Introduction to Microcontrollers

INTRODUCTION TO MICROCONTROLLERS

NEHA KARDAM

OpenWA
Olympia, WA

CONTENTS

ACKNOWLEDGMENTS

The Lake Washington Institute of Technology is grateful for the critical support of the National Science Foundation (NSF) Grant #2100136. This grant has been instrumental in propelling our research and development efforts in engineering through the creation of Open Educational Resources (OER) course materials.

I would also like to extend my sincere gratitude to Katherine Kelley and Priyanka Pant. Their expertise and dedication were invaluable in this project. Their encouragement played a key role in motivating me to develop these OER materials and the accompanying pressbook.

LICENSES AND PERMISSIONS

For Further Information

If you have any questions about using the content of this pressbook or require permission beyond the scope of the Creative Commons license, please feel free to contact the author, Neha Kardam (neha.kardam@lwtech.edu) and Katherin Kelley (katherine.kelley@lwtech.edu).

ABOUT THIS PRESSBOOK

This publication, “Introduction to Microcontroller,” was developed in Summer 2024 and was created as part of the National Science Foundation Advanced Technological Education grant titled, *Creation and Modernization of Technological Education in Electronics and Welding through Open Educational Resources that are Free to Share, Use, and Revise* (NSF ATE #2100136), awarded to Lake Washington Institute of Technology in 2021. In this three-year grant, three Electronics Technology courses were updated to accessible OER materials with hybrid delivery elements where possible. Faculty Librarian Katherine Kelley, and Dean of Instruction Priyanka Pant led the project. Electronics materials were authored by Electronics Professor, Neha Kardam.

This material is based upon work supported by the National Science Foundation under Grant No. 2100136. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

ABOUT THE COURSE : INTRODUCTION TO MICROCONTROLLER

Introduction to Microcontroller is a comprehensive course designed to provide students with a solid foundation in microcontroller technology and programming. The course covers fundamental concepts of microprocessors and microcontrollers, progressing through assembly language programming, and culminating in advanced microcontroller applications.

Course Highlights:

1. Comprehensive introduction to microcontrollers and microprocessors
2. In-depth study of assembly language programming
3. Hands-on experience with PIC16F18875 and PIC16(L)F18855/75 microcontrollers
4. Practical lab assignments covering various aspects of microcontroller programming
5. Focus on both theoretical concepts and practical applications
6. Introduction to advanced features like ADC2 module and its configuration
7. Real-world project implementations including home security systems and temperature indicators

Course Structure

I. Introduction to Microcontrollers

- **Fundamentals of Microprocessors and Microcontrollers:** Learn the basic functions and architecture of microprocessors and microcontrollers.
- **Microprocessor Architecture:** Understand the internal architecture and operation of microprocessors.
- **Introduction to Microcontroller Systems:** Explore the components and functionalities of microcontroller systems.
- **Getting Started with Microcontroller Programming:** Begin programming microcontrollers with practical examples and exercises.

II. Introduction to Assembly Language

- **Fundamentals of Assembly Language:** Learn the basics of assembly language programming, including mnemonics, labels, and directives.
- **Boolean Algebra in Assembly Programming:** Understand the role of Boolean algebra in assembly programming for logical operations and bit manipulation.
- **MPLAB IDE and Assembly Programming:** Get hands-on experience with the MPLAB IDE for writing and testing assembly programs.
- **Lab Assignments:** Engage in practical exercises to reinforce learning, such as reviewing instruction sets, creating block diagrams, and testing programs on MPLAB.

III. Assembly Language Programming – I

- **8085 Instruction Set: Part 1:** Study the basic instruction set of the 8085 microprocessor, including data transfer, arithmetic, and logical instructions.
- **8085 Instruction Set: Part 2:** Delve into advanced instructions and their applications, such as branching, stack operations, and machine control.
- **Lab Assignments:** Apply your knowledge in practical labs, such as coding delay routines and blinking LEDs.

IV. Assembly Language Programming – II

- **Introduction to PIC16F18875 Instruction Set:** Learn the basic instruction set of the PIC16F18875 microcontroller.
- **Detailed PIC16F18875 Instruction Set and Programming Techniques:** Explore advanced instructions and programming techniques for the PIC16F18875.
- **Advanced Instructions and Practical Applications:** Implement advanced instructions in practical applications.
- **Lab Assignments:** Complete labs such as rotating LEDs on button presses and consistently rotating LEDs to the right.

V. Assembly Programming – III

- **Introduction to PIC16(L)F18855/75 ADC2 Module:** Understand the ADC2 module and its capabilities.
- **Detailed Configuration and Operation of the PIC16(L)F18855/75 ADC2 Module:** Learn how

to configure and operate the ADC2 module.

- **Readings Specific to Lab Assignments:** Access readings and resources tailored to lab assignments.
- **Lab Assignments:** Engage in projects such as making programs using external circuits, designing a home security system, and building a temperature indicator.

This course is ideal for students pursuing careers in embedded systems, electronics engineering, or any field involving microcontroller-based design. By the end of the course, students will have the skills to program microcontrollers, understand their internal architecture, and implement them in various applications.

Investing in Your Future Success

By choosing LWTech’s “Introduction to Microcontrollers” course, you’ll gain valuable skills such as:

- Understanding and programming microcontrollers.
- Developing and implementing assembly language programs.
- Designing and troubleshooting microcontroller-based systems.
- Communicating effectively within a technical environment.

Join Us in Shaping the Future

LWTech is dedicated to preparing students with the skills needed to excel in the ever-changing tech environment. Get started today, and discover how our *“”Introduction to Microcontrollers”* course can empower you to thrive in the automated workforce.

Visit https://www.lwtech.edu/ to learn more about LWTech’s *”Introduction to Microcontrollers”* course and how it can prepare you for a successful career in the age of automation.

LAKE WASHINGTON
INSTITUTE OF TECHNOLOGY™
LWTech.edu

PART I
INTRODUCTION TO MICROCONTROLLERS

In this chapter, we will explore the foundational concepts of microprocessors and microcontrollers. Understanding their basic functions is crucial for grasping more advanced topics in microcontroller programming and applications. Microprocessors and microcontrollers are integral to modern computing, embedded systems, and a wide range of electronic devices.

FUNDAMENTALS OF MICROPROCESSORS AND MICROCONTROLLERS

1.1 Definitions and Basic Concepts

Microprocessor: A microprocessor is a computer processor on a single integrated circuit (IC) that contains the arithmetic, logic, and control circuitry required to perform the functions of a computer's central processing unit (CPU). It is designed to execute a set of instructions to perform tasks such as arithmetic operations, data movement, and control operations. The microprocessor is the brain of a computer system, capable of fetching, decoding, and executing instructions from memory.

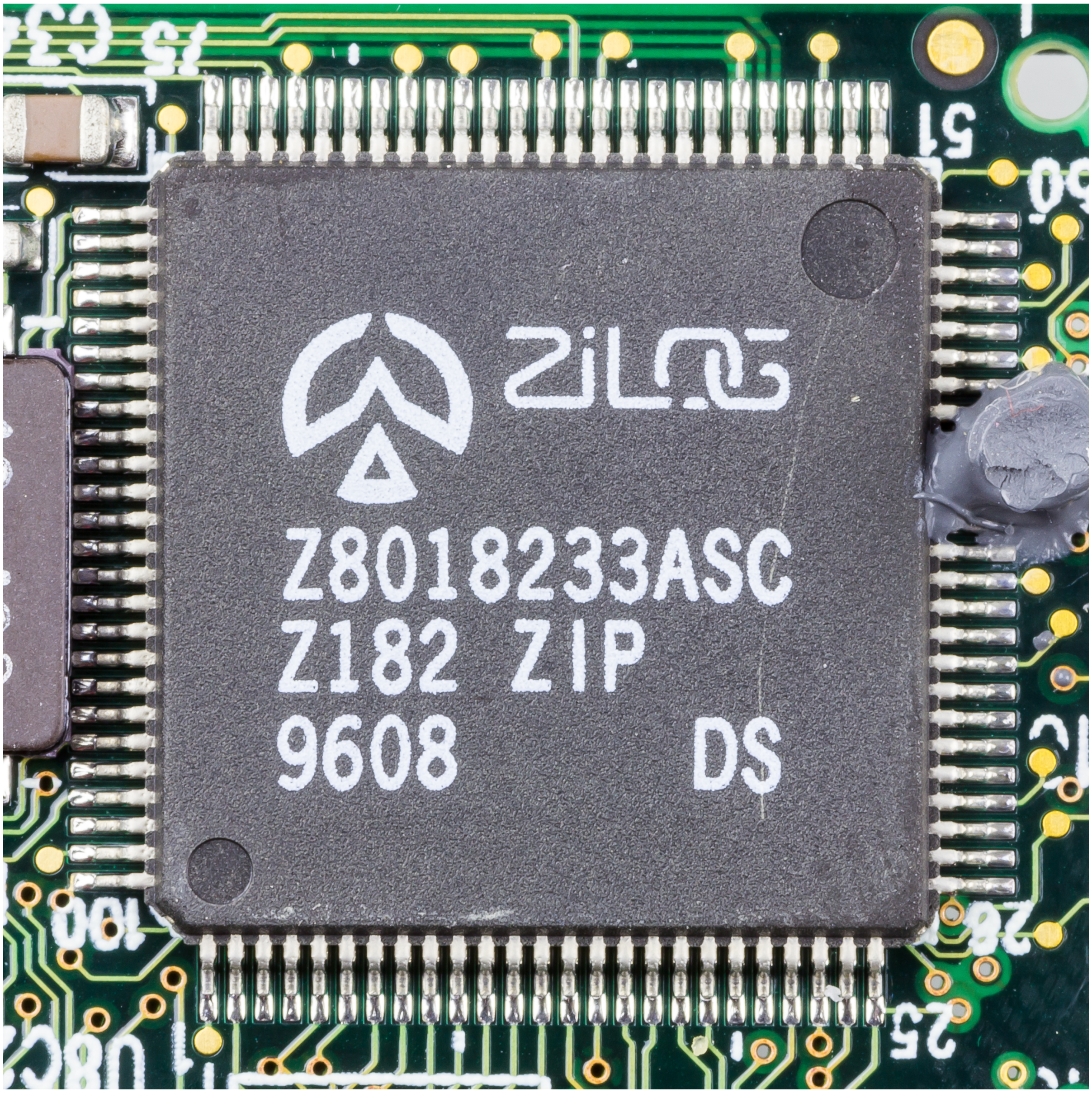
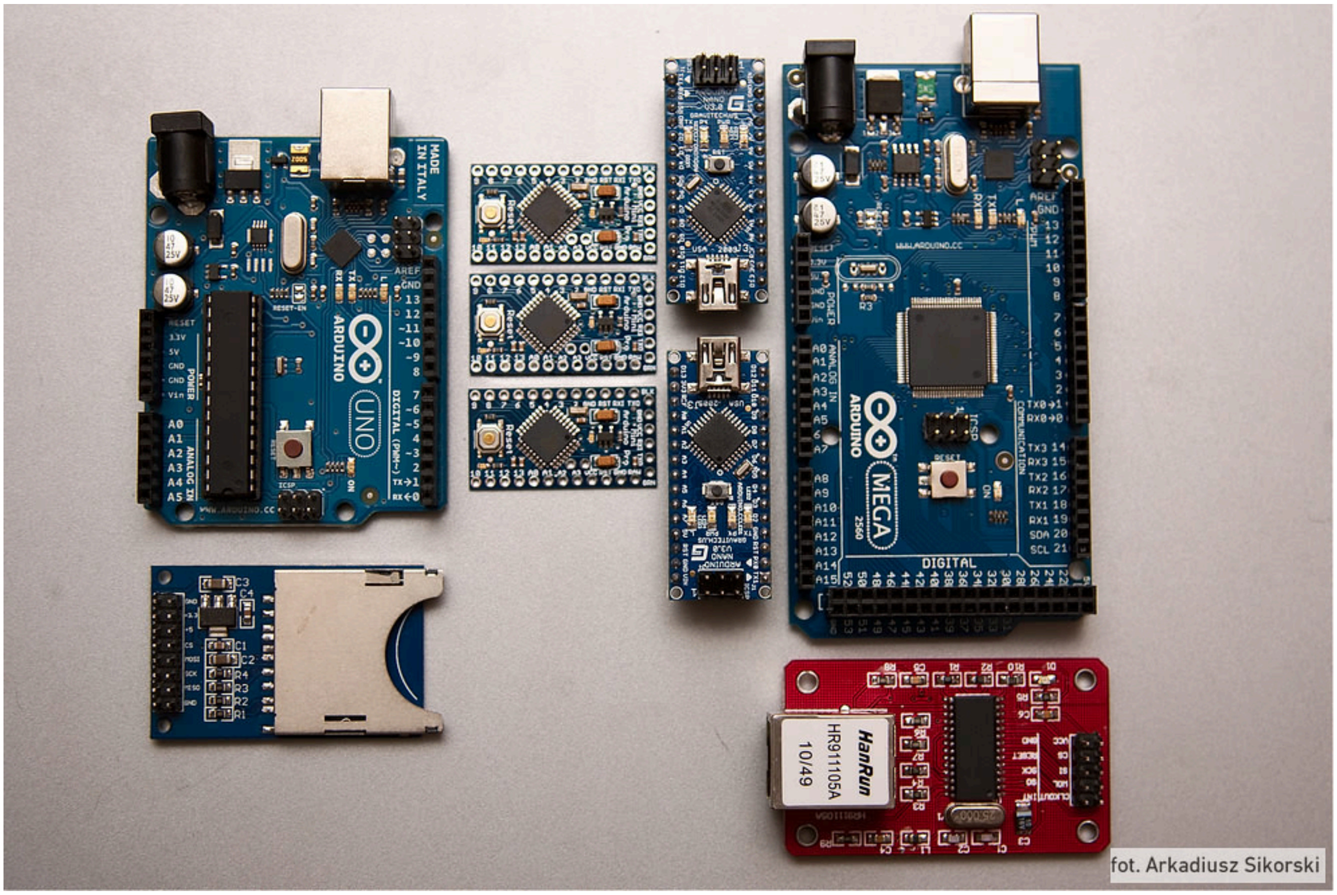


Figure 1: Microprocessor ("IBM PCMCIA Data-Fax Modem V.34 FRU 42H4326 – ZiLOG microprocessor Z80182-9359" by Raimond Spekking is licensed under CC BY-SA 4.0.)

Microcontroller: A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system. It includes a processor, memory (RAM, ROM, EPROM), and input/output (I/O) peripherals on a single chip. Microcontrollers are used in automatically controlled devices such as automobile engine control systems, medical devices, remote controls, office machines, appliances, power tools, toys, and other embedded systems.



fot. Arkadiusz Sikorski

Figure 2: Microcontroller (Arduino microcontroller / Arduino Uno microcontroller ATmega328 / Arduino Mega 2560 / Arduino Nano ATmega328 / Arduino Pro Mini ver.3.3V ATmega328 / Photo by Arkadiusz Sikorski http://www.arq.pl/" by Arkadiusz Sikorski is licensed under CC BY 2.0.)

