, 10, 11 and repeat using two LEDs. We implement the logic in C and realise it on a circuit.

### 2.4.2 Code

**Two-bit Standard Counter**

```
#include <avr/io.h>
#include <util/delay.h>
int main(void)
{
        DDRD = 0x03;
        while(1){
                PORTD = 0x00;
                _delay_ms(1000);
                PORTD = 0x01;
                _delay_ms(1000);
                PORTD = 0x02;
                _delay_ms(1000);
                PORTD = 0x03;
                _delay_ms(1000);
        }
}
```

We choose PORTD and, further, the first two pins to interface with the LEDs. Therefore, we set the corresponding pins in the Data Direction Register, DDRD as 1, ie, assigning a hexadecimal value of 3. For counting, we modify the PORTD pins to indicate the appropriate count and utilize the *util/delay.h* library to provide appropriate delays between the counts. The count sequence goes as 0, 1, 2, 3 and repeat. We set the first two pins of PORTD with the corresponding hexadecimal numbers between the delays. Additionally, this is enclosed within an indefinite loop to allow the program to run forever.

### 2.4.3 Circuit Connections

Refer to the base circuit implementation shown in 1

- Connect the anodes of two LEDs to PD0 and PD1 (pins 2 and 3 on ATmega8) and the cathodes are grounded.
- Ensure to connect 300Ω resistors (or other suitable values) between LED cathodes and the ground to limit current through the LEDs.

## 2.5 Adding Two Numbers using Port D pins

### 2.5.1 Description

The goal for this task in the assignment is to hard code a pair of 2-bit numbers in the program and perform addition of the numbers. The results are then shown on the three LEDs connected to PortD pins PD0,PD1,PD2 of the Atmega8.

### 2.5.2 Code

```
#include <avr/io.h>
#define NUM1 0b11
#define NUM2 0b10

int main(void)
{
        DDRD = 0x07;
        PORTD = 0x00;
        uint8_t RESULT = NUM1 + NUM2;
        RESULT &= 0x07;
        PORTD = RESULT;
}
```

- `#define NUM1 0b11` and `#define NUM2 0b10`: Define two 2-bit numbers: $NUM1 = 3$, $NUM2 = 2$.
- `DDRD = 0x07`: Set pins PD0, PD1, PD2 of PORTD as output.
- `PORTD = 0x00`: Initialize PORTD to turn off all LEDs.
- `RESULT = NUM1 + NUM2`: Add the two numbers ($3 + 2 = 5$ or `0b101` in binary).
- `RESULT &= 0x07`: Mask the result to the lowest 3 bits.
- `PORTD = RESULT`: Display the result on PD0, PD1, and PD2.

### 2.5.3 Circuit Connections

- Connect the anodes of three LEDs to PD0, PD1, PD2 (pins 2, 3, 4 on ATmega8).
- Use 300Ω resistors between LED cathodes and ground to limit current.

## 2.6 Three-bit Johnson Counter

### 2.6.1 Description

A Johnson Counter is a special type of modified ring counter in that it can count twice as much as a standard ring counter does. Detailed theory and a verilog implementation of a three-bit Johnson Counter was done in Assignment 2 and can be found here. In this assignment, we implement the same in C and realise it on a circuit.

### 2.6.2 Code

**Three-bit Johnson Counter**

```
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
        DDRD = 0x07;
        while(1){
                PORTD = 0x00;
                _delay_ms(1000);
                PORTD = 0x04;
                _delay_ms(1000);
                PORTD = 0x06;
                _delay_ms(1000);
```

```
                PORTD = 0x07;
                _delay_ms(1000);
                PORTD = 0x03;
                _delay_ms(1000);
                PORTD = 0x01;
                _delay_ms(1000);
        }
}
```

Similar to the logic for a standard counter, we choose PORTD and, further, the first three pins to interface with the LEDs. Therefore, we set the corresponding pins in the Data Direction Register, DDRD as 1, ie, assigning a hexadecimal value of 7. For counting, we modify the PORTD pins to indicate the appropriate count and utilize the *util/delay.h* library to provide appropriate delays between the counts. The count sequence goes as 0, 4, 6, 7, 3, 1 and repeat. We set the first three pins of PORTD with the corresponding hexadecimal numbers between the delays. Additionally, this is enclosed within an indefinite loop to allow the program to run forever.

