EE2016: Microprocessers Lab - Assignment 6 Report

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

September 27, 2024

Abstract

This is a detailed report on a assignment based on AVR Interrupt Handling 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 a RED LED on interrupt and (ii) extending the previous circuit to glow a white LED if no interrupt. We write the program in AVR assembly for (i) and in both assembly as well as C for (ii).

1.1 Interrupt Service Routine

An Interrupt Service Routine (ISR) is akin to a subroutine in programming, designed to handle specific tasks triggered by hardware events. In programming, a subroutine is a sequence of instructions that can be called multiple times, requiring the processor to save certain information, such as the program counter and registers, to ensure proper execution. Similarly, when a hardware device, like a keyboard or a button, generates an interrupt, the processor saves its current state before executing the ISR to address the device's request. Once the ISR completes its task, the processor restores its saved state, resuming operation from where it was interrupted. This mechanism of control transfer via ISRs underscores the importance of efficiently managing tasks within a microprocessor environment.

1.2 Assembly language for Interrupt Handling

In some cases, low-level hardware control and optimization are critical. Assembly language provides a more direct interaction with the hardware, allowing programmers to manage specific tasks, such as stack pointer setup and register configurations, with precise control. Unlike C, which abstracts many low-level details as can be seen in differences in the codes of 2.1 and 2.3 in the Design section, assembly enables us to optimize performance and resource utilization by writing code tailored to the specific architecture of the microprocessor. This is particularly beneficial for applications requiring real-time responses, such as interrupt handling, where managing the PORTx and DDRx registers is essential for enabling interrupts and ensuring accurate hardware interactions.

2 Design

2.1 A simple C program on Interrupt Handling

2.1.1 Description

This is a simple interrupt program in C to keep a white LED on, and in case of an interrupt, turn it off and blink a red LED indefinitely.

2.1.2 Code

```
#define F_CPU 8000000UL
#include <avr/io.h>
#include<util/delay.h>
#include <avr/interrupt.h>
int main(void)
{
DDRB=0x03;
DDRD=0x00;
GICR=0x40;
SREG=0x80;
while(1)
{
PORTB=0x01;
}
}
ISR(INT0_vect)
{
cli();
1
PORTB=0x02;
_delay_ms(100);
PORTB=0x00;
_delay_ms(100);
sei();
}
```

- The line #define F_CPU 8000000UL sets the microcontroller clock speed (UL denotes unsigned long; we set 8 MHz here).
- We use PORTB for ouput and setting DDRB to 0x03 allows output to a white LED as well as a red LED. Note that both of these are connected to Port B.
- We use PORTD for the interrupt and DDRD is set to 0 since the switch is an input device.
- The General Interrupt Control Register (GICR) is an 8-bit register.
- Our interest is in INT0; hence we set "6 bit" (7th from right end) to 1(as you can infer from 1) .

Figure 1: Structure of GICR

- The remaining bits are set to 0 (note that IVSEL and IVCE correspond to interrupt vector select and interrupt vector change enable, and these are set to 0 to allow interrupts).
- Setting the status register SREG to 0x80 corresponds to global interrupt enable as can be seen from 2.

Figure 2: Structure of Status Register(SREG)

- Within the ISR, we note that CLI clears the global interrupt flag (I) in the status register (SREG) and thereby disables the interrupts.
- No interrupt will be executed after the CLI instruction, even if it occurs simultaneously with CLI.
- However, while coming out of the ISR, we have an SEI which sets the global interrupt flag (I) to 1.
  Note: Unlike function (or subroutine) calls, there is no explicit call to the interrupt inside the main function. The transfer of control takes place whenever there is a key press.
- The main loop keeps the white LED on, it uses a while loop to continuously assign PORTB = 0x01; thus the program doesn't finish executing and pop off the stack, instead it stays in the indefinite loop. Thus we can trigger the interrupt routine by sending an input through INT0. The ISR remains defined in the context of the program alone and we use the loop to keep the program running.
- On pressing the push button, control is transferred to the ISR, which turns off the white LED and then indefinitely blinks the red LED by turning it on and off with delays.