1.2 Historical Development

The development of microprocessors and microcontrollers has been a significant milestone in the evolution of computing technology.

Microprocessors:

- **1971:** Intel introduced the first microprocessor, the Intel 4004, a 4-bit processor capable of performing basic arithmetic and logic operations.
- **1972-1978:** The second generation saw the development of 8-bit microprocessors like the Intel 8008 and 8080, which were used in early personal computers.
- **1978-1980:** The third generation introduced 16-bit processors such as the Intel 8086 and Motorola 68000, which offered improved performance and capabilities.
- **1981-1995:** The fourth generation brought 32-bit processors like the Intel 80386, which became

popular for both control applications and number-crunching operations.

- **1995-Present:** The fifth generation includes 64-bit processors like the Intel Pentium series, which are used in modern computers and offer high performance and speed.

Microcontrollers:

- **1971:** The first microcontroller, the TMS 1000, was developed by Texas Instruments, combining a processor, memory, and I/O on a single chip.
- **1974:** Intel introduced the 8048, which became widely used in embedded systems such as PC keyboards.
- **1993:** The introduction of EEPROM memory allowed for electrically erasable and programmable microcontrollers, facilitating rapid prototyping and in-system programming.
- **Present:** Modern microcontrollers are highly integrated and used in a wide range of applications, from consumer electronics to industrial automation.

1.3 Comparison Between Microprocessors and Microcontrollers

Feature	**Microprocessor**	**Microcontroller**
Components	CPU only	CPU, memory, I/O ports, peripherals
Purpose	General-purpose computing	Specific control tasks in embedded systems
Memory	External	On-chip
I/O Ports	External	On-chip
Architecture	Von Neumann	Harvard
Cost	Higher	Lower
Power Consumption	Higher	Lower
Applications	Personal computers, servers, complex systems	Home appliances, automotive systems, toys
Processing Power	Higher	Lower
Development Complexity	More complex	Simpler

1.4 Basic Functions and Applications

Basic Functions of Microprocessors:

- **Fetch, Decode, Execute:** The microprocessor fetches instructions from memory, decodes them to understand the required operations, and executes them.
- **Arithmetic and Logic Operations:** Performs basic arithmetic operations (addition, subtraction) and logic operations (AND, OR, NOT).
- **Data Transfer:** Moves data between different parts of the system, such as from memory to registers.
- **Control Operations:** Manages the control signals to coordinate the activities of the system components.

Basic Functions of Microcontrollers:

- **Embedded Control:** Designed to perform specific control tasks within an embedded system.
- **I/O Operations:** Interfaces with external devices through built-in I/O ports.
- **Timers and Counters:** Includes peripherals for timing and counting operations.
- **Analog to Digital Conversion:** Converts analog signals to digital form for processing.

Applications:

- **Microprocessors:** Used in personal computers, servers, high-performance computing systems, and complex industrial controllers.
- **Microcontrollers:** Found in household appliances (washing machines, microwaves), automotive systems (engine control units), medical devices (ECG machines), and consumer electronics (remote controls, digital cameras).

Deepen your understanding: Watch the accompanying lecture video to get deeper into the concepts covered in the reading.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=5#oembed-1

MICROPROCESSOR ARCHITECTURE

2.1 Components of a Microprocessor

A microprocessor is composed of several key components that work together to process data and execute instructions. The three main components are:

1. Arithmetic Logic Unit (ALU):

The ALU is the mathematical brain of the microprocessors. It performs arithmetic operations (addition, subtraction, multiplication, division) and logical operations (AND, OR, NOT, XOR). The first commercial ALU was the Intel 74181, released in 1970 as part of the 7400 series TTL integrated circuits.

2. Control Unit:

The control unit coordinates the activities of other components by interpreting instructions and generating control signals. It manages the fetch-decode-execute cycle and directs the flow of data between the ALU, registers, and memory.

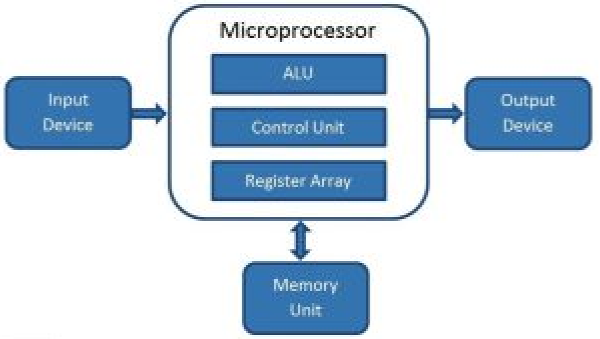


Figure 1: Microprocessor Architecture

3. Registers:

Registers are small, high-speed storage locations within the processor. They store temporary data, instructions, and memory addresses. Common registers include:

- Program Counter (PC): Holds the address of the next instruction to be executed.
- Instruction Register (IR): Stores the current instruction being decoded and executed.
- Memory Address Register (MAR): Holds the memory address for read/write operations.
- Memory Data Register (MDR): Temporarily stores data being transferred to/from memory.
- General Purpose Registers: Used for various data storage needs during processing.

2.2 Bus Systems

Buses are communication systems that transfer data between components. There are three main types of buses in a microprocessor system.

1. Address Bus:

- Unidirectional (microprocessor to memory/I/O devices)
- Carries memory or I/O device addresses
- Width determines addressable memory (e.g., 16-bit bus can address 65,536 locations)

2. Data Bus:

- Bidirectional
- Transfers data between microprocessor and memory/I/O devices
- Width affects the amount of data transferred per cycle (e.g., 8-bit bus can transfer values 0-255)

3. Control Bus:

- Bidirectional
- Carries control signals (read, write, interrupt, etc.)
- Coordinates operations between microprocessor and other components

2.3 Memory Organization

Memory in a computer system is organized in a hierarchy to balance speed, cost, and capacity:

1. **Registers:** Fastest, smallest capacity, within the CPU
2. **Cache Memory:** Very fast, small capacity, close to CPU
3. **Main Memory (RAM):** Fast, larger capacity, directly accessible by CPU
4. **Secondary Memory:** Slow, largest capacity, not directly accessible by CPU

This hierarchy is designed to optimize access times and storage capacity. Frequently used data is kept in faster, more expensive memory closer to the CPU, while less frequently used data is stored in slower, cheaper memory farther from the CPU.

2.4 Instruction Cycle and Fetch-Execute Process

The instruction cycle, also known as the fetch-decode-execute cycle, is the basic operational process of a CPU. It consists of several steps:

1. **Fetch:**
   - CPU retrieves instruction from memory address in Program Counter (PC)
   - Instruction is loaded into the Instruction Register (IR)
   - PC is incremented to point to the next instruction
2. **Decode:**
   - CPU interprets the instruction in the IR
   - Determines the operation to be performed and identifies operands
3. **Execute:**
   - CPU performs the operation specified by the instruction
   - May involve ALU operations, data transfers, or control flow changes
4. **Additional Steps (depending on architecture):**
   - Fetch operands: Retrieve data needed for instruction execution
   - Store results: Write results back to memory or registers
   - Interrupt handling: Check for and process any interrupts

This cycle repeats continuously as the CPU processes instructions. The efficiency of this cycle significantly impacts the overall performance of the microprocessor.

For further explanation: A video lecture following this reading material provides additional insights and clarifications

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=23#oembed-1

References:

[1] https://www.geeksforgeeks.org/introduction-of-alu-and-data-path/

[2] https://www.geeksforgeeks.org/bus-organization-of-8085-microprocessor/

[3] https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/

[4] https://www.geeksforgeeks.org/different-instruction-cycles/

INTRODUCTION TO MICROCONTROLLER SYSTEMS

3.1 Microcontroller Components

Microcontrollers are compact integrated circuits designed for embedded applications. They typically consist of the following key components:

1. Central Processing Unit (CPU):

The CPU is the core of the microcontroller, responsible for executing instructions and performing calculations. It's optimized for embedded systems, often prioritizing low power consumption over high performance.

2. Memory:

Microcontrollers incorporate different types of memory:

- Flash memory: Stores the program code
- RAM: Holds variables and data during operation
- EEPROM: Non-volatile memory for storing configuration data

3. Input/Output (I/O) Ports:

I/O ports allow the microcontroller to interact with external devices. These can include:

- Digital I/O pins: For interfacing with sensors and actuators
- Analog I/O: Often including Analog-to-Digital Converters (ADC) for reading analog signals.

4. Timers/Counters:

These components measure time intervals and count events, crucial for tasks like generating PWM signals or measuring input frequencies.

5. Interrupt Controller:

Manages interrupt requests from various sources, allowing the microcontroller to respond to external events quickly

6. Communication Interfaces:

Many microcontrollers include built-in interfaces such as UART, SPI, I2C, and sometimes USB or Ethernet for communication with other devices.

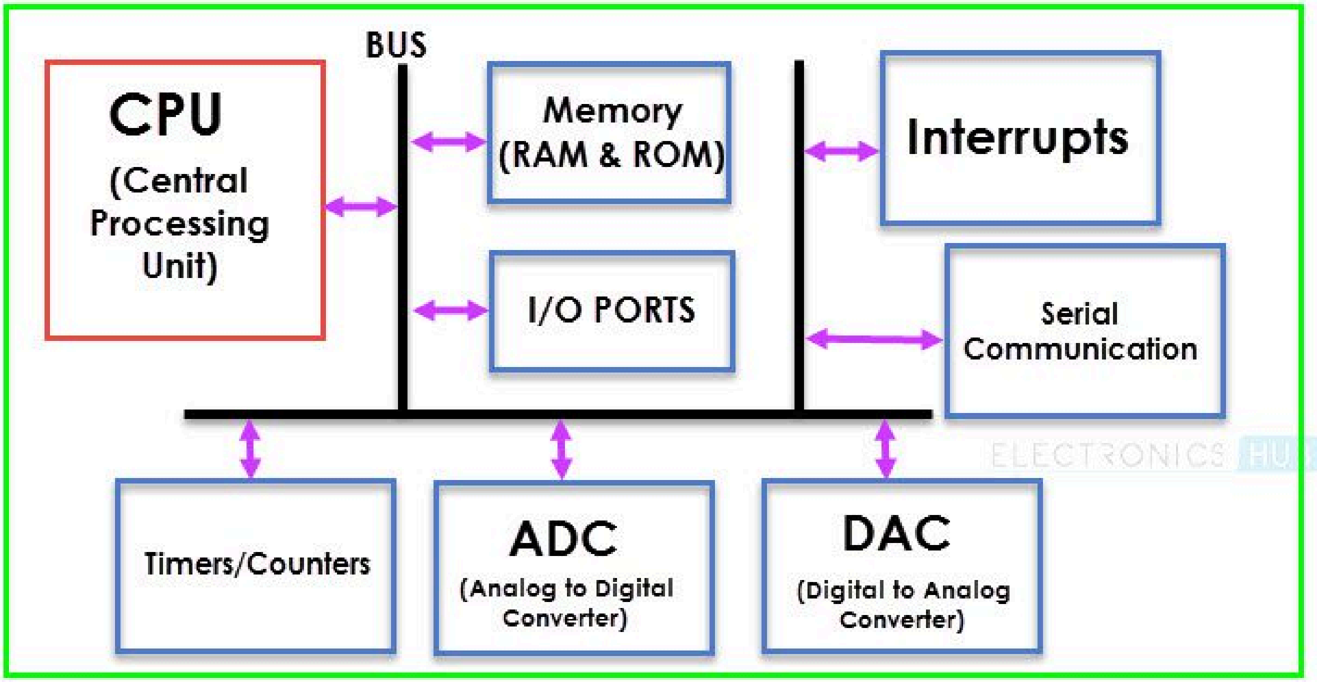


Figure 1: Microcontroller Component

3.2 Embedded System Concepts

Embedded systems are specialized computing systems designed to perform dedicated functions within larger systems. Key concepts include:

- **Real-time operation:** Many embedded systems must respond to events within strict time constraints.
- **Resource constraints:** Embedded systems often have limited processing power, memory, and energy resources.
- **Reliability:** These systems must operate reliably, often in harsh environments or safety-critical applications.
- **Integration:** Embedded systems are tightly integrated with their surrounding hardware and environment.

3.3 Types of Microcontrollers

Microcontrollers are categorized based on their data bus width:**1. 8-bit Microcontrollers:**

- Can process 8 bits of data at a time
- Suitable for simple applications with low processing requirements

- Examples: Atmel AVR, PIC16 series
- Advantages: Low cost, low power consumption
- Limitations: Limited processing power, smaller address space (typically up to 64 KB).

2. 16-bit Microcontrollers:

- Can process 16 bits of data at a time
- Offer a balance between processing power and energy efficiency
- Examples: PIC24, MSP430
- Advantages: Higher processing power than 8-bit, still relatively low power consumption
- Limitations: More expensive than 8-bit, less powerful than 32-bit.

3. 32-bit Microcontrollers:

- Can process 32 bits of data at a time
- Suitable for complex applications requiring significant processing power
- Examples: ARM Cortex-M series, PIC32
- Advantages: High processing power, large address space (up to 4 GB)
- Limitations: Higher cost and power consumption compared to 8-bit and 16-bit options.

3.4 Common Microcontroller Families

1. PIC (Peripheral Interface Controller):

- Developed by Microchip Technology
- Wide range of 8-bit, 16-bit, and 32-bit options
- Known for reliability and robustness
- Rich set of peripherals and modules
- Challenges: Can be complex to program for beginners.

2. AVR:

- Developed by Atmel (now part of Microchip)
- Popular among hobbyists and in educational settings
- Easy to program and debug
- Low power consumption
- Limitations: Limited memory and peripheral capacity in some models.

3. ARM:

- Developed by ARM Holdings
- Dominates the 32-bit microcontroller market
- High performance and energy efficiency
- Widely used in smartphones, tablets, and other complex embedded systems
- Extensive ecosystem and development tools
- Challenges: Can be more complex for simple applications.

Each microcontroller family has its strengths and is suited for different types of applications. The choice between them often depends on factors such as processing requirements, power constraints, development ecosystem, and familiarity with the platform.

In the next chapter, we'll explore how to start programming these microcontrollers and implement basic input/output operations.

Deepen your understanding: Watch the accompanying lecture video to delve deeper into the concepts covered in the reading.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=40#oembed-1

GETTING STARTED WITH MICROCONTROLLER PROGRAMMING

4.1 Programming Languages for Microcontrollers

Microcontrollers can be programmed using various languages, each with its own advantages and limitations. The most commonly used languages are Assembly and C.