### 2.6.3 Circuit Connections

Refer to the base circuit implementation shown in 1

- Connect the anodes of three LEDs to PD0, PD1 and PD2 (pins 2, 3 and 4 on ATmega8) and the cathodes are grounded.
- Ensure to connect 300Ω resistors (or other suitable values) between LED cathodes and the ground to limit current through the LEDs.

## 2.7 Addition of 2-bit numbers using PortC pins

### 2.7.1 Description

The goal for this task is to hard code a pair of 2-bit numbers in the program and perform addition of the numbers. The results are then shown on the three LEDs connected to the bread board.

### 2.7.2 Code

```
#include <avr/io.h>
#define NUM1 0b11
#define NUM2 0b10
//DDRC must turn on pins 3,4,5
//DDRC = 0b00111000
int main(void)
{
        DDRC = 0b00111000;
        PORTC = 0x00;
        uint8_t RESULT = NUM1 + NUM2;
        RESULT<<=3;
        RESULT &= 0x38;
        PORTC = RESULT;
}
```

### 2.7.3 Circuit Connections

- Connect the anodes of three LEDs to PC3, PC4, PC5 (pins 23, 24, 25 on ATmega8).
- Use 300Ω resistors between LED cathodes and ground to limit current.

## 2.8 Three-bit Johnson Counter using PortC pins

### 2.8.1 Description

The goal for this task in the assignment is to implement a Three-bit Johnson Counter in C using PortC pins. It uses the same concept as that from the previous task which used PortD pins. The results are then shown on the three LEDs connected to the bread board.

### 2.8.2 Code

```
#include <avr/io.h>
#include <util/delay.h>
//use pins 3,4,5 of PORTC
//DDRC = 0b00111000, shifted 0x07 by 3
//DDRC = 0x38
int main(void)
{
        DDRC = 0b00111000;
        while(1){
                PORTC = 0b00000000;
                _delay_ms(1000);
                PORTC = 0b00100000;
                _delay_ms(1000);
                PORTC = 0b00110000;
                _delay_ms(1000);
                PORTC = 0b00111000;
                _delay_ms(1000);
                PORTC = 0b00011000;
                _delay_ms(1000);
                PORTC = 0b00001000;
                _delay_ms(1000);
        }
}
```

### 2.8.3 Circuit Connections

Refer to the base circuit implementation shown in 1

- Connect the anodes of three LEDs to PC0, PC1 and PC2 (pins 23, 24 and 25 on ATmega8) and the cathodes are grounded.
- Ensure to connect 300Ω resistors (or other suitable values) between LED cathodes and the ground to limit current through the LEDs.

## 2.9 Adding 2 four bit numbers using PORTC pins

### 2.9.1 Description

The goal for this task in the assignment is to hard code a pair of 4-bit numbers in the program and perform addition of the numbers. The results are then shown on the five LEDs connected to PortC pins PC0 to PC4 of the Atmega8.

### 2.9.2 Code

```
#include <avr/io.h>
#define NUM1 0b0100
#define NUM2 0b0010
```

```
int main(void)
{
        DDRC = 0x1F;
        PORTC = 0x00;
        uint8_t RESULT = NUM1 + NUM2;
        RESULT &= 0x1F;
        PORTC = RESULT;
}
```

- `#define NUM1 0b0100` and `#define NUM2 0b0010`: Define two numbers: $NUM1 = 4$, $NUM2 = 2$.
- `DDRC = 0x1F`: Set pins PC0 to PC4 of PORTC as output.
- `PORTC = 0x00`: Initialize PORTC to turn off all outputs.
- `RESULT = NUM1 + NUM2`: Add the two numbers ($4 + 2 = 6$ or `0b110` in binary).
- `RESULT &= 0x1F`: Mask the result to the lowest 5 bits.
- `PORTC = RESULT`: Display the result on PC0 to PC4.

