# Peripheral I/O Handling on Atmel Atmega8 using AVR Assembly Language

EE2016 Microprocessors

Sanjeev Subrahmaniyan S B

EE23B102

**Abstract**

The Atmel AVR microcontroller series are a RISC based 8-bit processor suited for beginners and professionals to implement small scale computational and embedded applications. In this experiment, peripheral I/O devices such as push buttons, DIP switches and LEDs are integrated with the AVR microcontroller and programmed in assembly language. The programs are simulated on microchip studio and physically tested on a breadboard with the appropriate components. The experiment enables one to gain a deeper understanding of memory mapped I/O, assembled hex file generation and programming a microcontroller as a part of an electronic circuit. The tasks undertaken in this experiment are blinking an LED with a fixed cycle, triggering and LED with a push button switch and performing addition of 4-bit numbers by taking inputs through DIP switches and displaying outputs on LEDs.

## 1 Introduction

The AVR microcontroller controls and reads from its connected peripheral I/O devices using the concept of memory mapped I/O. This enables the microcontroller to use the same hardware pins for both input and output as per needs by proper configuration. The idea is that all input and output can be interpreted as the value held in a register corresponding to that port. When the port is set to output and a value is written to its corresponding register, the hardware pin outputs the value. Similarly, input is read through the value stored in a register corresponding to the port under concern. This experiment concerns handling these I/O registers to use peripheral devices on a hardware level. The circuits are realised using the Atmel ATmega8 microcontroller and corresponding hardware on a breadboard.

# 2 Objectives

The following tasks are implemented on a breadboard using the Atmel Atmega8 microcontroller

1. Blinking an LED with a time period of 1s and duty cycle of 50 percent
2. Triggering an LED when a push button switch is pressed
3. Addition of 4-bit numbers and indication of the output on LEDs

# 3 Assembly language programs

## 3.1 Blinking LED

The objective of this experiment was to make an LED blink at a frequency of 1Hz and a duty cycle of 50 percent. This is achieved by the assembly code segment as part of this section, which is clearly explained by the comments.

The process is done by the following steps:

1. Define the register identifiers
2. Define control loop variables which are used to achieve the required delay. The values of the inner and outer loops are selected such that the delay is obtained to the maximum possible precision. The selection is explained at the end of this section.
3. Set the DDRB register such that PINB0 is configured to output, where the led will be connected with a current limiting resistor
4. Repeat the process of toggling the led, waiting for half a second using the delay loop and toggle again. This will let us obtain the required frequency and duty cycle.

```
    .include "m8def.inc"

    .def    mask    = r16       ; mask register
    .def    ledR    = r17       ; led register
    .def    oLoopR  = r18       ; outer loop register
    .def    iLoopRl = r24       ; inner loop register low
    .def    iLoopRh = r25       ; inner loop register high

    .equ    oVal    = 71        ; outer loop value
```

```
    .equ    iVal    = 1760      ; inner loop value

    .cseg
    .org    0x00
    clr ledR                ; clear led register
    ldi mask,(1<<PINB0)     ; load 00000001 into mask register
    out DDRB,mask           ; set PINB0 to output

start:  eor ledR,mask           ; toggle PINB0 in led register
    out PORTB,ledR          ; write led register to PORTB

    ldi oLoopR,oVal         ; initialize outer loop count

oLoop:  ldi iLoopRl,LOW(iVal)   ; intialize inner loop count in inner
    ldi iLoopRh,HIGH(iVal)  ; loop high and low registers

iLoop:  sbiw    iLoopRl,1       ; decrement inner loop registers
    brne    iLoop           ; branch to iLoop if iLoop registers != 0

    dec oLoopR              ; decrement outer loop register
    brne    oLoop           ; branch to oLoop if outer loop register != 0

    rjmp    start           ; jump back to start
```

A snippet from the video we recording showing the image is attached:

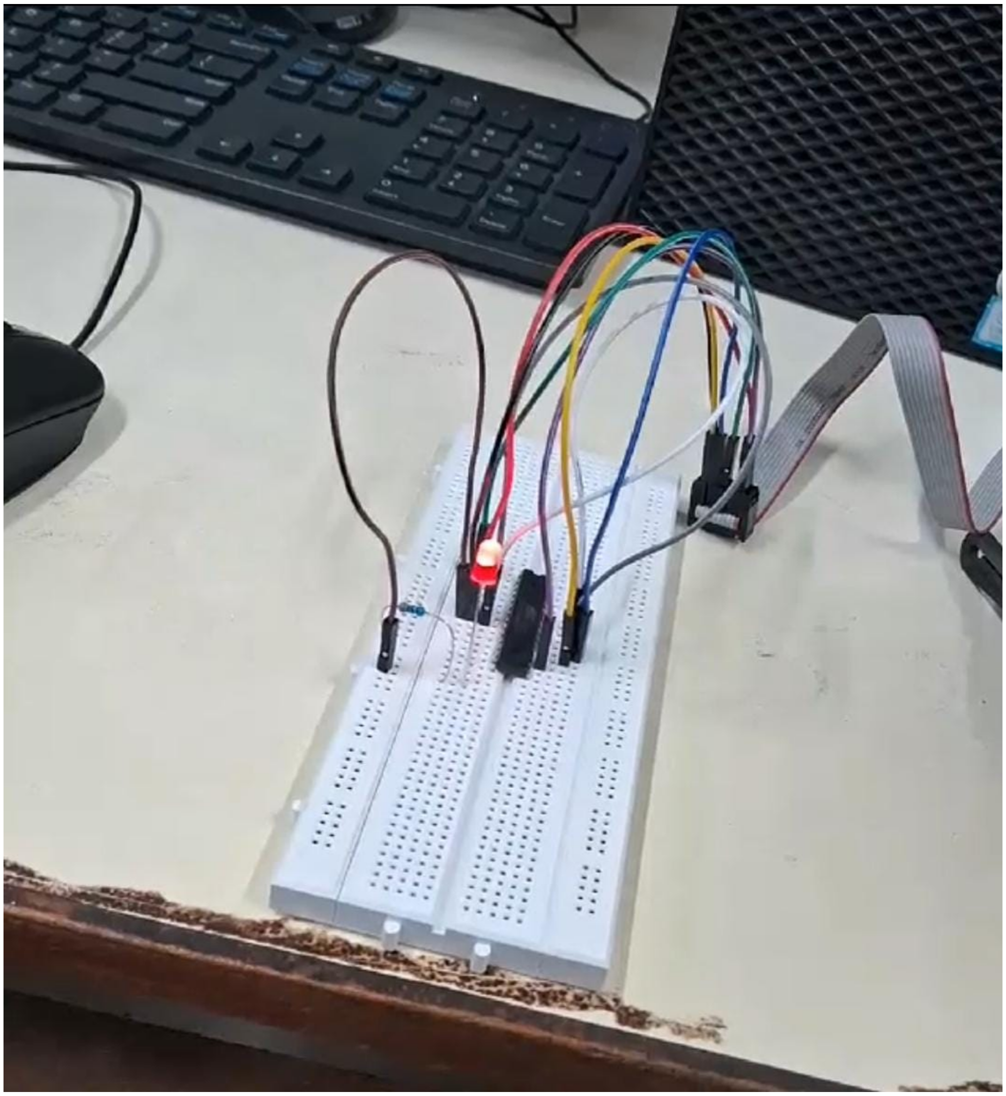

Figure 1: Blinking of an LED with 1Hz frequency

#### 3.1.1 The selection of control variables for the delay and calculation of delay

The required delay is achieved by carefully choosing the loop lengths to attain the required number of cycles. It can be clearly seen from the program that the toggling of the LED occurs at the start of the 'start' loop. This implies that the duration elapsed from the 'start' instruction to the final 'rjmp start' instruction must take 0.5 seconds.

The instructions and the time it takes to execute them:

1. eor - 1 cycle
2. out - 1 cycle

3. ldi - 1 cycle

This means the 'start' loop will take 3 cycles.

The oLoop has two ldi commands and will take 2 cycles.