Assembly Language

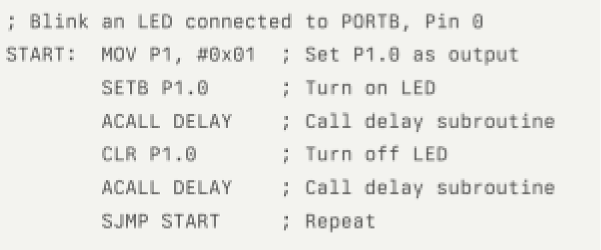
Assembly language is a low-level programming language that provides direct control over the hardware. It is specific to a particular microcontroller architecture.**Pros:**

- High efficiency and speed
- Direct hardware manipulation
- Small code size

Cons:

- Complex and difficult to learn
- Error-prone and tedious
- Non-portable across different microcontroller families

Example:



```
; Blink an LED connected to PORTB, Pin 0
START:  MOV P1, #0x01  ; Set P1.0 as output
        SETB P1.0      ; Turn on LED
        ACALL DELAY    ; Call delay subroutine
        CLR P1.0       ; Turn off LED
        ACALL DELAY    ; Call delay subroutine
        SJMP START     ; Repeat
```

C Language

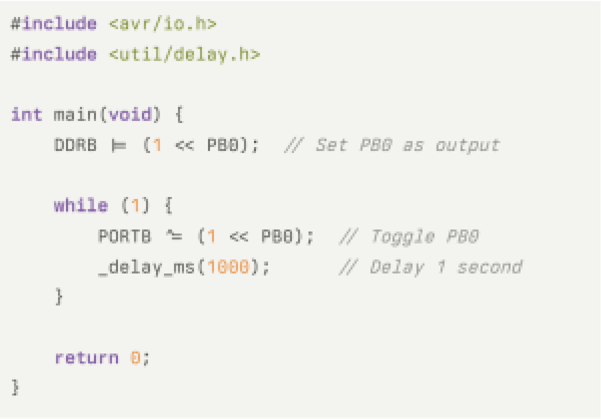
C is a high-level programming language widely used for microcontroller programming due to its balance between control and ease of use.**Pros:**

- Easier to learn and write than Assembly
- Portable across different microcontrollers
- Rich set of libraries and functions

Cons:

- Less efficient than Assembly
- Larger code size
- Requires a compiler

Example:



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

int main(void) {
    DDRB |= (1 << PB0);  // Set PB0 as output

    while (1) {
        PORTB ^= (1 << PB0);  // Toggle PB0
        _delay_ms(1000);      // Delay 1 second
    }

    return 0;
}
```

4.2 Development Environments and Tools

Effective microcontroller programming requires a variety of tools to write, compile, debug, and upload code. Here are some of the most popular development environments and tools:

Integrated Development Environments (IDEs)

IDEs provide a unified interface for writing, compiling, and debugging code. Popular IDEs include:

- **Arduino IDE:** User-friendly, ideal for beginners

- **MPLAB X IDE:** Used for PIC microcontrollers
- **Atmel Studio:** Used for AVR microcontrollers
- **STM32CubeIDE:** Used for STM32 microcontrollers

Compilers and Assemblers

- **Compilers:** Convert high-level language code (e.g., C) into machine code. Examples include GCC for AVR and XC8 for PIC.
- **Assemblers:** Convert assembly language code into machine code.

Simulators and Debuggers

- **Simulators:** Allow testing code on a PC without hardware.
- **Debuggers:** Help identify and fix errors by providing features like breakpoints and step-by-step execution.

Programmers

- **Hardware Programmers:** Transfer code from a PC to the microcontroller. Examples include AVRISP for AVR and PICkit for PIC.
- **Bootloaders:** Software that allows programming via serial or USB without additional hardware.

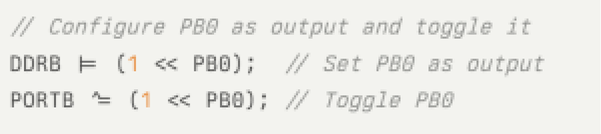
4.3 Basic Input/Output Operations

Microcontrollers interact with the external world through their I/O ports. Understanding how to perform basic I/O operations is crucial for any microcontroller project.

Digital I/O

Digital I/O pins can be configured as either inputs or outputs.

Example:

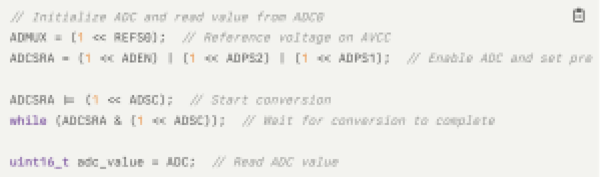


```
// Configure PB0 as output and toggle it
DDRB |= (1 << PB0);  // Set PB0 as output
PORTB ^= (1 << PB0); // Toggle PB0
```

Analog I/O

Many microcontrollers have built-in Analog-to-Digital Converters (ADC) for reading analog signals.

Example:



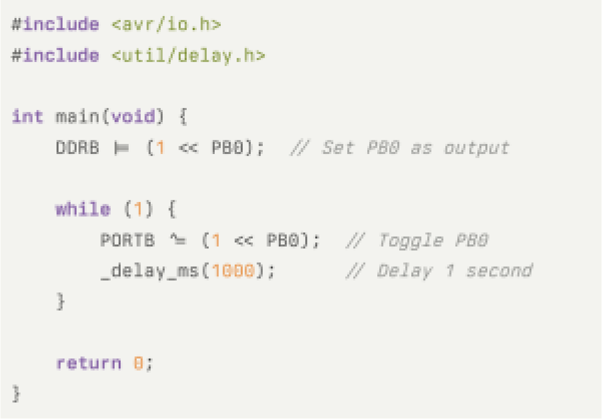
```
// Initialize ADC and read value from ADC0
ADMUX = (1 << REFS0);  // Reference voltage on AVCC
ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1);  // Enable ADC and set pre

ADCSRA |= (1 << ADSC);  // Start conversion
while (ADCSRA & (1 << ADSC));  // Wait for conversion to complete

uint16_t adc_value = ADC;  // Read ADC value
```

4.4 Simple Programming Examples

Example 1: Blinking an LED

This is the "Hello World" of microcontroller programming. It involves toggling an LED on and off.

Code:

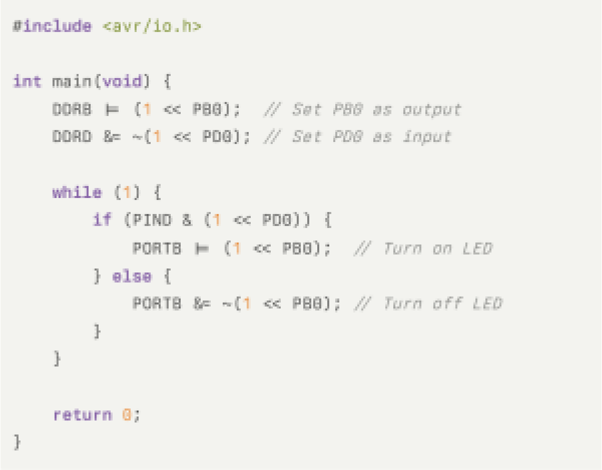


```
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
    DDRB |= (1 << PB0);  // Set PB0 as output

    while (1) {
        PORTB ^= (1 << PB0);  // Toggle PB0
        _delay_ms(1000);      // Delay 1 second
    }

    return 0;
}
```

Example 2: Reading a Button Press

This example reads the state of a button and turns on an LED when the button is pressed.

Code:



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

int main(void) {
    DDRB |= (1 << PB0);  // Set PB0 as output
    DDRD &= ~(1 << PD0); // Set PD0 as input

    while (1) {
        if (PIND & (1 << PD0)) {
            PORTB |= (1 << PB0);  // Turn on LED
        } else {
            PORTB &= ~(1 << PB0); // Turn off LED
        }
    }

    return 0;
}
```

End-of-Chapter Review Questions

1. What are the advantages and disadvantages of using Assembly language for microcontroller programming?
2. Describe the role of an Integrated Development Environment (IDE) in microcontroller programming.
3. Write a simple C program to toggle an LED connected to a microcontroller pin.
4. Explain the difference between digital and analog I/O operations in microcontrollers.
5. What tools are essential for debugging microcontroller code?

PART II
INTRODUCTION TO ASSEMBLY LANGUAGE

In the 'Introduction to Assembly Language' module, students will learn:

- **Definition and Key Features:** Understand what assembly language is and its suitability for time-critical and hardware-specific tasks.
- **Advantages and Disadvantages:** Explore the benefits of efficiency and control provided by assembly language, as well as its complexities and lack of portability.
- **Basic Concepts:** Familiarize with mnemonics, labels, operands, directives, and comments, and learn how to use these elements to write assembly programs.
- **The Assembly Process:** Learn the steps involved in creating and running an assembly language program, including writing the source code, assembling, linking, loading, and executing the program.
- **Practical Skills:** Develop the ability to write efficient code, directly control hardware, optimize performance, and implement specialized applications such as interrupt service routines and device drivers.

By the end of this module, students will have a solid foundation in assembly language, enabling them to program microcontrollers effectively and understand the inner workings of computer hardware at a low level.

FUNDAMENTALS OF ASSEMBLY LANGUAGE

Introduction

In this course, we will delve into the world of assembly language, a low-level programming language that provides direct control over a computer's hardware. Assembly language is essential for understanding how software interacts with hardware, and it is particularly useful for tasks that require precise timing, direct hardware manipulation, or optimized performance.

1.1 What is Assembly Language?

Assembly language is a low-level programming language that uses symbolic code to represent machine language instructions. Unlike high-level languages such as C, Java, or Python, which abstract many hardware details, assembly language provides a closer view of the computer's architecture and operations.

Key Features of Assembly Language:

- **Processor-specific:** Each assembly language is tailored to a particular processor family, meaning the instructions are specific to that architecture.
- **Direct hardware manipulation:** Assembly language allows programmers to interact directly with hardware components such as registers, memory locations, and I/O devices.
- **Efficiency:** Programs written in assembly language typically require less memory and execution time compared to those written in high-level languages.
- **Suitability for specific tasks:** Assembly language is ideal for time-critical and hardware-specific tasks, such as writing interrupt service routines and device drivers.

1.2 Advantages and Disadvantages

Advantages:

1. **Efficient use of memory and processing power:** Assembly language allows for optimal use of the computer's resources.
2. **Direct hardware control:** Programmers can directly manipulate hardware components, providing

greater control over the system.

3. **Performance optimization:** Assembly language enables the creation of highly optimized code tailored to specific hardware configurations.
4. **Ideal for specific applications:** It is well-suited for writing interrupt service routines, device drivers, and memory-resident programs.

Disadvantages:

1. **Complexity:** Assembly language is more difficult to learn and use compared to high-level languages.
2. **Limited portability:** Programs written in assembly language are typically specific to a particular processor architecture and are not easily transferable to other systems.
3. **Time-consuming:** Writing and maintaining assembly code is often more time-intensive than using high-level languages.
4. **Steeper learning curve:** It requires a deeper understanding of computer architecture and hardware operations.

1.3 Basic Assembly Language Concepts

1. Mnemonics:

Mnemonics are symbolic representations of machine instructions. They make the code more readable and easier to understand. Examples include:

- **MOV:** Move data from one location to another
- **ADD:** Add two values
- **JMP:** Jump to a specified address

2. Labels:

Labels are symbolic names used to mark specific memory addresses or program locations. They help organize the code and making it more readable. For example:

```
START:
        MOV  AX, 1
        ADD  AX, 2
        JMP   START
```

3. Operands:

Operands are the data or memory locations that instructions operate on. They can be immediate values, registers, or memory addresses. For example:

```
MOV AX, 5 ; Immediate value
   MOV BX, AX ; Register
   MOV [1234H], AX ; Memory address
```

4. Directives:

Directives are special commands for the assembler that do not translate directly to machine code but affect the assembly process. Examples include:

- **ORG:** Set the origin (starting address) for the code
- **END:** Mark the end of the program

5. Comments:

Comments are explanatory text ignored by the assembler. They are used to improve the readability of code and provide explanations. For example:

```
   MOV AX, 5 ; Load the value 5 into register
   AX ADD AX, 3 ; Add 3 to the value in AX
```

1.4 The Assembly Process

The process of creating and running an assembly language program involves several steps:

1. **Writing the source code:**
   The programmer writes the assembly language instructions using a text editor. The code is saved with a specific file extension, such as `.asm`.
2. **Assembling the code:**
   An assembler converts the assembly language code into machine language, generating an object file with an extension like `.obj`.
3. **Linking:**
   If the program consists of multiple source files or uses external libraries, a linker combines these into a single executable file. This step resolves references between different parts of the program.
4. **Loading:**
   The operating system loads the executable file into the computer's memory, preparing it for execution.
5. **Execution:**
   The CPU reads and executes the machine-language instructions, performing the tasks specified by the program.

For further explanation: A video lecture following this reading material provides additional insights and clarifications

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=58#oembed-1

BOOLEAN ALGEBRA IN ASSEMBLY PROGRAMMING

2.1 Introduction to Boolean Algebra

Boolean algebra, developed by George Boole in the mid-19th century, is a fundamental mathematical system that forms the basis of digital logic and computer science. It deals with the manipulation of logical values, typically represented as:

- True (1)
- False (0)

In the context of computer systems and assembly programming, Boolean algebra provides the foundation for:

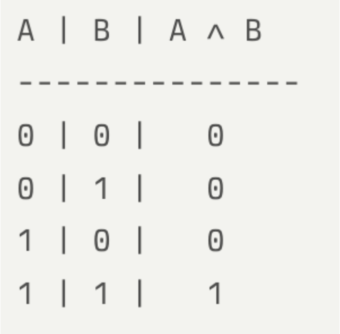
1. Logical operations in digital circuits
2. Decision-making in software
3. Bit-level manipulations in data processing

Understanding Boolean algebra is crucial for assembly language programmers as it directly relates to how computers process and manipulate data at the lowest level.

2.2 Basic Boolean Operations

The three fundamental Boolean operations are:

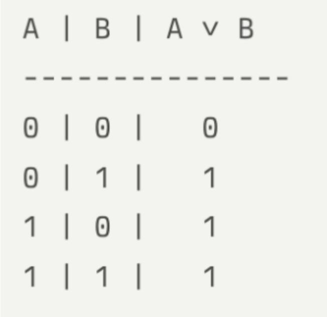
1. **AND (∧)**
    - Symbol: ∧ or •
    - Truth table:



```
A | B | A ∧ B
---------------
0 | 0 |   0
0 | 1 |   0
1 | 0 |   0
1 | 1 |   1
```

    - The result is true only if both inputs are true.

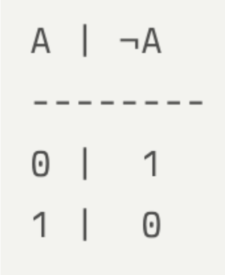
2. **OR (∨)**
    - Symbol: ∨ or +
    - Truth table:



```
A | B | A ∨ B
---------------
0 | 0 |   0
0 | 1 |   1
1 | 0 |   1
1 | 1 |   1
```

    - The result is true if at least one input is true.