#### 2.9.3 Circuit Connections

- Connect the anodes of five LEDs to PC0, PC1, PC2, PC3, and PC4 (pins 7 to 11 on ATmega8).
- Use 300Ω resistors between LED cathodes and ground to limit current.

# 3 Implementation

This section outlines the steps to develop, compile, and load a C program onto the **ATmega8** microcontroller using **Microchip Studio**, **USBASP**, and **Burn-o-Mat**.

## 3.1 Compilation in Microchip Studio

We begin by writing the C code in **Microchip Studio**, followed by compiling it into a `.hex` file, which will be used to program the microcontroller.

- Open **Microchip Studio** and create a new **GCC C Executable Project**.
- Select **ATmega8** as the target microcontroller.
- Write the desired C program in `main.c` and save it.
- Go to **Build** and select **Build Solution** to generate a `.hex` file.
- The `.hex` file is a binary representation of the compiled code required to load onto the ATmega8's flash memory.

## 3.2 Utility of USBASP and Burn-o-Mat

After compiling the C program into a `.hex` file, the next step is to load it onto the ATmega8 microcontroller using the **USBASP** programmer and **Burn-o-Mat** software.

- **USBASP** is an in-circuit serial programmer (ISP) used to transfer the `.hex` file onto the ATmega8's flash memory.
- It enables programming via **USB** without needing a bootloader, simplifying code uploads to AVR microcontrollers.

- The programmer connects to the microcontroller through specific pins like **MISO**, **MOSI**, **SCK**, **RESET**, **VCC**, and **GND**, allowing direct communication with the microcontroller's memory.
- USBASP sends data from the computer to the microcontroller in binary form. It uses a specific protocol (SPI) to write data bit by bit into the microcontroller's flash memory.
- **Burn-o-Mat** is a graphical user interface for `avrdude`, the programming tool for AVR microcontrollers.
- It simplifies the process of uploading the `.hex` file by allowing easy configuration of settings such as the **programmer type** (USBASP) and the **port** (USB).

### 3.3 Loading the Hex File onto ATmega8 Using Burn-o-Mat

Once the **USBASP** and **Burn-o-Mat** are configured, the final step is to upload the compiled `.hex` file to the ATmega8.

- Connect the ATmega8 to the USBASP programmer as per the circuit diagram.
- Open **Burn-o-Mat** and navigate to the **Settings**.
- Set the **Port** to **USB** and the **Programmer** to **USBASP**.
- Select the `.hex` file from the project folder in **Microchip Studio**.
- Click **Write Flash** to burn the hex file to the ATmega8's flash memory.

This completes the process of programming the ATmega8 microcontroller.

## 4 Observation

### 4.1 Turning on LED indefinitely

- The program compiles without errors in Microchip Studio.
- In the I/O window, the PORTD register is set to 0x01, indicating that only the LED connected to PD0 is turned on, while PD1 remains off.
- On the breadboard, the LED connected to PD0 is illuminated, while the LED connected to PD1 is off, confirming that PD0 is configured as an output and set to a high state.

### 4.2 Making an LED blink

- The LED connected to PD0 turns on for 1 second and then turns off for 1 second, creating a visible blinking effect.
- In the I/O window, the PORTD register toggles between 0x01 (LED on) and 0x00 (LED off) every second, reflecting the blinking pattern of the LED.
- The PIND register shows the current state of the pins on Port D, with PD0 reflecting the output state (high or low) as the LED blinks.
- During debugging, it is advisable to comment out `_delay_ms()` before stepping into instructions to observe the toggling of PORTD output and the state of PIND in the I/O window more clearly.