The iLoop can be broken down into:

1. sbiw - 2 cycle
2. brne - 2 cycles for the not equal condition

Now because iLoopR is initialised with 1760, the first part of the start loop will run for 1759 times before the condition becomes false and it moves to the next line. This means, the number of cycles taken is

$$1759 * 4 + 1 * 2 = 7038 cycles$$

Note that the last iteration of this part takes only 2 cycles because brne takes 1 cycle when the condition is true.

The next part of the program, the dec instruction takes 1 cycle and the next brne takes 2 cycles. Because oLoop is initialised with 71, the loop will run 71 times before breaking. Note that the loop breaks out into the oLoop part of the program. This means, the time taken would be:

$$70 * (7038 + 2 + 1 + 2) + 1 * (7038 + 2 + 1 + 1) = 500052 cycles$$

Here, it is noted that the last brne instruction takes only 1 cycle. Adding two cycles for rjmp, we see that the loop takes 500054 cycles between every toggling of the led state. Using the fact that the processor operates at a frequency of 1MHz, the cycle time is 1 microsecond. This makes the approximate duration of each toggle loop = 500054 microseconds, about 0.500054 seconds, which is the closest approximate to 0.5 seconds without wasting too many lines of program memory. This lets us obtain the required frequency and duty cycle.

## 3.2 Triggering LED when push button is pressed

The second objective of this lab session was to make an LED glow when a push button is pressed and turn it off when the button is unpressed. It is achieved by relaying the value read at the input pin of the AVR microcontroller into the output pin to which the LED is connected. The working is explained as follows:

1. Configure DDRB such that PINB0 is set to input, where the push button is connected

2. Set a loop which continuously reads the value input at PINB0 and jumps to the led trigger loop when the pin reads 1. This is established using the SBIS operation, which skips the next instruction as long as the I/O register is cleared. When the push button is pressed, the PIN reads high and control jumps to the light_led funtion

3. Trigger the led by directing output to the PORTD and return control back to the checker loop

```
.include "m8def.inc"

start:
    .cseg
    .org    0x00

    ldi r16, 0x00; load pinb0 to r16
    out DDRB,r16 ;setting it to input

check_input:


    SBIS PINB, 0;0-> switch on
    rjmp light_led

    ldi r17, 0x0
    OUT PORTD, r17
    rjmp check_input

light_led:
    ldi r16, 0xFF
    OUT DDRD, r16
    ldi r17, 0x1
    OUT PORTD, r17
    rjmp check_input
```

A snippet from the video we recording showing the image is attached, where my teammate presses the button to show the led glowing:

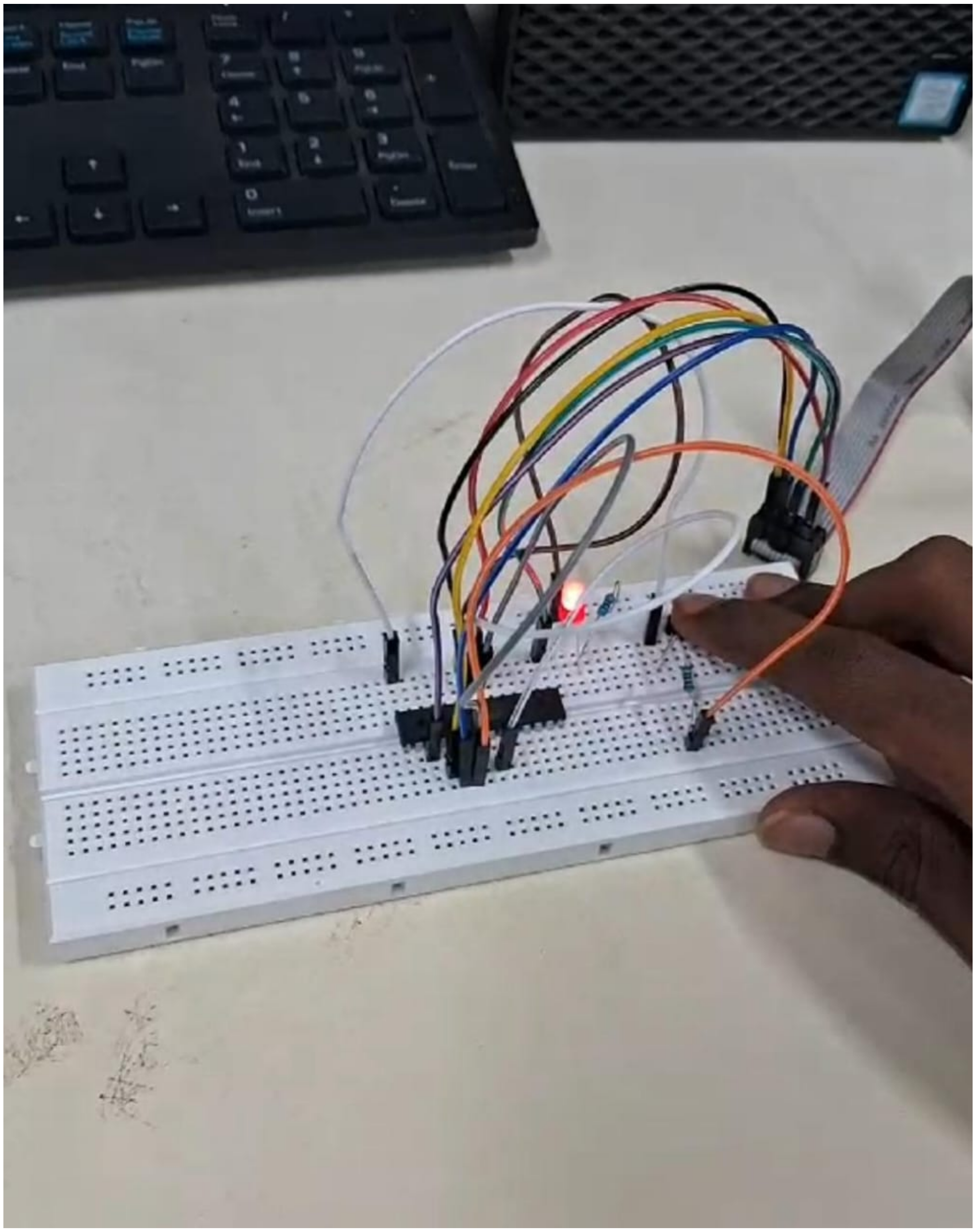

Figure 2: Triggering an LED when the button is pushed

### 3.3 Addition of 4 bit unsigned numbers

The next part of the lab was to take two 4 bit numbers as inputs through DIP hardware switches, add them and display the output using indicator LEDs. This is accomplished using the assembly program given below. The program is self explanatory, and reads the input, computes the addition with carry and displays the output on the connected LEDs.

```
.include "m8def.inc"
```

```

start:
    .cseg
    .org    0x00

    ldi r16, 0x00; load pinb0 to r16
    out DDRB,r16 ;setting it to input

check_input:


    IN r16, PINB
    COM r16
    mov r17, r16
    mov r18, r16

    ANDI R17, 0x0F
    ANDI R18, 0XF0

    LSR R18
    LSR R18
    LSR R18
    LSR R18

    add r17, r18

    ldi r16, 0xFF
    OUT DDRD, r16
    OUT PORTD, r17
    rjmp check_input
```

A snippet from the video we recording showing the image is attached:

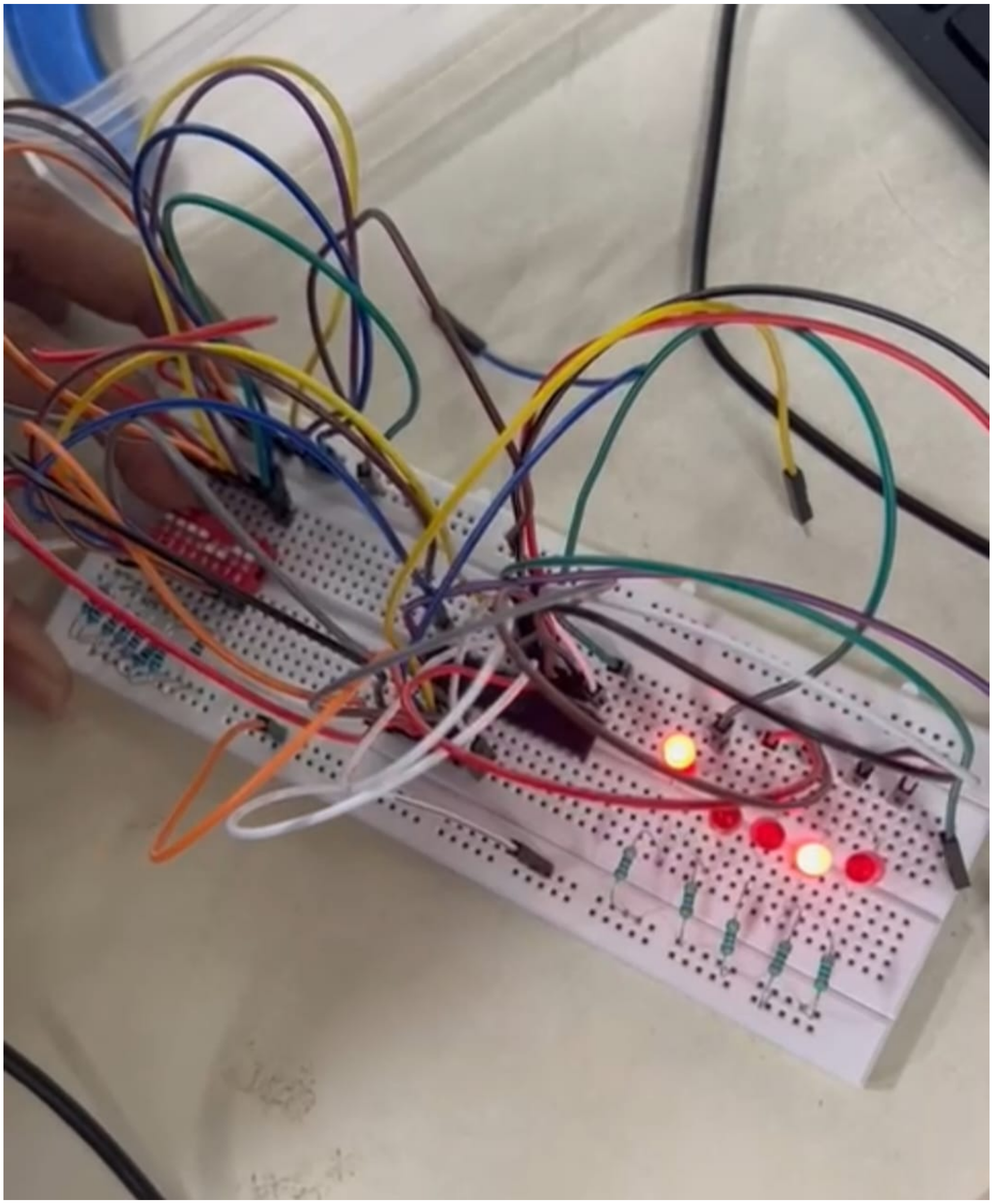

Figure 3: Addition of 4 bit numbers and displaying output on connected LEDs

As can be seen in the left side of the picture, the inputs given are 15 and 3 in the DIP switch inputs, leading to a sum of 18 as shown in the led, as read from left to right.

# 4 Procedure

1. Write the assembly program for the required functionality on microchip studio's editor
2. Verify the correct working of the program by monitoring the registers, memory and

clock during program execution

3. Wire the circuit on a breadboard with the microcontroller and other components
4. Generate the .hex file for the assembly program after debugging
5. Connect the microcontroller to the PC and burn the generated .hex file using a cable appropriately connected
6. Test the circuit for different input states and report them

# 5 Conclusion

This experiment involved writing assembly program and building them into .hex machine code files. These were then flashed into the microcontroller's memory using a burner software. The microcontroller was interfaced with a number of hardware peripherals such as LEDs, push buttons and DIP switches. The experiment was useful to learn the concepts of memory mapped I/O as implemented in AVR and programming a board through a programmer. The lab session was useful to gain experience in implementing simple circuitry using hardware peripherals and microcontroller.