3. **NOT (¬)**
    - Symbol: ¬ or ‘
    - Truth table:



```
A | ¬A
--------
0 |  1
1 |  0
```

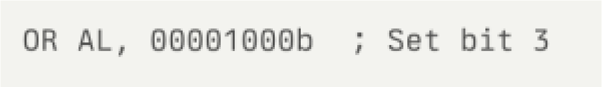
    - Inverts the input.

These basic operations can be combined to create more complex logical expressions and functions.

2.3 Boolean Algebra in Assembly Language

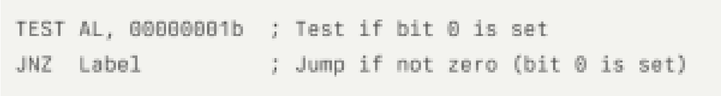
In assembly language, Boolean operations are fundamental for various programming tasks:

1. **Bit Manipulation**
    - Setting, clearing, or toggling individual bits in registers or memory locations.
    - Example (setting bit 3 in AL register):



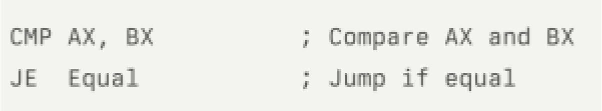
```
OR AL, 00001000b  ; Set bit 3
```

2. **Conditional Branching**
    - Making decisions based on the result of logical operations.
    - Example:



```
TEST AL, 00000001b  ; Test if bit 0 is set
JNZ  Label          ; Jump if not zero (bit 0 is set)
```

3. **Logical Comparisons**
    - Comparing values and setting flags based on the result.
    - Example:



```
CMP AX, BX          ; Compare AX and BX
JE  Equal           ; Jump if equal
```

Common assembly instructions for Boolean operations include:

- **AND:** Performs bitwise AND operation
- **OR:** Performs bitwise OR operation
- **XOR:** Performs bitwise exclusive OR operation
- **NOT:** Inverts all bits
- **TEST:** Performs AND operation without storing the result, affects flags
- **CMP:** Compares two operands by subtraction, affects flags

2.4 Applications in Microcontroller Programming

Boolean algebra is extensively used in microcontroller programming for various purposes:

1. **Setting/Clearing Specific Bits in Control Registers**

   - Example: Configuring an I/O pin as output

     text

     ```
     MOV AL, [PORTA_DIR]     ; Load current direction register
     OR AL, 00000001b        ; Set bit 0 (configure as output)
     MOV [PORTA_DIR], AL     ; Store back to direction register
     ```

2. **Creating Bit Masks for I/O Operations**

   - Example: Reading specific pins while ignoring others

     text

     ```
     MOV AL, [PORTA_IN]      ; Read input port
     AND AL, 00001111b       ; Mask out upper 4 bits
     ```

3. **Implementing Conditional Logic**

   - Example: Checking multiple conditions

     text

     ```
     MOV AL, [STATUS]     ; Load status byte
     AND AL, 00000011b    ; Check lower two bits
     CMP AL, 00000011b    ; Are both bits set?
     JE BothSet           ; Jump if both set
     ```

4. **Optimizing Code**

   - Using Boolean operations can often lead to more efficient code than using arithmetic operations or conditional statements.

5. **Implementing State Machines**

   - Boolean algebra is useful for implementing finite state machines, which are common in embedded systems.

A video lecture following this reading material provides additional insights.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=69#oembed-1

MPLAB IDE AND ASSEMBLY PROGRAMMING

3.1 Introduction to MPLAB X IDE

MPLAB X IDE is a powerful Integrated Development Environment designed specifically for developing applications for Microchip microcontrollers and digital signal controllers. It provides a comprehensive set of tools to streamline the development process from writing code to debugging and testing.Key features of MPLAB X IDE include:

1. **Code Editor with Syntax Highlighting:**
    - Supports multiple languages including Assembly and C
    - Provides intelligent code completion and error highlighting
    - Offers customizable color schemes for improved readability
2. **Debugger:**
    - Allows step-by-step execution of code
    - Provides real-time variable watching and memory inspection
    - Supports breakpoints and conditional breakpoints
3. **Project Manager:**
    - Organizes source files, header files, and libraries
    - Manages project configurations and build settings
    - Supports version control integration
4. **Simulator:**
    - Allows testing of code without physical hardware
    - Simulates various microcontroller peripherals
    - Provides cycle-accurate simulation for precise timing analysis

3.2 Setting Up an Assembly Project in MPLAB X

To create a new assembly project in MPLAB X:

1. **Creating a new project:**
    - Open MPLAB X IDE
    - Click File > New Project
    - Select "Microchip Embedded" > "Standalone Project"
    - Choose your target device (e.g., PIC16F877A)
    - Select "MPLAB X ASM" as the tool chain
2. **Configuring the project for assembly language:**
    - In the project properties, ensure the language toolchain is set to MPASM
    - Set the optimization level (typically "0" for debugging)
3. **Adding source files:**
    - Right-click on "Source Files" in the project tree
    - Select "New" > "ASM File"
    - Name your file (e.g., "main.asm")
4. **Setting up the build tools:**
    - Verify that MPASM is selected as the assembler
    - Configure any necessary include paths or library directories

3.3 Writing and Testing Assembly Code in MPLAB X

1. **Using the code editor:**
    - Write your assembly code in the created .asm file
    - Use the syntax highlighting to identify different elements (labels, mnemonics, operands)
    - Utilize code folding for better organization of large files
2. **Assembling the code:**

- Click the "Build Main Project" button or press F11
- Review the Output window for any errors or warnings

3. **Using the simulator for testing:**

- Set the simulator as the debug tool in project properties
- Set breakpoints in your code by clicking in the left margin
- Start debugging (F5) and use step commands (F7 for step into, F8 for step over)

4. **Debugging techniques:**

- Use the Watch window to monitor variable values
- Utilize the Memory window to inspect RAM and program memory
- Use the Disassembly window to view machine code alongside your assembly code

3.4 Implementing Delays and LED Blinking

1. **Creating delay routines using loops:**

```
Delay:
   MOVLW D'255'    ; Load W with 255
Loop:
  DECFSZ W, 1     ; Decrement W, skip next if zero

  GOTO Loop       ; Repeat until W becomes 0
  RETURN
```

2. **Configuring I/O ports for LED control:**

```
; Configure PORTB pin 0 as output
BSF STATUS, RP0       ; Select Bank 1
BCF TRISB, 0          ; Set RB0 as output
BCF STATUS, RP0       ; Return to Bank 0
```

3. **Writing code for blinking LEDs:**

```
Start:
   BSF PORTB, 0        ; Turn on LED
   CALL Delay          ; Wait
   BCF PORTB, 0        ; Turn off LED
   CALL Delay          ; Wait
```

```
GOTO
    Start                   ; Repeat
```

Testing and debugging the program:

- Use the simulator to step through the code
- Monitor the PORTB register in the Watch window
- Adjust delay values if necessary

3.5 Best Practices for Assembly Programming in MPLAB X

1. **Proper code organization and commenting:**
   - Use meaningful labels for routines and variables
   - Comment each section of code explaining its purpose
   - Use consistent indentation for improved readability
2. **Efficient use of registers and memory:**
   - Utilize bank selection efficiently to minimize bank switching
   - Use bit-oriented instructions when possible for flag manipulation
   - Optimize loop counters and temporary variables
3. **Optimizing for speed and size:**
   - Use lookup tables for complex calculations when appropriate
   - Minimize subroutine calls in time-critical sections
   - Use conditional assembly directives for different optimization levels
4. **Version control integration:**
   - Use MPLAB X's built-in Git integration or external version control
   - Commit changes regularly with meaningful commit messages
   - Utilize branches for experimental features or bug fixes

By following these practices and utilizing the powerful features of MPLAB X IDE, you can efficiently develop, test, and optimize your assembly language programs for microcontrollers.

A video lecture following this reading material provides additional insights.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=84#oembed-1

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=84#oembed-2

LAB #1 : REVIEW INSTRUCTION SET AND ORDER PARTS

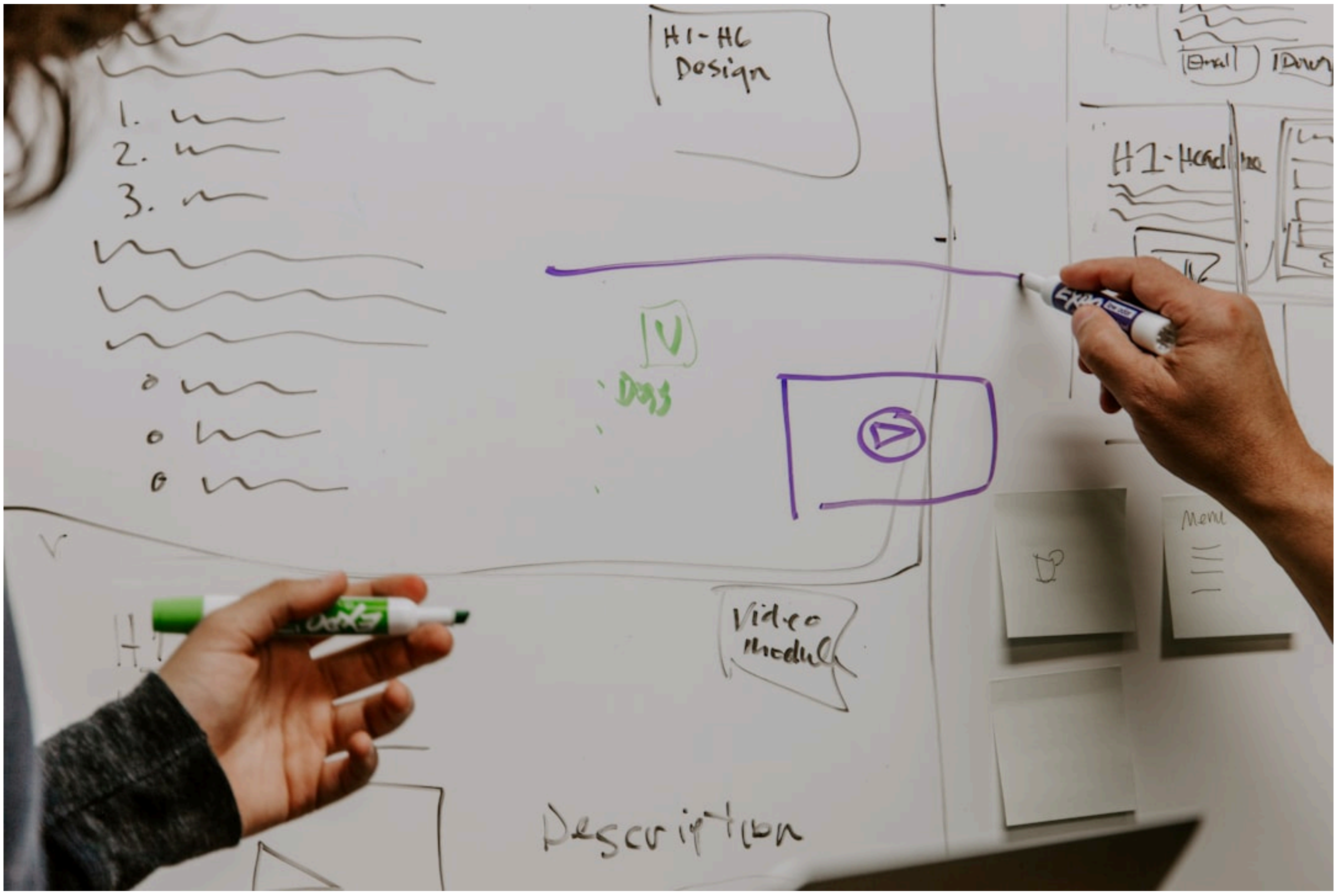
1. Download PIC16F18875 Data Sheets (www.microchip.comLinks to an external site.) and start reading it. Order MICROCHIP TECHNOLOGY DM164136 PIC Micro MCU Curiosity High Pin Count (HPC) Development Board PIC MCU 8-Bit

https://www.microchip.com/Developmenttools/ProductDetails/DM164136Links to an external site.

2. Download and install MPLAB IDE (Latest version) (Report – screenshot that the software is installed. If you have the development board, add a screenshot that it works)

3. Read about the MICROCHIP TECHNOLOGY DM164136 PIC Micro MCU Curiosity High Pin Count (HPC) Development Board PIC MCU 8-Bit and make notes.

LAB #2 STUDENT ROBOT BLOCK DIAGRAM/FLOWCHART EXERCISE



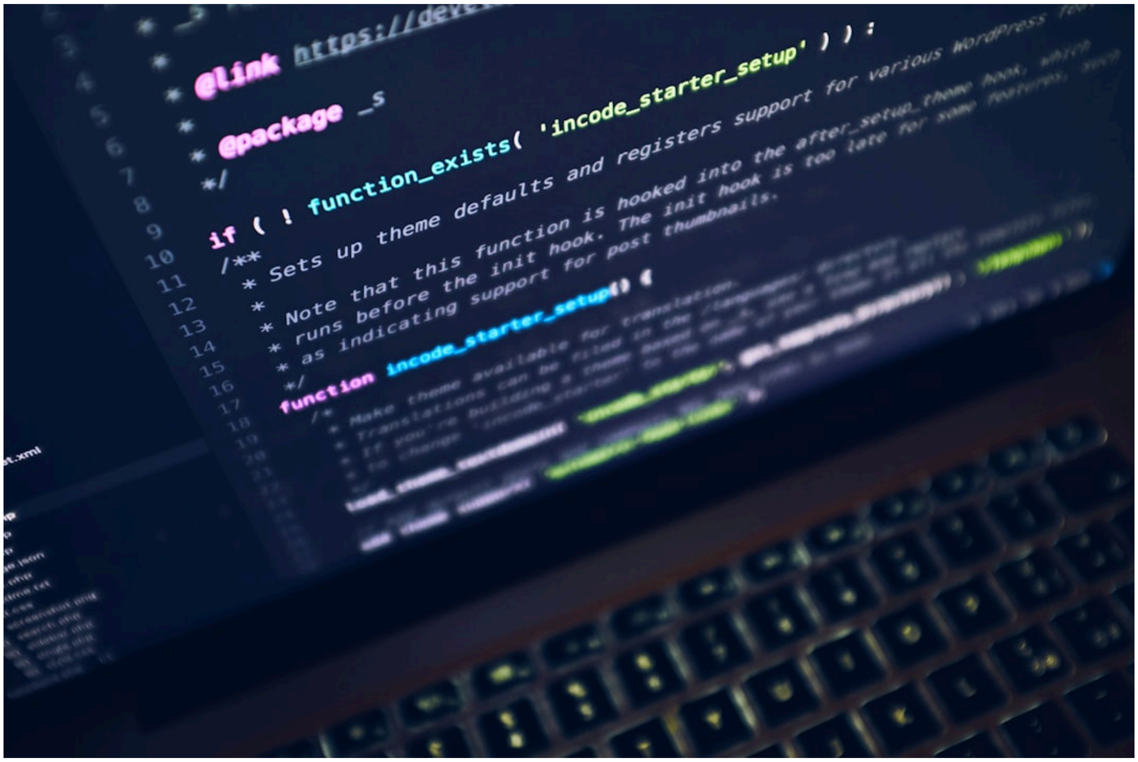
In-class Exercise:

- Students will create a block diagram of a program for students, starting from sitting in the chair to opening the classroom door.
- Once the block diagram is complete, other students will behave as student robots, and they will follow the instructions you made.
- Check if student Robot is able to understand your instructions.
- All the teams in the class should now have a chance to see the Me Robot execute their program. Remember that the Me Robot cannot do anything other than the instructions and knows nothing more than what you have commanded it—he/She/they may crash into walls, etc.