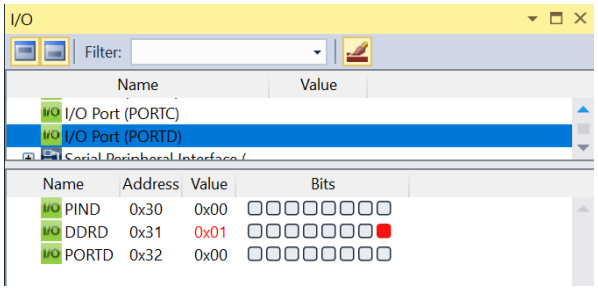


Figure 2: Make PD0 output

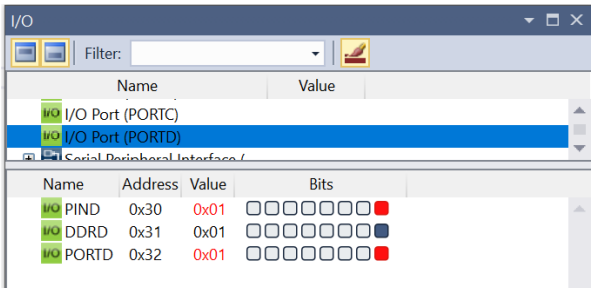


Figure 3: Make PD0 high

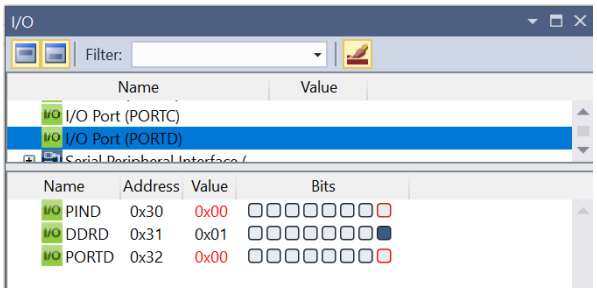


Figure 4: Make PD0 low

## 4.3 Making LEDs blink alternately

- On the breadboard, two LEDs connected to PD0 and PD1 alternately blink every 1 second as expected.
- During debugging, the output alternates between PD0 and PD1, with one LED turning on while the other turns off.
- The setup demonstrates basic control of multiple output pins and timing using the `_delay_ms()` function.

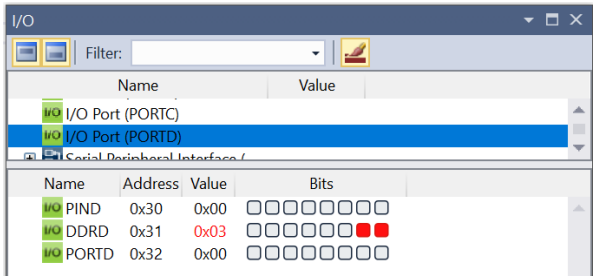


Figure 5: Make both PD0 and PD1 outputs

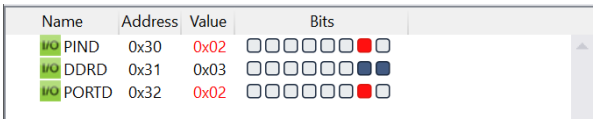


Figure 6: Turn on second LED

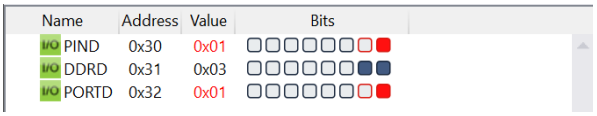


Figure 7: Turn off second LED, turn on first

## 4.4 Two-bit Standard Counter

- In the I/O window, the PORTD register toggles through the states (00, 01, 10, 11), indicating the output corresponding to the LEDs connected to PD0 and PD1.
- On the breadboard, two LEDs connected to PD0 and PD1 blink in sequence, displaying a 2-bit counter pattern (00, 01, 10, 11) with a 1-second interval for each state, confirming the correct operation of the circuit.