2.1.3 Circuit Connections

- Make the connections of the USBASP as shown in 3, and connect accordingly onto the bread-board.
- Connect the anode of the white LED to PB0 (pin 14 on ATmega8).
- Connect the anode of the red LED to PB1 (pin 15 on ATmega8).
- The cathode of the white and the red LED should be grounded through a 330Ω resistor (or a suitable value) to limit current and protect the LED.
- Connect one terminal of the push button to PD2 (pin 4 on ATmega8).
- The other terminal of the push button should be grounded.
- Additionally, place a 10kΩ resistor between the ground and the grounded terminal of the push button.

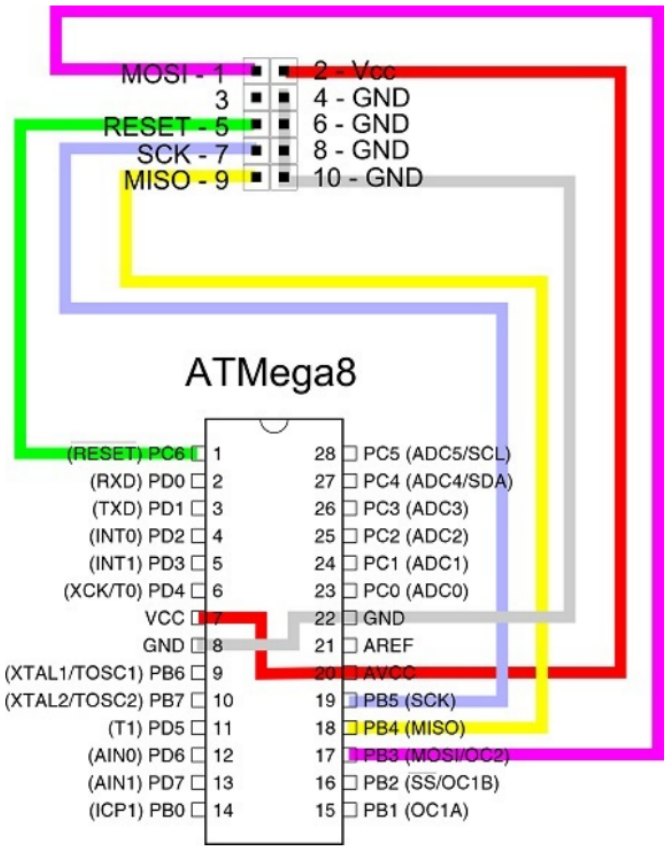

Figure 3: ATmega8 to USBASP board Connections

2.2 Blinking of a red LED on Interrupt

2.2.1 Description

We desire to write a simple interrupt program in AVR assembly to make a red LED blink 10 times on an interrupt due to a button press. The code for the Assembly program is given below.