Answer the following question:

1. What is a processor? What components does it have?
2. What makes a microprocessor different from the processor of a large computer?
3. What makes an MCU different from a general-purpose microprocessor?

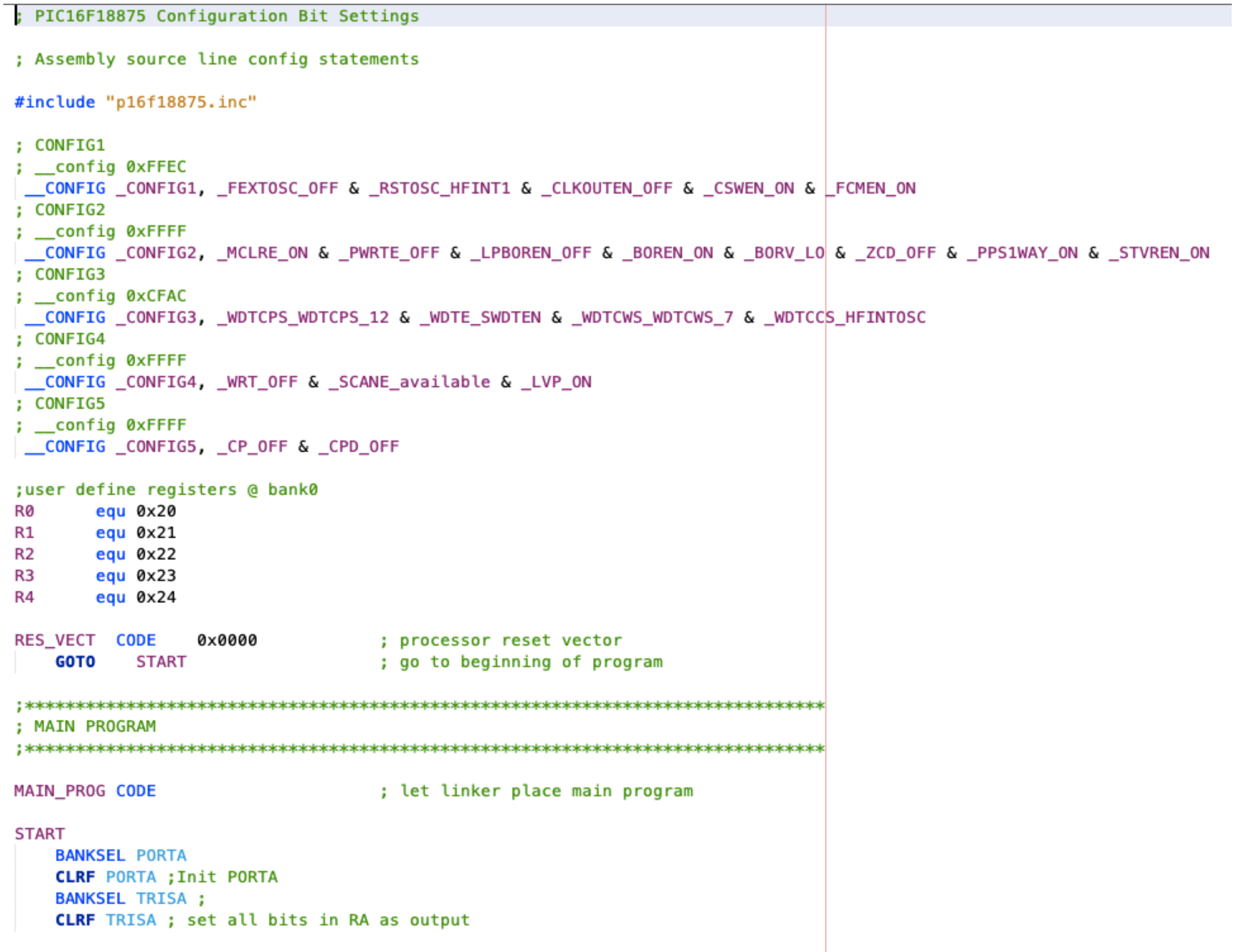
LAB #3 : TEST A PROGRAM ON MPLAB



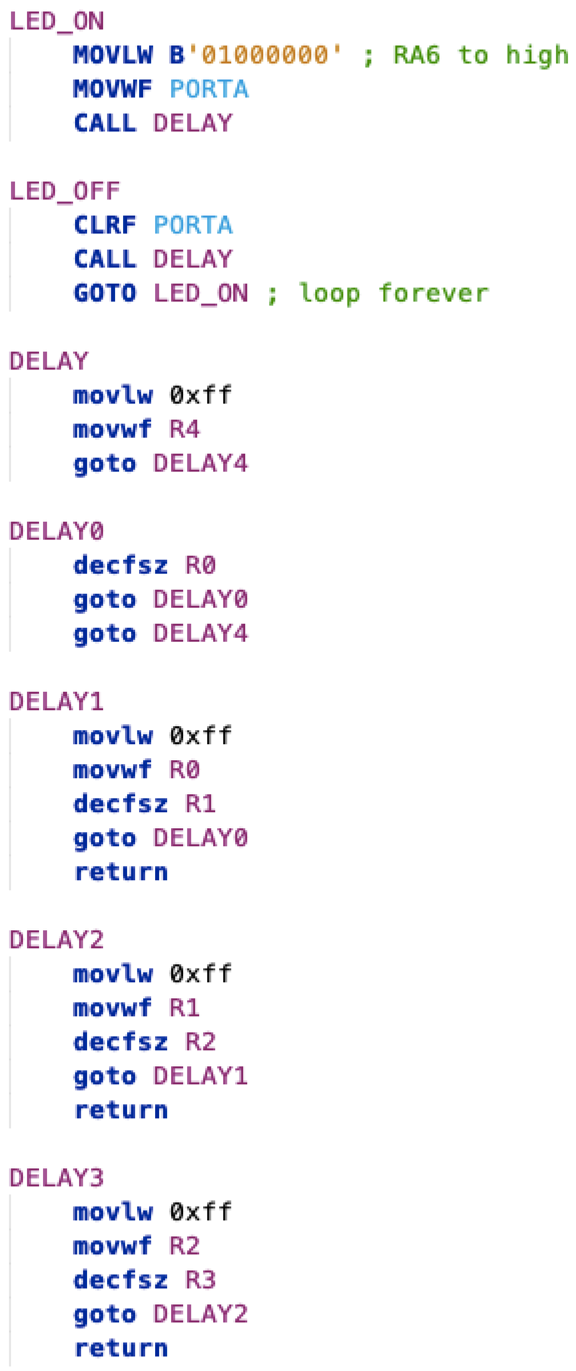
In this design project, the students are to copy the program attached and enter it into the MPLAB editor.

1. Build the program to get aHEX file.
2. Download the.HEX file to the development board.
3. Demonstrate that the program runs by looking at the output pin.
4. Produce a technical report as if you were in the industry using template.
5. A conclusion of your results and discussion of anything you found especially interesting or not expected from your work on this project.

CODE to test



```
; PIC16F18875 Configuration Bit Settings

; Assembly source line config statements

#include "p16f18875.inc"

; CONFIG1
; __config 0xFFEC
 __CONFIG _CONFIG1, _FEXTOSC_OFF & _RSTOSC_HFINT1 & _CLKOUTEN_OFF & _CSWEN_ON & _FCMEN_ON
; CONFIG2
; __config 0xFFFF
 __CONFIG _CONFIG2, _MCLRE_ON & _PWRTE_OFF & _LPBOREN_OFF & _BOREN_ON & _BORV_LO & _ZCD_OFF & _PPS1WAY_ON & _STVREN_ON
; CONFIG3
; __config 0xCFAC
 __CONFIG _CONFIG3, _WDTCPS_WDTCPS_12 & _WDTE_SWDTEN & _WDTCWS_WDTCWS_7 & _WDTCCS_HFINTOSC
; CONFIG4
; __config 0xFFFF
 __CONFIG _CONFIG4, _WRT_OFF & _SCANE_available & _LVP_ON
; CONFIG5
; __config 0xFFFF
 __CONFIG _CONFIG5, _CP_OFF & _CPD_OFF

;user define registers @ bank0
R0      equ 0x20
R1      equ 0x21
R2      equ 0x22
R3      equ 0x23
R4      equ 0x24

RES_VECT  CODE    0x0000            ; processor reset vector
    GOTO    START                   ; go to beginning of program

;*******************************************************************************
; MAIN PROGRAM
;*******************************************************************************

MAIN_PROG CODE                      ; let linker place main program

START
    BANKSEL PORTA
    CLRF PORTA ;Init PORTA
    BANKSEL TRISA ;
    CLRF TRISA ; set all bits in RA as output
```



```
LED_ON
    MOVLW B'01000000' ; RA6 to high
    MOVWF PORTA
    CALL DELAY

LED_OFF
    CLRF PORTA
    CALL DELAY
    GOTO LED_ON ; loop forever

DELAY
    movlw 0xff
    movwf R4
    goto DELAY4

DELAY0
    decfsz R0
    goto DELAY0
    goto DELAY4

DELAY1
    movlw 0xff
    movwf R0
    decfsz R1
    goto DELAY0
    return

DELAY2
    movlw 0xff
    movwf R1
    decfsz R2
    goto DELAY1
    return

DELAY3
    movlw 0xff
    movwf R2
    decfsz R3
    goto DELAY2
    return
```

Answer the following questions:

1. What is the length (number of bits) of a PIC16 instruction?
2. What is the length of the PIC18 program counter? How many different program memory locations can be addressed by the PIC16 MCU?
3. How many bits are used to select a data register?
4. What is the access bank? What benefits does the access bank provide?
5. What is an assembler? What is a compiler?
6. What is instruction pipelining? What benefits does it provide?

Below video will guide you on how to create a project and run it on the MPLAB:

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=104#oembed-1

ASSIGNMENT #1

Assignment #1

1. Write an instruction sequence to swap the contents of data registers at Ox300 and Ox200.
2. Write an instruction sequence to load the value of Ox39 into data memory locations Oxl OO-Oxl03.
3. Write an instruction sequence to subtract 10 from data memory locations Ox30-0x34.
4. Write an instruction to store the contents of the WREG register in the data register located at Ox25 of bank 4.
5. Write an instruction sequence to copy the contents of data memory at Oxl00-0xl03 to Ox200-0x203 using the postincrement addressing mode.
6. Write an instruction sequence to copy the contents of data memory at Oxl00-0xl03 to Ox203-0x200, that is, in reverse order, by combining the use of postincrement and postdecrement modes.
7. Write an instruction sequence to add 5 to data registers at $300–$303.
8. Write an instruction sequence to add the contents of the data registers at $10 and $20 and store the sum at $30.
9. Write an instruction sequence to subtract the contents of the data register at Ox20 from that of the data register at Ox30 and store the difference in the data register at OxlO.

Reference:

Chartrand, L., & Huang, H. W. (2005). (PIC microconteroller: an introduction to software & hardware interfacing.)

PART III

ASSEMBLY LANGUAGE PROGRAMMING - I

In this module, students will delve into the fundamentals of assembly language programming, focusing on the 8085 microprocessor. The module is structured to provide both theoretical knowledge and practical experience through a series of lectures and lab assignments. By the end of this module, students will have a solid understanding of the 8085 instruction set and the ability to implement basic assembly programs.

8085 Instruction Set: Part 1

The first part of the instruction set will cover the basic categories of instructions used in the 8085 microprocessor. Students will learn about:

- **Data Transfer Instructions:** Instructions that move data between registers, memory, and I/O ports.
- **Arithmetic Instructions:** Instructions that perform arithmetic operations like addition, subtraction, increment, and decrement.
- **Logical Instructions:** Instructions that perform bitwise logical operations such as AND, OR, XOR, and complement.
- **Branch Instructions:** Instructions that alter the flow of execution, including jumps, calls, returns, and restarts.
- **Stack, I/O, and Machine Control Instructions:** Instructions that manage the stack, perform I/O operations, and control the microprocessor's operation.

8085 Instruction Set: Part 2

The second part of the instruction set will delve deeper into the advanced instructions and their applications. Students will explore:

- **Advanced Data Manipulation:** Instructions for more complex data handling and manipulation.
- **Conditional and Unconditional Branching:** Detailed use of branching instructions for creating loops and conditional execution.
- **Interrupt Handling:** Instructions and techniques for managing interrupts and implementing

interrupt service routines.

- **Timing and Delay Instructions:** Instructions used for creating precise timing delays, which are essential for various applications.

Lab #4: Code for DELAY – Use the Delay Code with Blinking LED Program

In this lab assignment, students will write an assembly program to implement a delay routine and use it to blink an LED. The lab will cover:

- **Writing Delay Routines:** Techniques for creating accurate delay loops using the 8085 instruction set.
- **LED Control:** Using I/O ports to control an LED, turning it on and off at regular intervals.
- **Combining Delay and LED Control:** Integrating the delay routine with LED control to create a blinking effect.

Objectives:

- Understand how to create delay loops in assembly language.
- Learn to control I/O ports to manipulate external hardware.
- Gain practical experience in writing and debugging assembly programs.

Lab #5: Blink Two LEDs at a Time

In this lab assignment, students will extend their knowledge by writing an assembly program to blink two LEDs alternately. The lab will cover:

- **Controlling Multiple LEDs:** Using multiple I/O ports to control more than one LED.
- **Synchronized Blinking:** Creating a program that alternates the blinking of two LEDs.
- **Optimizing Delay Routines:** Refining delay routines to ensure synchronized operation.

Objectives:

- Learn to control multiple I/O ports simultaneously.
- Understand the importance of synchronized timing in embedded systems.
- Develop skills in writing more complex assembly programs.

8085 INSTRUCTION SET: PART 1

Introduction

The 8085 microprocessor, developed by Intel, has a rich set of instructions that allow it to perform a wide range of operations. These instructions can be broadly classified into several categories, each serving a specific purpose in the operation of the microprocessor. Understanding these instructions is crucial for programming the 8085 effectively.

