National University
Of Computer and Emerging Sciences

Hardware Lab Session Report:

Name – **Muhammad Taha**

Roll NO – **23p-0559**

SECTION – **BCS(3A)**

Subject - **Computer Organization and Assembly Language (LAB)**

Introduction:

In our Computer Organization and Assembly Language (COAL) course, we had a hands-on lab session where we learned to program and simulate using the Microchip Studio development environment and the ATmega328P microcontroller. This session was a key step in our understanding of microcontroller programming, giving us practical experience along with the theory we had learned. In this report, I will explain the main concepts, the features of the ATmega328P chip, programming techniques, memory structures, how registers work, and the simulations we performed during the lab.

Understanding Microchip Studio and Microcontroller Programming:

The lab session started with an introduction to **Microchip Studio**, a comprehensive integrated development environment (IDE) used for programming microcontrollers in assembly language or C. This environment allows us to write, compile, and simulate assembly code, making it a crucial tool for embedded systems development.

We were introduced to the **ATmega328P** microcontroller, a widely used chip in embedded systems and one that powers the famous **Arduino** boards. The ATmega328P is an 8-bit microcontroller from the AVR family and is preferred due to its versatility, low power consumption, and ability to handle complex tasks despite its small size. One of the first things we learned about the chip was its **memory structure**:

1. **Flash Memory (Program Memory):** The ATmega328P comes with 32KB of flash memory, where the user's code is stored. This memory is divided into sections, including the *bootloader* section that helps in programming the chip without external hardware.
2. **Data Memory (SRAM):** It has 2KB (2048 bytes) of SRAM used for storing data during the chip's operation.
3. **EEPROM:** This non-volatile memory of 1KB (1024 bytes) is used to store data that needs to persist even after the power is turned off.

The chip also has **32 general-purpose registers** used for various operations during execution, and **64 special function registers (SFRs)** that control the operation of the microcontroller's hardware peripherals.

Chip Variants and Packages:

The ATmega328P microcontroller is available in two common forms:

- **DIP (Dual Inline Package):** This form has 28 pins and is primarily used in laboratory experiments because it is easier to work with in a breadboard environment.
- **TQFP (Thin Quad Flat Package):** This form has 32 pins and is more compact, typically used in real-world applications due to its space-saving advantages.

Our lab mainly used the DIP package for experiments, which we found practical for connecting external sensors and performing basic I/O operations. However, for more advanced applications, the TQFP form is preferred because of its extra pins and reduced size.

Memory and Port Structure:

The ATmega328P is equipped with three main I/O ports:

1. **Port B**
2. **Port C**
3. **Port D**

Each port consists of 8 pins, except for **Port C**, which has only 7 pins due to limitations. Every port has three associated **registers**:

- **PINx (Input Pin Register):** Reads the current logic levels of the pins (1 for HIGH, 0 for LOW).
- **DDRx (Data Direction Register):** Configures each pin as either input (0) or output (1).
- **PORTx (Data Register):** Controls the output value sent to the pins or enables internal pull-up resistors for inputs.

For example, by setting all bits of the DDRB register to 1 (DDRB = 11111111), we configure **Port B** as an output port. Conversely, setting all bits to 0 (DDRB = 00000000) configures Port B as an input port. This simple mechanism is fundamental to interfacing external components, such as sensors or LEDs, with the microcontroller.

Clock Speed and Oscillators:

The **clock speed** of a microcontroller dictates how fast it can process data. The ATmega328P has a default clock speed of **16 MHz**, provided by an external crystal oscillator. The oscillator plays a crucial role in synchronizing the microcontroller's operations, generating pulses that dictate when instructions are executed.

To operate, the microcontroller requires a **ground (GND)**, **VSS**, and two additional pins for the crystal oscillator. These components supply pulses to the microcontroller, allowing it to perform tasks at a frequency that can be adjusted to suit different applications.

An important concept we explored was the **blinking of LEDs**. Using the microcontroller's fast clock speed (1/16,000,000), it can blink an LED at a rate too fast for the human eye to detect. To make the blinking visible, we slow down the clock cycle using delay functions written in assembly language.beacuse the human eyes can sees the blinking upto the 20msec, For instance, writing 0xAA (10101010) to the **PORTB** register followed by 0x55 (01010101) creates a blinking effect that toggles the LEDs connected to Port B.

Input/Output Operations and Protocols:

The microcontroller uses input/output ports to communicate with external devices. For example, we experimented with an ultrasonic sensor that had four pins: VCC, GND, and two data pins that could be connected to Port C pins (C1 and C0).

When two devices communicate, they do so using predefined communication protocols. We were introduced to three major protocols during the lab:

1. SPI (Serial Peripheral Interface)
2. USART (Universal Synchronous and Asynchronous Receiver-Transmitter)
3. I2C (Inter-Integrated Circuit)

These protocols facilitate data transfer between the microcontroller and external peripherals such as sensors, displays, or other microcontrollers.

Working with Registers:

Registers play a critical role in microcontroller programming. Our instructor introduced us to three important registers for Port B:

1. PINB (Input Pin Register B)
2. DDRB (Data Direction Register B)
3. PORTB (Data Register B)

The PINB register is used to read input data from the pins, while the PORTB register controls the output. A practical example involved writing 0xAA (10101010) and 0x55 (01010101) to PORTB, which caused the LEDs connected to the port to blink. The blinking effect depended on the clock speed and was visible due to the slower toggling of the output bits.

Memory Management and Stack Usage:

The stack is a vital memory structure in microcontroller programming. It operates in a Last In, First Out (LIFO) manner, storing temporary data like function return addresses. The stack pointer always points to the top of the stack, and care must be taken to prevent stack overflow, which occurs when data is pushed beyond the stack's limit.

One critical aspect of microcontroller programming is avoiding direct manipulation of registers. Instead, data is first moved into a temporary register and then transferred to the final destination. For instance, when writing to the DDRB register, we first move data into a temporary register before transferring it to DDRB. This prevents errors and ensures data integrity.

Simulation and Practical Demonstration: LED Blinking and 7-Segment Display:

Towards the end of the session, we were introduced to Proteus, a powerful tool for simulating microcontroller behavior. Using Proteus, we simulated how the ATmega328P microcontroller

interacts with external devices, like LEDs and 7-segment displays. The instructor demonstrated how to simulate blinking LEDs by alternating the hexadecimal values 0xAA and 0x55, which are binary patterns of 10101010 and 01010101. This allowed us to see the LEDs turning on and off in an alternating pattern, with the speed controlled by the microcontroller's clock.

We also simulated a counting sequence on a 7-segment display. The instructor showed us how to make the microcontroller count from 0 to 9 by writing a simple assembly program. This involved sending specific codes to the 7-segment display, which updated in real-time to show the digits. These simulations helped us understand how the microcontroller interacts with hardware components.

In addition to the simulations, the instructor provided a practical demonstration using real hardware. He programmed the ATmega328P to make an LED blink by writing the hexadecimal values 0xAA and 0x55 to the microcontroller’s output port, causing the LED to alternate its blinking pattern. The blinking speed was controlled by the microcontroller's clock, and we learned how to adjust the speed by modifying the assembly code.

The instructor also demonstrated how to control a 7-segment display. By programming the ATmega328P, the microcontroller was able to display numbers from 0 to 9 in sequence on the 7-segment display, reinforcing our understanding of how microcontrollers manage external hardware using assembly code.