2.2.2 Code

Blinking of a red LED on Interrupt

```
; Program to blink white LED on interrupt
.org 0
rjmp reset ; start program
.org 0x002 ; Interrupt vector address for INT1
rjmp int1_ISR
reset:
    ldi R16, 0x70 ; Setup the stack pointer to point to address 0x0070
    out spl, R16 ; Set low
    ldi  R16, 0x00
    out sph, R16 ; Set high
    ldi R16, 0x01 ; Set PB1 to output in DDR
    out DDRB, R16
    ldi R16, 0x00 ; Set PORTD to input in DDR
    out DDRD, R16
```

```
    ldi R16, 0x08 ; Use pull-up resistor for PD3
    out PORTD, R16 ; Enable pin3  pull-up resistor
    in R16, GICR
    ori R16, 0x80 ; Enable INT1 interrupt in General Interrupt Control
        Register
    out GICR, R16
    ldi R16, 0x00 ; Turn off LED
    out PORTB, R16
    sei ; Enable interrupts in SREG
indefiniteloop:
    rjmp indefiniteloop ; Main program -  do nothing (unless interrupt
       )
int1_ISR: ; INT1 interrupt handler or ISR
    cli ; Clear interrupts
    in R16, SREG ; Save status register SREG
    push R16 ; Push SREG contents onto stack
ldi R16, 10
mov R0, R16 ;  Counter for LED blinks - no. = 10
back5:
    ldi R16, 0x02 ; Turn on LED
    out PORTB, R16
delay1:
    ldi R16, 0xFF ; counter for delay
back2:
    ldi r17, 0xFF ;  counter for more delay
back1:
    dec r17
    brne back1
    dec r16
    brne back2
; Above block - delay in nested loop
ldi R16, 0x00 ; Turn off LED
out PORTB, R16
; Delay block - agin
delay2:
    ldi R16, 0xFF ; delay
back3:
    ldi r17, 0xFF
back4:
    dec r17
    brne back4
    dec R16
    brne back3
    dec R0 ; Dec blink count
brne back5 ; Check if LED has blinked 10 times
pop R16 ; Retrieve status register from stack to R16
out SREG, R16 ; Restore SREG
rjmp indefiniteloop ; Go back to main program
```

The logic behind the above code is as follows.

- Set the appropriate origin for Reset and Interrupt Vector Address for INT1.
- In the reset section, set up the stack pointer to a location (say, 0x70) which will be used to store the current SREG on interrupt. We shall use the first pin of PORTB for the output red LED and PD3 (INT1) for the button interrupt. Configure DDRB for output, DDRD for input, set PORTB to 0x00 (LED is off initially), and enable the PD3 pull-up resistor for the interrupt.

Since we desire to use INT1, enable its corresponding pin in the GICR (General Interrupt Control Register).

- The main program is an indefinite loop of nothing because our interests lie in the interrupt.
- When an interrupt occurs, the code enters the Interrupt section (`int1_ISR`). First, clear the interrupt flag and save the SREG onto the stack. Then, blink the red LED 10 times.
- We use a counter to characterize the delays, and since a register can hold a value no more than 0xFF, we define a nested loop for counting to increase the delay to an appropriate value.
- Once the LED blinks 10 times, we retrieve the saved SREG from the stack and return to the main program.

2.2.3 Circuit Connections

Refer to the base circuit implementation shown in 3

- Connect the anode of a red LED to PB0 (pin 14 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.
- Connect one terminal of a push button to PD3 (pin 5 on ATmega8) and the other terminal is grounded.
- Also ensure to connect a higher resistance value (say, 10000 Ω) between the ground and the latter terminal of the push button.

2.3 Extending the previous circuit include white LED

2.3.1 Description

We desire to rewrite the C program of task 1 in Assembly, ie, to keep a white LED on, and in case of an interrupt, turn it off and blink a red LED ten times.