Categories of Instructions

The 8085 instruction set can be divided into the following categories:

1. **Data Transfer Instructions**
2. **Arithmetic Instructions**
3. **Logical Instructions**
4. **Branch Instructions**
5. **Stack, I/O, and Machine Control Instructions**

1. Data Transfer Instructions

Data transfer instructions are used to move data between registers, between memory and registers, or between I/O ports and the accumulator. These instructions do not alter the data being transferred.

Common Data Transfer Instructions:

Opcode	Operand	Description
MOV	Rd, Rs	Copy the contents of the source register (Rs) to the destination register (Rd).
MOV	Rd, M	Copy the contents of the memory location pointed to by the HL pair to the destination register (Rd).
MOV	M, Rs	Copy the contents of the source register (Rs) to the memory location pointed to by the HL pair.
MVI	Rd, Data	Move immediate 8-bit data to the destination register (Rd).
MVI	M, Data	Move immediate 8-bit data to the memory location pointed to by the HL pair.
LXI	Reg. pair, 16-bit data	Load immediate 16-bit data into the register pair.
LDA	16-bit address	Load the contents of the specified memory location into the accumulator.
STA	16-bit address	Store the contents of the accumulator into the specified memory location.
LHLD	16-bit address	Load H and L registers directly from the specified memory locations.
SHLD	16-bit address	Store H and L registers directly into the specified memory locations.
LDAX	B/D	Load the accumulator indirectly from the memory location pointed to by the specified register pair (B or D).
STAX	B/D	Store the contents of the accumulator into the memory location pointed to by the specified register pair (B or D).
XCHG	None	Exchange the contents of the H and L registers with the contents of the D and E registers.

2. Arithmetic Instructions

Arithmetic instructions perform arithmetic operations such as addition, subtraction, increment, and decrement on the data in registers or memory.

Common Arithmetic Instructions:

Opcode	Operand	Description
ADD	R	Add the contents of the specified register (R) to the accumulator.
ADD	M	Add the contents of the memory location pointed to by the HL pair to the accumulator.
ADI	Data	Add immediate 8-bit data to the accumulator.
ADC	R	Add the contents of the specified register (R) to the accumulator along with the carry flag.
ADC	M	Add the contents of the memory location pointed to by the HL pair to the accumulator along with the carry flag.
ACI	Data	Add immediate 8-bit data to the accumulator along with the carry flag.
SUB	R	Subtract the contents of the specified register (R) from the accumulator.
SUB	M	Subtract the contents of the memory location pointed to by the HL pair from the accumulator.
SUI	Data	Subtract immediate 8-bit data from the accumulator.
SBB	R	Subtract the contents of the specified register (R) and the borrow flag from the accumulator.
SBB	M	Subtract the contents of the memory location pointed to by the HL pair and the borrow flag from the accumulator.
SBI	Data	Subtract immediate 8-bit data and the borrow flag from the accumulator.
INR	R	Increment the contents of the specified register (R) by 1.
INR	M	Increment the contents of the memory location pointed to by the HL pair by 1.
DCR	R	Decrement the contents of the specified register (R) by 1.
DCR	M	Decrement the contents of the memory location pointed to by the HL pair by 1.
INX	Reg. pair	Increment the contents of the specified register pair by 1.
DCX	Reg. pair	Decrement the contents of the specified register pair by 1.
DAD	Reg. pair	Add the contents of the specified register pair to the contents of the HL pair.
DAA	None	Decimal adjust the accumulator.

3. Logical Instructions

Logical instructions perform bitwise logical operations such as AND, OR, XOR, and complement on the data in registers or memory.**Common Logical Instructions:**

Opcode	Operand	Description
ANA	R	Logical AND the contents of the specified register (R) with the accumulator.
ANA	M	Logical AND the contents of the memory location pointed to by the HL pair with the accumulator.
ANI	Data	Logical AND immediate 8-bit data with the accumulator.
XRA	R	Logical XOR the contents of the specified register (R) with the accumulator.
XRA	M	Logical XOR the contents of the memory location pointed to by the HL pair with the accumulator.
XRI	Data	Logical XOR immediate 8-bit data with the accumulator.
ORA	R	Logical OR the contents of the specified register (R) with the accumulator.
ORA	M	Logical OR the contents of the memory location pointed to by the HL pair with the accumulator.
ORI	Data	Logical OR immediate 8-bit data with the accumulator.
CMP	R	Compare the contents of the specified register (R) with the accumulator.
CMP	M	Compare the contents of the memory location pointed to by the HL pair with the accumulator.
CPI	Data	Compare immediate 8-bit data with the accumulator.
CMA	None	Complement the contents of the accumulator.
CMC	None	Complement the carry flag.
STC	None	Set the carry flag.
RLC	None	Rotate the contents of the accumulator left through the carry flag.
RRC	None	Rotate the contents of the accumulator right through the carry flag.
RAL	None	Rotate the contents of the accumulator left.
RAR	None	Rotate the contents of the accumulator right.

Examples of Data Transfer Instructions

1. **MOV Instruction**:
   - `MOV A, B`: Copies the contents of register B into register A.
   - `MOV M, A`: Copies the contents of the accumulator into the memory location pointed by the H-L pair.
2. **MVI Instruction**:

  - `MVI A, 0x32`: Loads the immediate value 0x32 into register A.
  - `MVI M, 0x45`: Loads the immediate value 0x45 into the memory location pointed by the H-L pair.

3. **LDA and STA Instructions**:

  - `LDA 2500H`: Loads the contents of memory location 2500H into the accumulator.
  - `STA 2500H`: Stores the contents of the accumulator into memory location 2500H.

4. **LXI Instruction**:

  - `LXI H, 2500H`: Loads the immediate value 2500H into the H-L pair.

5. **LDAX and STAX Instructions**:

  - `LDAX B`: Loads the contents of the memory location pointed by the B-C pair into the accumulator.
  - `STAX D`: Stores the contents of the accumulator into the memory location pointed by the D-E pair.

Examples of Arithmetic Instructions

1. **ADD Instruction**:

  - `ADD B`: Adds the contents of register B to the accumulator.

2. **ADI Instruction**:

  - `ADI 0x12`: Adds the immediate value 0x12 to the accumulator.

3. **SUB Instruction**:

  - `SUB C`: Subtracts the contents of register C from the accumulator.

4. **SUI Instruction**:

  - `SUI 0x10`: Subtracts the immediate value 0x10 from the accumulator.

5. **INR and DCR Instructions**:
    - `INR A`: Increments the contents of register A by one.
    - `DCR B`: Decrements the contents of register B by one.
6. **INX and DCX Instructions**:
    - `INX H`: Increments the contents of the H-L pair by one.
    - `DCX D`: Decrements the contents of the D-E pair by one.
7. **DAD Instruction**:
    - `DAD B`: Adds the contents of the B-C pair to the H-L pair.

Summary

In this part of the reading material, we have covered the first three categories of the 8085 instruction set: Data Transfer Instructions, Arithmetic Instructions, and Logical Instructions. These instructions form the core of the operations that the 8085 microprocessor can perform, allowing it to manipulate data, perform calculations, and execute logical operations.In the next part, we will explore the remaining categories: Branch Instructions and Stack, I/O, and Machine Control Instructions. Understanding these instructions will provide a comprehensive view of the capabilities of the 8085 microprocessor.

To enhance understanding, below are the lecture videos that may help in completing the lab assignment.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=132#oembed-1

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=132#oembed-2

8085 INSTRUCTION SET: PART 2

Branch Instructions

Branch instructions in the 8085 microprocessor allow the program to change its execution sequence. These instructions are crucial for implementing decision-making and loop structures in assembly language programs.

Types of Branch Instructions:

1. **Unconditional Jump Instructions:**
    - **JMP**: Unconditional jump to a specified address.
    - **JC**: Jump if carry flag is set.
    - **JNC**: Jump if carry flag is not set.
    - **JZ**: Jump if zero flag is set.
    - **JNZ**: Jump if zero flag is not set.

2. **Conditional Jump Instructions:**

Opcode	**Description**
JC	Jump if Carry flag is set
JNC	Jump if Carry flag is not set
JZ	Jump if Zero flag is set
JNZ	Jump if Zero flag is not set
JP	Jump if Sign flag is not set (positive)
JM	Jump if Sign flag is set (minus)
JPE	Jump if Parity flag is set (even parity)
JPO	Jump if Parity flag is not set (odd parity)

3. **Call Instructions:**
    - **CALL**: Unconditional call to a subroutine.
    - **CC**: Call if carry flag is set.
    - **CNC**: Call if carry flag is not set.

    - **CZ**: Call if zero flag is set.
    - **CNZ**: Call if zero flag is not set.

4. **Return Instructions:**

    - **RET**: Unconditional return from a subroutine.
    - **RC**: Return if carry flag is set.
    - **RNC**: Return if carry flag is not set.
    - **RZ**: Return if zero flag is set.
    - **RNZ**: Return if zero flag is not set.

5. **Compare Instructions**:

    - **CMP**: Compares a register/memory with the accumulator.
    - **CPI**: Compares immediate data with the accumulator.

6. **Rotate Instructions**:

    - **RLC**: Rotates the accumulator left through the carry bit.
    - **RRC**: Rotates the accumulator right through the carry bit.
    - **RAL**: Rotates the accumulator left without the carry bit.
    - **RAR**: Rotates the accumulator right without the carry bit.

7. **Restart Instructions:**

    - RST n: Calls one of eight subroutines located in the first 64 bytes of memory.

5. Stack, I/O, and Machine Control Instructions

These instructions manage the stack, perform I/O operations, and control various aspects of the microprocessor's operation.**Stack Instructions:**

- PUSH rp: Push register pair onto stack.
- POP rp: Pop top of stack to register pair.
- XTHL: Exchange top of stack with HL.
- SPHL: Move HL to stack pointer.

I/O Instructions:

- IN port: Input from I/O port to accumulator.
- OUT port: Output accumulator to I/O port.

Machine Control Instructions:

- HLT: Halt the processor.
- NOP: No operation (used for timing or to replace instructions).
- DI: Disable interrupts.
- EI: Enable interrupts.
- RIM: Read interrupt mask.
- SIM: Set interrupt mask.

Example: Using Branch Instructions

Here's an example of how branch instructions can be used to create a simple loop:
text

```
     MVI C, 5    ; Initialize counter
LOOP:
     ; Your loop body here
     DCR C ; Decrement counter
     JNZ LOOP ; Jump to LOOP if counter is not zero
     ; Continue with the rest of the program
```

This code will execute the loop body 5 times before continuing with the rest of the program.

Examples of Logical Instructions

1. **ANA Instruction**:
    - `ANA B`: Performs a bitwise AND operation between the contents of register B and the accumulator.
2. **XRA Instruction**:
    - `XRA C`: Performs a bitwise XOR operation between the contents of register C and the accumulator.

3. **CMP Instruction**:
    - `CMP D`: Compares the contents of register D with the accumulator.

Examples of Branch Instructions

1. **JMP Instruction**:
    - `JMP 2000H`: Jumps unconditionally to the address 2000H.
2. **JC Instruction**:
    - `JC 2500H`: Jumps to the address 2500H if the carry flag is set.
3. **CALL Instruction**:
    - `CALL 3000H`: Calls the subroutine located at address 3000H.
4. **RET Instruction**:
    - `RET`: Returns unconditionally from a subroutine.

Summary

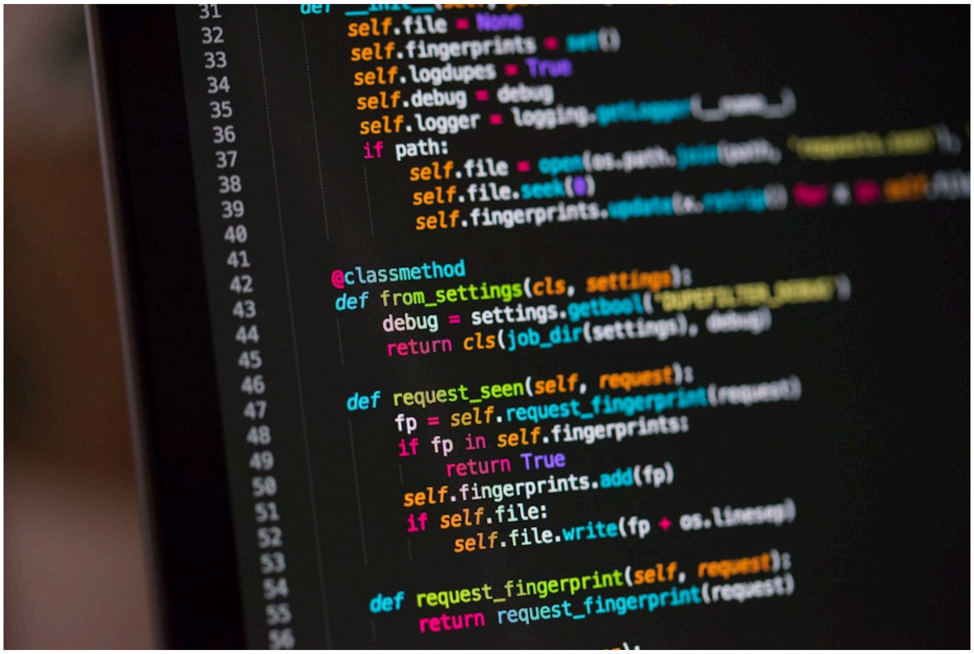
Understanding these instructions is important for effective assembly language programming on the 8085 microprocessor. Branch instructions allow for complex program flow control, while stack and machine control instructions provide essential functionality for managing subroutines, interrupts, and overall program execution.Remember that efficient use of these instructions requires careful consideration of program flow and system resources. Practice and experience will help in choosing the most appropriate instructions for specific programming tasks.

To enhance your understanding and to complete the lab assignment, please see the video lectures below:

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=134#oembed-1

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=134#oembed-2

LAB #4 : CODE FOR DELAY



This report requires you to write down the assembly code for delay (use the delay code with the blinking LED program). Use the following instructions:

1. Blinking LED code available from the lecture below for your review
2. Make the block diagram for the delay code which explains the algorithm you used
3. Based on the algorithm create an assembly code to provide a delay
4. Once the Delay code is created then add it to the existing blinking LED code from the lecture
5. Write a report stating the whole process, the difficulty, and tips you found in that process.

LAB #5 : BLINK TWO LEDS AT A TIME

In this Lab, you will be creating your own code for blinking two LEDs at a time, follow the instructions for this lab below:

1. Write a code that can turn the two LEDs (RA4 & RA6) ON at a time while LEDs (RA5 & RA7) are OFF and vice versa in the second cycle
2. Make a block diagram or write a step-by-step algorithm for this program
3. Write a report stating the whole process, the difficulty, and tips you found in that process.

PART IV ASSEMBLY LANGUAGE PROGRAMMING - II

In this module, students will further explore assembly language programming, focusing on the PIC16F18875 microcontroller. This module builds on foundational knowledge and introduces more advanced programming techniques and practical applications. By the end of this module, students will have a thorough understanding of the PIC16F18875 instruction set and the ability to implement complex assembly programs.