## 4.5 Three-bit Standard Counter

- In the I/O window, we observe the PORTD register cycling through six distinct states, corresponding to the 3-bit Johnson counter sequence.
- The I/O View window tracks the toggling of PD0, PD1, and PD2 as they progress through the Johnson counter pattern.
- On the breadboard, three LEDs connected to PD0, PD1, and PD2 follow the expected Johnson counter sequence, with each state displayed for 1 second, confirming the correct operation of the circuit.

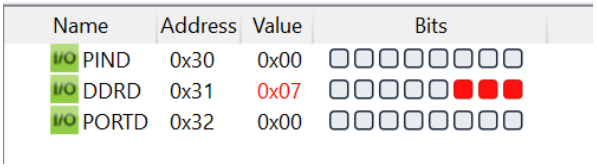


Figure 8: Make PD0, PD1, PD2 outputs

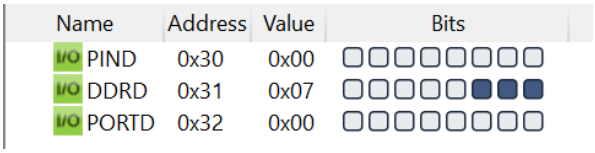


Figure 9: State 0: PD0, PD1, PD2 are all low (000).

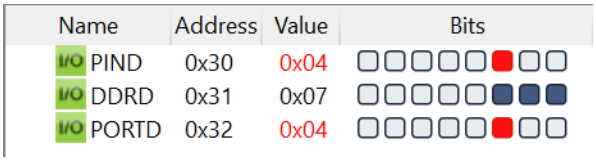


Figure 10: State 4: PD0 is low, PD1 is low, PD2 is high (100).

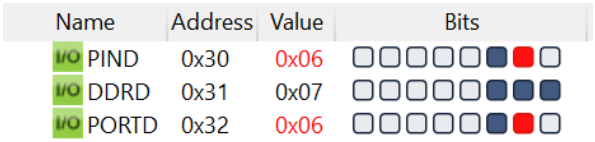


Figure 11: State 6: PD0 is low, PD1 is high, PD2 is high (110).

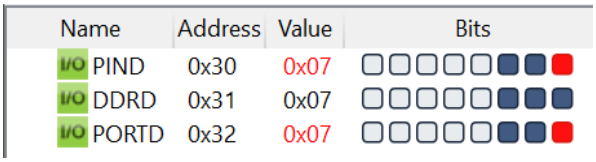


Figure 12: State 7: PD0, PD1, and PD2 are all high (111).

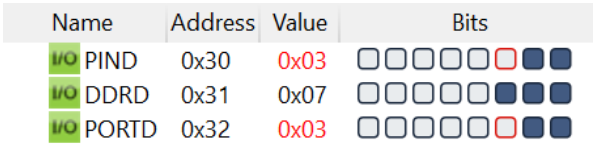


Figure 13: State 3: PD0 and PD1 are high, PD2 is low (011).

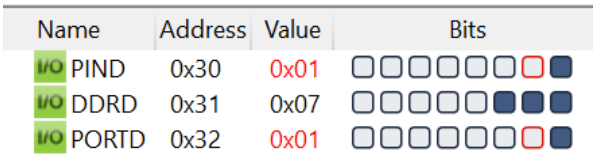


Figure 14: State 1: PD0 is high, PD1 and PD2 are low (001).

## 4.6 Addition

- In the I/O window, the PORTD register reflects the current output state. The output pins (PD0, PD1, PD2 for two bit addition)(PC0,PC1,PC2,PC3,PC4 for four bit addition)(similar analogues for addition using PORTC pins) indicate the results of the calculations or operations executed in the code.

- On the breadboard, the LEDs connected to the output pins illustrate the output or sum.

# 5 Conclusion

The experiment successfully demonstrated the programming of an AVR ATmega8 microcontroller in C language by attempting various tasks such as making multiple LEDs blink simultaneously and showing the output of counters and addition using the same. All the code written for this assignment can also be found here