2.3.2 Code

```
; Program to blink red LED on interrupt

.org 0
rjmp reset ; start program
.org 0x002 ; Interrupt vector address for INT1
rjmp int1_ISR

reset:
    ldi R16, 0x70 ; Setup the stack pointer to point to address 0x0070
    out spl, R16 ; Set low
    ldi  R16, 0x00
    out sph, R16 ; Set high
    ldi R16, 0x03 ; Set PB1, PB0 to output in DDR
    out DDRB, R16
    ldi R16, 0x00 ; Set PORTD to input in DDR
    out DDRD, R16
    ldi R16, 0x08 ; Use pull-up resistor for PD3
    out PORTD, R16 ; Enable pin3  pull-up resistor

    in R16, GICR
    ori R16, 0x80 ; Enable INT1 interrupt in General Interrupt Control
        Register
```

```
    out GICR, R16

    ldi R16, 0x00 ; Turn off red and white LED
    out PORTB, R16

    sei ; Enable interrupts in SREG

indefiniteloop:
    ldi R16, 0x01
    out PORTB, R16 ; Turn on white LED
    rjmp indefiniteloop ; Main program -  do nothing (unless interrupt
       )

int1_ISR: ; INT1 interrupt handler or ISR
    cli ; Clear interrupts
    in R16, SREG ; Save status register SREG
    push R16 ; Push SREG contents onto stack

ldi R16, 0x00
out PORTB, R16 ; Explicitly turn of white LED on interrupt
ldi R16, 10
mov R0, R16 ;  Counter for LED blinks - no. = 10

back5:
    ldi R16, 0x02 ; Turn on red LED
    out PORTB, R16

delay1:
    ldi R16, 0xFF ; counter for delay
back2:
    ldi r17, 0xFF ;  counter for more delay
back1:
    dec r17
    brne back1
    dec r16
    brne back2

; Above block - delay in nested loop

ldi R16, 0x00 ; Turn off red LED
out PORTB, R16

; Delay block - agin
delay2:
    ldi R16, 0xFF ; delay
back3:
    ldi r17, 0xFF
back4:
    dec r17
    brne back4
    dec R16
    brne back3
    dec R0 ; Dec blink count
brne back5 ; Check if red LED has blinked 10 times
pop R16 ; Retrieve status register from stack to R16
out SREG, R16 ; Restore SREG
```

```
rjmp indefiniteloop ; Go back to main program
```

The code is essentially the same as that for the task 2, modifying it to add a white LED as well. We use the second bit of PORTB for the white LED and initialize it similar to that of the red LED. We update the indefinite loop and set white LED to be switched on. Upon interrupt, we switch the white LED off before starting to blink the red LED.

2.3.3 Circuit Connections

- To the circuit implemented in the previous task, connect the anode of the white LED to PB0 (pin 14 on ATmega8).
- The cathode of the white LED should be grounded through a 330Ω resistor (or a suitable value) to limit current and protect the LED.

3 Implementation

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

3.1 Compilation in Microchip Studio

We begin by writing the 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 **Project**.
- Select **ATmega8** as the target microcontroller.
- Write the desired C (Assembly) program in `main.c`(`.asm`) 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 Loading the Hex File onto ATmega8 Using Burn-o-Mat

After compiling the C (Assembly) 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.

- 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 Task 1

- After powering the circuit by writing to the flash of the ATmega8 chip, the white LED (connected to PB0) remains on permanently.
- Upon pressing the push button connected to PD3 (which triggers the INT1 interrupt), the white LED turns off immediately, and the red LED (connected to PB1) starts blinking. The red LED blinks indefinitely, with a visible delay between each blink, indicating the program's delay routine is functioning correctly.

  Note: We notice the importance of the while loop in the main program. Without the loop, the program would simply assign PORTB = 0x01, thereby turning on the white LED then finishes executing, and there would be no response to triggering the interrupt pins. From this we observe that the interrupt routine is only defined within the context of the program. Ensure that there is an indefinite loop to keep the program running.

4.2 Task 2

- After powering the circuit and writing to the flash of the ATmega8 chip, the red LED(connected to PB0) is switched off initially.
- Upon pressing the push button connected to PD3 (which triggers the INT1 interrupt), the red LED (connected to PB0) starts blinking. It blinks exactly 10 times, with a visible delay between each blink, indicating the program's delay routine is functioning correctly.
- Once the red LED completes 10 blinks, it returns the system to its initial state. The system then waits for the next interrupt. Pushing the push button again will cause this process to repeat.

4.3 Task 3

- We observe the same results as in the case of task 1, since, this is the same C program but written in Assembly. The white LED remains switched on indefinitely until a press of a push button. Upon the press, the white LED turns off and a red LED blinks 10 times. Once done, the white LED turns back on again returning the system to its initial state.

5 Conclusion

The experiment successfully demonstrated the programming of an AVR ATmega8 microcontroller in C and assembly by attempting various tasks such as making an LED glow and interrupting it using a switch to make another LED blink. All the code written for this assignment can also be found here.