Part 1: Introduction to PIC16F18875 Instruction Set

The first part of the instruction set will cover the basic categories of instructions used in the PIC16F18875 microcontroller. Students will learn about:

- **Data Transfer Instructions:** Instructions that move data between registers, memory, and I/O ports.
- **Arithmetic Instructions:** Instructions that perform arithmetic operations like addition, subtraction, increment, and decrement.
- **Logical Instructions:** Instructions that perform bitwise logical operations such as AND, OR, XOR, and complement.
- **Control Instructions:** Instructions that manage program flow, including jumps, calls, returns, and resets.
- **Special Instructions:** Instructions for specific microcontroller operations, such as sleep, watchdog timer reset, and no operation.

Part 2: Detailed PIC16F18875 Instruction Set and Programming Techniques

The second part of the instruction set will delve deeper into the advanced instructions and their applications. Students will explore:

- **Advanced Data Manipulation:** Instructions for more complex data handling and manipulation.
- **Conditional and Unconditional Branching:** Detailed use of branching instructions for creating loops and conditional execution.

- **Interrupt Handling:** Instructions and techniques for managing interrupts and implementing interrupt service routines.
- **Timing and Delay Instructions:** Instructions used for creating precise timing delays, which are essential for various applications.

Part 3: Advanced Instructions and Practical Applications for PIC16F18875 Assembly Programming

In this part, students will learn about the advanced instructions and their practical applications. Topics will include:

- **Rotate and Shift Instructions:** Used for bit manipulation, essential in applications such as cryptography and data compression.
- **Bit-Oriented Instructions:** Precise control over individual bits in registers, crucial for tasks like setting flags, toggling LEDs, or reading sensor states.
- **Literal and Control Instructions:** Immediate data manipulation and controlling the flow of the program.
- **Practical Applications:** Implementing delays, reading and writing to I/O ports, using interrupts, and implementing finite state machines.

Part 4: Instruction Set for 16F18875 Specific to Lab Assignments and Video Lectures

This part will focus on the specific instructions and techniques required for the lab assignments and video lectures. Students will learn how to apply the instruction set to real-world scenarios, including:

- **Lab #6: Rotating LED to RIGHT with Consistency:** Writing an assembly program to rotate an LED to the right consistently.
- **Lab #7: Rotate LED on Each Button Press:** Writing an assembly program to rotate an LED each time a button is pressed.

Lab #6: Rotating LED to RIGHT with Consistency

In this lab assignment, students will write an assembly program to rotate an LED to the right consistently. The lab will cover:

- **Configuring I/O Ports:** Setting up the microcontroller ports for LED control.
- **Using Rotate Instructions:** Implementing rotate instructions to shift the LED pattern to the right.
- **Creating Delay Routines:** Adding delay loops to control the speed of the LED rotation.

Objectives:

- Understand how to configure I/O ports for LED control.
- Learn to use rotate instructions for bit manipulation.
- Gain practical experience in writing and debugging assembly programs with delays.

Lab #7: Rotate LED on Each Button Press

In this lab assignment, students will extend their knowledge by writing an assembly program to rotate an LED each time a button is pressed. The lab will cover:

- **Configuring I/O Ports for Input:** Setting up the microcontroller ports to read button presses.
- **Debouncing Button Inputs:** Implementing debouncing techniques to avoid false triggering.
- **Using Rotate Instructions:** Implementing rotate instructions to shift the LED pattern on each button press.

Objectives:

- Learn to configure I/O ports for both input and output.
- Understand the importance of debouncing in reading button inputs.
- Develop skills in writing assembly programs that respond to external inputs.

PART 1: INTRODUCTION TO PIC16F18875 INSTRUCTION SET

Overview of PIC16F18875 Architecture

The PIC16F18875 is part of Microchip's mid-range core devices, featuring a 14-bit wide instruction word. This architecture provides a balance between code density, performance, and flexibility, making it suitable for a wide range of embedded applications.Key features of the PIC16F18875 instruction set:

- 14-bit instruction word
- 49 instructions (in the enhanced mid-range core)
- Harvard architecture (separate program and data memory)
- 8-bit data path
- 16-bit wide program memory

2. Instruction Formats

The PIC16F18875 uses several instruction formats:

1. **Byte-oriented operations**
2. **Bit-oriented operations**
3. **Literal operations**
4. **Control operations**

Each format is designed to efficiently handle different types of operations, from data manipulation to program flow control.

3. Special Function Registers (SFRs)

The PIC16F18875 uses Special Function Registers for controlling various peripheral functions and device operations. These registers can often be used as source or destination operands in instructions, allowing direct manipulation of device settings and status.

4. Basic Instruction Categories

1. **Arithmetic and Logic Instructions**
    - ADD, SUB, AND, OR, XOR, etc.
2. **Data Movement Instructions**
    - MOV, MOVF, MOVWF, etc.
3. **Bit Manipulation Instructions**
    - BSF, BCF, BTFSS, BTFSC, etc.
4. **Program Control Instructions**
    - GOTO, CALL, RETURN, etc.
5. **Special Instructions**
    - SLEEP, CLRWDT, NOP, etc.

5. Addressing Modes

The PIC16F18875 supports several addressing modes:

1. **Direct Addressing**: The operator's address is part of the instruction
2. **Indirect Addressing**: Address is specified by a pointer register
3. **Immediate Addressing**: The operator value is part of the instruction

Below are the lecture videos to support the reading and lab assignments.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=140#oembed-1

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=140#oembed-2

PART 2: DETAILED PIC16F18875 INSTRUCTION SET AND PROGRAMMING TECHNIQUES

1. Arithmetic and Logic Instructions

detailed explanation of key instructions:

- **ADDWF f,d**: Add W and f
  ```
  ADDWF PORTA, W ; Add PORTA to W, store result in W.
  ```
- **ANDWF f,d**: AND W with f
  ```
  ANDWF STATUS, F ; AND W with STATUS, store in STATUS
  ```
- **COMF f,d**: Complement f
  ```
  COMF COUNT, F ; Complement COUNT, store in COUNT
  ```

2. Data Movement Instructions

- **MOVF f,d**: Move f
  ```
  MOVF TEMP, W ; Move TEMP to W
  ```
- **SWAPF f,d**: Swap nibbles in f
  ```
  SWAPF PORTB, F ; Swap nibbles in PORTB, store in PORTB
  ```

3. Bit Manipulation Instructions

- **BSF f,b**: Bit Set f
  ```
  BSF PORTA, 3 ; Set bit 3 in PORTA
  ```
- **BTFSC f, b**: Bit Test f, Skip if Clear
  ```
  BTFSC STATUS, Z ; Test Zero flag, skip next if clear.
  ```

4. Program Control Instructions

- **CALL k**: Call subroutine
  CALL DELAY ; Call subroutine named DELAY
- **GOTO k**: Go to address
  GOTO MAIN ; Jump to label MAIN

5. Special Instructions and Features

- **OPTION**: Load OPTION register (for compatibility with older PICs)
- **TRIS f**: Load TRIS register (also for compatibility)

6. Advanced Programming Techniques

Efficient Register Usage

- Use the Working (W) register for temporary storage to minimize memory access.
- Utilize the STATUS register flags (Z, C, DC) for conditional operations.

Bank Selection

The PIC16F18875 has multiple register banks. Use the `BANKSEL` directive for efficient bank switching:

```
BANKSEL TRISA ; Select bank containing TRISA
MOVLW 0xFF
MOVWF TRISA ; Set PORTA as all inputs
```

Interrupt Handling

Implement interrupt service routines (ISRs) using the `RETFIE` instruction:

```
ORG 0x0004 ; Interrupt vector location
GOTO ISR ; Jump to ISR
    ; ... (main program)
ISR:
  ; Interrupt handling code
```

```
RETFIE ; Return from interrupt
```

Loop Optimization

Use decrement and skip instructions for efficient loops:

```
MOVLW 10 ; Load loop counter
MOVWF COUNTER
LOOP:
; Loop body
DECFSZ COUNTER, F
GOTO LOOP
; Continue after loop
```

Bit Manipulation for I/O Control

Use bit-oriented instructions for efficient I/O control:

```
BSF LATA, 2 ; Set RA2 high (turn on LED)
BCF LATA, 2 ; Clear RA2 low (turn off LED)
BTFSC PORTA, 1 ; Check if RA1 is low (button pressed)
GOTO BUTTON_PRESSED
```

Summary

While the instruction set is compact, it provides all the necessary tools for complex embedded system development. Regular practice and reference to the official Microchip documentation will help in becoming proficient in PIC assembly programming. Remember to always consult the latest PIC16F18875 datasheet and programming manual for the most up-to-date and detailed information on instructions and their usage.

Below are the lecture videos to support the reading and lab assignments.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=147#oembed-1

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=147#oembed-2

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=147#oembed-3

Refrences:

[1] https://ww1.microchip.com/downloads/en/DeviceDoc/31029a.pdf

[2] https://forum.microchip.com/s/topic/a5C3l000000MbYLEA0/t374413

[3] https://ww1.microchip.com/downloads/en/DeviceDoc/PIC16%28L%29F18855_75%20Data%20Sheet_DS40001802E.pdf

[4] https://www.microchip.com/en-us/product/pic16f18875

[5] https://www.instructables.com/Programming-PIC-Microcontrollers/

PART 3: ADVANCED INSTRUCTIONS AND PRACTICAL APPLICATIONS FOR PIC16F18875 ASSEMBLY PROGRAMMING

1. Advanced Instructions

The PIC16F18875 microcontroller includes several advanced instructions that enhance its functionality and allow for more complex operations. These instructions are essential for efficient and powerful assembly programming.

1.1 Rotate and Shift Instructions

Rotate and shift instructions are used for bit manipulation, which is crucial in applications such as cryptography, data compression, and error detection.

- **RLF f,d**: Rotate Left through Carry
  ```
  RLF PORTA, F ; Rotate PORTA left through carry, store in PORTA
  ```
- **RRF f,d**: Rotate Right through Carry
  ```
  RRF PORTB, W ; Rotate PORTB right through carry, store in W
  ```
- **SWAPF f,d**: Swap Nibbles in f
  ```
  SWAPF TEMP, F ; Swap nibbles in TEMP, store in TEMP
  ```

1.2 Bit-Oriented Instructions

Bit-oriented instructions allow for precise control over individual bits in registers, which is essential for tasks like setting flags, toggling LEDs, or reading sensor states.

- **BSF f,b**: Bit Set f
  ```
  BSF STATUS, Z ; Set Zero flag in STATUS
  ```
- **BCF f,b**: Bit Clear f
  ```
  BCF STATUS, C ; Clear Carry flag in STATUS
  ```
- **BTFSC f,b**: Bit Test f, Skip if Clear

```
BTFSC PORTA, 1 ; Test bit 1 of PORTA, skip next if clear
```

- **BTFSS f,b**: Bit Test f, Skip if Set

  ```
  BTFSS PORTB, 0 ; Test bit 0 of PORTB, skip next if set
  ```

1.3 Literal and Control Instructions

These instructions are used for immediate data manipulation and controlling the flow of the program.

- **MOVLW k**: Move Literal to W

  ```
  MOVLW 0x55 ; Move 0x55 to W
  ```

- **ADDLW k**: Add Literal to W

  ```
  ADDLW 0x0A ; Add 10 to W
  ```

- **SUBLW k**: Subtract W from Literal

  ```
  SUBLW 0x20 ; Subtract W from 32, store in W
  ```

- **RETFIE**: Return from Interrupt

  ```
  RETFIE ; Return from interrupt, enable global interrupts
  ```

2. Practical Applications

Understanding how to apply these instructions in real-world scenarios is crucial for effective programming. Here are some practical applications of the PIC16F18875 instruction set.

2.1 Implementing Delays

Delays are essential in many embedded applications, such as blinking LEDs or creating time intervals for sensor readings.

```
; Delay subroutine (approx. 1 millisecond delay)
DELAY_MS:
   MOVLW D'250' ; Load W with 250
   MOVWF COUNT1 ; Move W to COUNT1
   MOVWF COUNT2 ; Move W to COUNT2
DELAY_LOOP:
   DECFSZ COUNT1, F ; Decrement COUNT1, skip if zero
   GOTO DELAY_LOOP ; Repeat loop
   DECFSZ COUNT2, F ; Decrement COUNT2, skip if zero
```

```
    GOTO DELAY_LOOP ; Repeat loop
    RETURN ; Return from subroutine
```

2.2 Reading and Writing to I/O Ports

Controlling I/O ports is fundamental in embedded systems for tasks such as reading sensor data or controlling actuators.

text

```
; Initialize PORTA as output
BANKSEL TRISA
CLRF TRISA ; Clear TRISA register (set PORTA as output); Set RA0 high
BSF LATA, 0 ; Set bit 0 of LATA (turn on LED connected to RA0); Read RA1 and store result in W
BANKSEL PORTA
BTFSC PORTA, 1 ; Test bit 1 of PORTA
GOTO BIT_SET
GOTO BIT_CLEARBIT_SET:
MOVLW 0x01 ; Load W with 1
GOTO CONTINUE

BIT_CLEAR:
MOVLW 0x00 ; Load W with 0
  CONTINUE:
; Continue with rest of the program
```

2.3 Using Interrupts

Interrupts allow the microcontroller to respond to external events promptly. Proper handling of interrupts is crucial for real-time applications.

```
ORG 0x0004 ; Interrupt vector location
GOTO ISR ; Jump to ISRISR:
; Handle interrupt
BCF INTCON, INTF ; Clear interrupt flag
; Perform interrupt-specific tasks
RETFIE ; Return from interrupt
```

2.4 Implementing Finite State Machines

Finite State Machines (FSMs) are used to model the behavior of systems with a finite number of states. They are widely used in control systems, user interfaces, and communication protocols.

```
; Define states
#define STATE_IDLE 0
#define STATE_PROCESS 1
#define STATE_ERROR 2; Initialize state
MOVLW STATE_IDLE
MOVWF STATEFSM_LOOP:
MOVF STATE, W
CPFSEQ STATE_IDLE
GOTO PROCESS_STATE
CPFSEQ STATE_PROCESS
GOTO ERROR_STATE
GOTO FSM_LOOPPROCESS_STATE:
; Processing code
GOTO FSM_LOOP

ERROR_STATE:
; Error handling code
GOTO FSM_LOOP
```

3. Best Practices for Assembly Programming

3.1 Code Organization and Commenting

Organize code into modules and use meaningful labels and comments to enhance readability and maintainability.

```
; Subroutine to initialize PORTA
Init_PORTA:
BANKSEL TRISA
CLRF TRISA ; Set PORTA as output
RETURN; Main programMAIN:
CALL Init_PORTA ; Initialize PORTA
; Other code
```

```
GOTO MAIN
```

3.2 Efficient Use of Registers and Memory

Minimize bank switching and optimize register usage to reduce memory access time.

```
BANKSEL TRISA ; Select bank containing TRISA
CLRF TRISA ; Clear TRISA registerBANKSEL PORTA ; Select bank containing PORTA
MOVLW 0xFF
MOVWF PORTA ; Set all PORTA pins high
```

3.3 Interrupt Handling

Keep Interrupt Service Routines (ISRs) short and efficient to minimize latency.

```
ORG 0x0004 ; Interrupt vector location
GOTO ISR ; Jump to ISRISR:
; Handle interrupt
BCF INTCON, INTF ; Clear interrupt flag
; Perform interrupt-specific tasks
RETFIE ; Return from interrupt
```

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=155#oembed-1

Refrences:

[1] https://ww1.microchip.com/downloads/en/DeviceDoc/31029a.pdf

[2] https://forum.microchip.com/s/topic/a5C3l000000MbYLEA0/t374413

[3] https://ww1.microchip.com/downloads/en/DeviceDoc/PIC16%28L%29F18855_75%20Data%20Sheet_DS40001802E.pdf

[4] https://www.microchip.com/en-us/product/pic16f18875

[5] https://www.instructables.com/Programming-PIC-Microcontrollers/

[6] https://www.youtube.com/watch?v=fCqq9OQmztY

[7] https://www.youtube.com/watch?v=8kA9t4Wv2Tk

[8] https://www.youtube.com/watch?v=7bZg_GzUbHI

[9] https://ww1.microchip.com/downloads/en/DeviceDoc/50002027E.pdf

PART 4: INSTRUCTION SET FOR 16F18875 SPECIFIC TO LAB ASSIGNMENTS AND VIDEO LECTURES

This section provides an overview of the crucial data transfer, arithmetic, logical, and bit-oriented instructions for the 16F18875 microcontroller, which are fundamental to the Lab Assignments and Lecture Videos. Some of these instruction sets might overlap with previous readings. Understanding these instructions is key for effective assembly language programming and practical application in various embedded systems projects.

Data Transfer Instructions

1. **MOVLW (Move Literal to W register)**:
    - Syntax: `MOVLW k`
    - Function: Loads the literal value `k` into the W register.
    - Example: `MOVLW 0x55` ; Loads the value 0x55 into the W register.
2. **MOVWF (Move W register to File register)**:
    - Syntax: `MOVWF f`
    - Function: Copies the contents of the W register to the specified file register `f`.
    - Example: `MOVWF 0x20` ; Copies the contents of the W register to file register 0x20.
3. **MOVF (Move File register to W register)**:
    - Syntax: `MOVF f, d`
    - Function: Moves the contents of the file register `f` to the W register if `d` is 0.
    - Example: `MOVF 0x20, W` ; Moves the contents of file register 0x20 to the W register.

Arithmetic Instructions

1. **ADDWF (Add W register to File register)**:

  - Syntax: `ADDWF f, d`
  - Function: Adds the contents of the W register to the file register `f`, and stores the result in `f` if `d` is 1.
  - Example: `ADDWF 0x20, F` ; Adds W register to file register 0x20, storing the result in 0x20.

2. **SUBWF (Subtract W register from File register)**:

    - Syntax: `SUBWF f, d`
    - Function: Subtracts the contents of the W register from the file register `f`, and stores the result in `W` if `d` is 0.
    - Example: `SUBWF 0x20, W` ; Subtracts W register from file register 0x20, storing the result in W.

3. **INCF (Increment File register)**:

    - Syntax: `INCF f, d`
    - Function: Increments the contents of the file register `f` by one and stores the result in `f` if `d` is 1.
    - Example: `INCF 0x20, F` ; Increments the contents of file register 0x20 by one, storing the result in 0x20.

4. **DECF (Decrement File register)**:

    - Syntax: `DECF f, d`
    - Function: Decrements the contents of the file register `f` by one and stores the result in `W` if `d` is 0.
    - Example: `DECF 0x20, W` ; Decrements the contents of file register 0x20 by one, storing the result in W.

Logical Instructions

1. **ANDWF (AND W register with File register)**:

    - Syntax: `ANDWF f, d`
    - Function: Performs a bitwise AND operation between the W register and the file register `f`, storing the result in `f` if `d` is 1.
    - Example: `ANDWF 0x20, F` ; Performs AND operation between W register and file register 0x20, storing result in 0x20.

2. **IORWF (Inclusive OR W register with File register)**:

- Syntax: `IORWF f, d`
- Function: Performs a bitwise OR operation between the W register and the file register `f`, storing the result in `f` if `d` is 1.
- Example: `IORWF 0x20, F` ; Performs OR operation between W register and file register 0x20, storing result in 0x20.

3. **XORWF (Exclusive OR W register with File register)**:

   - Syntax: `XORWF f, d`
   - Function: Performs a bitwise XOR operation between the W register and the file register `f`, storing the result in `f` if `d` is 1.
   - Example: `XORWF 0x20, F` ; Performs XOR operation between W register and file register 0x20, storing result in 0x20.

Bit-Oriented Instructions

1. **BCF (Bit Clear File register)**:

   - Syntax: `BCF f, b`
   - Function: Clears the specified bit `b` in the file register `f`.
   - Example: `BCF 0x20, 2` ; Clears bit 2 of file register 0x20.

2. **BSF (Bit Set File register)**:

   - Syntax: `BSF f, b`
   - Function: Sets the specified bit `b` in the file register `f`.
   - Example: `BSF 0x20, 3` ; Sets bit 3 of file register 0x20.

3. **BTFSC (Bit Test File register, Skip if Clear)**:

   - Syntax: `BTFSC f, b`
   - Function: Tests bit `b` in the file register `f` and skips the next instruction if the bit is clear.
   - Example: `BTFSC 0x20, 4` ; Tests bit 4 of file register 0x20, skips next instruction if bit is clear.

4. **BTFSS (Bit Test File register, Skip if Set)**:

   - Syntax: `BTFSS f, b`

- Function: Tests bit `b` in the file register `f` and skips the next instruction if the bit is set.
- Example: `BTFSS 0x20, 5` ; Tests bit 5 of file register 0x20, skips next instruction if bit is set.

Control Instructions

1. **NOP (No Operation)**:
   - Syntax: `NOP`
   - Function: Executes no operation and moves to the next instruction.
   - Example: `NOP` ; Simply does nothing for one instruction cycle.
2. **CLRWDT (Clear Watchdog Timer)**:
   - Syntax: `CLRWDT`
   - Function: Resets the watchdog timer to prevent a system reset.
   - Example: `CLRWDT` ; Resets the watchdog timer.
3. **SLEEP**:
   - Syntax: `SLEEP`
   - Function: Puts the microcontroller into sleep mode to save power.
   - Example: `SLEEP` ; Puts the microcontroller into sleep mode.
4. **RESET**:
   - Syntax: `RESET`
   - Function: Resets the microcontroller.
   - Example: `RESET` ; Resets the microcontroller.

Branching Instructions

1. **GOTO**:
   - Syntax: `GOTO k`
   - Function: Unconditionally branches to the specified address `k`.
   - Example: `GOTO 0x100` ; Jumps to address 0x100.

2. **CALL**:
    - Syntax: `CALL k`
    - Function: Calls a subroutine at the specified address `k`.
    - Example: `CALL 0x200` ; Calls the subroutine at address 0x200.
3. **RETURN**:
    - Syntax: `RETURN`
    - Function: Returns from a subroutine to the calling function.
    - Example: `RETURN` ; Returns from a subroutine.
4. **RETFIE (Return from Interrupt)**:
    - Syntax: `RETFIE`
    - Function: Returns from an interrupt service routine.
    - Example: `RETFIE` ; Returns from an interrupt service routine.
5. **Conditional Branch Instructions**:
    - **BRA**: Branches to a relative address.
    - **BRW**: Branches to the address in the W register.
    - **BC**: Branch if Carry bit is set.
    - **BNC**: Branch if Carry bit is not set.
    - **BZ**: Branch if Zero bit is set.
    - **BNZ**: Branch if Zero bit is not set.

Example Programs

- **LED Blinking**:

```
START:
BSF TRISB, 0 ; Set RB0 as input
BCF TRISB, 1 ; Set RB1 as output
   LOOP:
BTFSS PORTB, 0 ; Check if button is pressed
GOTO SKIP
```

```
BSF PORTB, 1 ; Turn on LED
GOTO LOOP
   SKIP:
BCF PORTB, 1 ; Turn off LED
GOTO LOOP
END
```

- **Button Press Detection**:

```
START:
BSF TRISB, 0 ; Set RB0 as input
BCF TRISB, 1 ; Set RB1 as output
   CHECK_BUTTON:
BTFSS PORTB, 0 ; Check if button is pressed
GOTO BUTTON_PRESSED
GOTO CHECK_BUTTON
   BUTTON_PRESSED:
BSF PORTB, 1 ; Turn on LED
GOTO CHECK_BUTTON
   END
```

- **Simple Subroutine**:

```
MAIN:
CALL SUBROUTINE
GOTO MAIN
   SUBROUTINE:
BSF PORTB, 1 ; Turn on LED
RETURN
   END
```

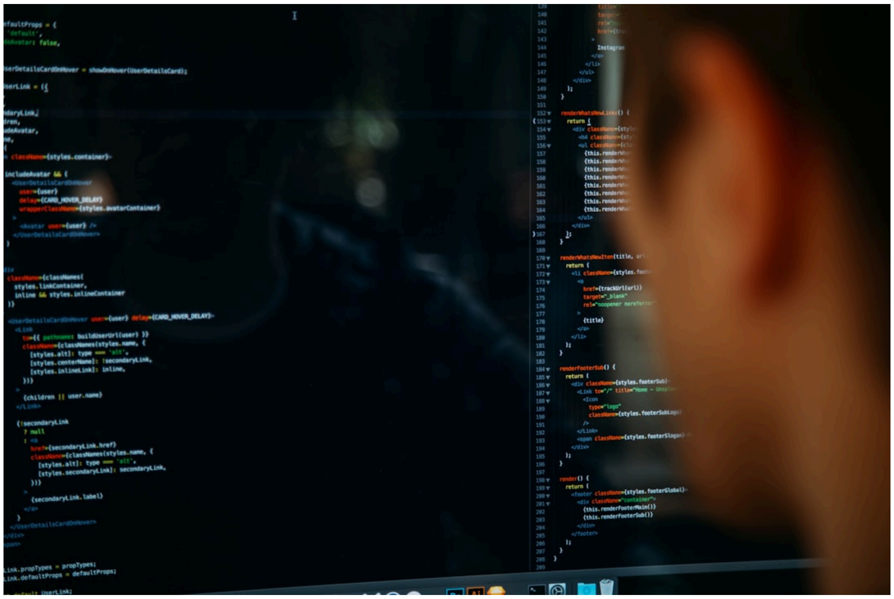
One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=158#oembed-1

LAB #7: ROTATE LED ON EACH BUTTON PRESS

In this Lab, you will be creating your own code for Rotating LED on each button press, follow the instructions for this lab below:

1. Write a code that can rotate the LED whenever the button is pressed.
2. Rotate the button press from left (RA7) to right (RA4)
3. Make a block diagram or write a step by step algorithm for this program
4. Write a report stating the whole process, difficulty, tips you found in that process.

LAB #6: ROTATING LED TO RIGHT WITH CONSISTENC



In this Lab, you will be creating your own code for RIGHT rotating LED consistency, follow the instructions for this lab below:

1. Write a code that can turn the LED on the development board towards the right (from RA7 to RA4) with consistency.
2. Make a block diagram or write a step by step algorithm for this program
3. Write a report stating the whole process, difficulty, tips you found in that process.
4. Attach the report, asm file of your code for me to test, and a demo video of the running code (optional)

PART V
ASSEMBLY PROGRAMMING -III

In this module, students will explore advanced assembly programming concepts, focusing on the PIC16(L)F18855/75 microcontroller and its ADC2 module. This module is designed to provide an in-depth understanding of the ADC2 functionalities and configurations, along with practical lab assignments that reinforce theoretical knowledge through hands-on projects.

Part 1: Introduction to PIC16(L)F18855/75 ADC2 Module

This section introduces the ADC2 module, which allows the conversion of analog input signals to 10-bit binary representations. Key features include:

- **Sample and Hold Circuit:** Captures and holds the analog input voltage.
- **Successive Approximation Converter:** Performs the analog-to-digital conversion.
- **Result Registers (ADRESH:ADRESL):** Store the conversion results.
- **Advanced Features:** 8-bit acquisition timer, capacitive voltage divider (CVD) support, automatic repeat and sequencing, computation features such as averaging and low-pass filter functions, and selectable interrupts.

Part 2: Detailed Configuration and Operation of the PIC16(L)F18855/75 ADC2 Module

This section covers the detailed configuration and operation of the ADC2 module, including:

- **Port Configuration:** Setting the I/O pins for analog input.
- **Channel Selection:** Choosing the appropriate analog input channel.
- **Voltage Reference Selection:** Configuring the positive and negative voltage references.
- **Conversion Clock Source:** Selecting the clock source for the ADC.
- **Interrupt Control:** Enabling and handling ADC interrupts.
- **Result Formatting:** Choosing between left-justified and right-justified formats for the 10-bit conversion result.
- **Starting and Completing a Conversion:** Procedures for initiating and completing an ADC conversion.

Readings Specific to Lab Assignments

This section provides readings and resources specific to the lab assignments, helping students understand the practical applications of the ADC2 module and other microcontroller features.

Lab #8: Make Three Programs Using External Circuit

In this lab assignment, students will write three assembly programs to interact with external circuits. The lab will cover:

- **Configuring I/O Ports:** Setting up the microcontroller ports for external circuit control.
- **Using External Inputs and Outputs:** Implementing programs that read from and write to external circuits.
- **Creating Delay Routines:** Adding delay loops to control the timing of interactions with external circuits.

Objectives:

- Understand how to configure I/O ports for external circuit control.
- Learn to use external inputs and outputs in assembly programs.
- Gain practical experience in writing and debugging assembly programs with delays.

Lab #9: Home Security System (Alarm System)

In this lab assignment, students will design and implement a microcontroller-based home security system. The lab will cover:

- **Sensor Inputs:** Configuring the microcontroller to read inputs from door/window sensors and motion detectors.
- **User Interface:** Implementing a keypad for arming/disarming the system and an LCD display for system status.
- **Outputs:** Controlling LEDs and a piezo buzzer or siren for alarm indications.
- **Timing Considerations:** Implementing entry/exit delays and alarm activation delays.
- **State Machine:** Managing the system's behavior in different states (disarmed, armed, alarm triggered).

Objectives:

- Learn to configure sensor inputs and user interface components.
- Understand the importance of timing and state management in security systems.
- Develop skills in writing complex assembly programs for real-world applications.

Lab #10: Temperature Indicator

In this lab assignment, students will build and program a temperature indicator using the MCP9701 sensor and the ADC2 module. The lab will cover:

- **Analog-to-Digital Conversion:** Configuring the ADC2 module to read the temperature from the MCP9701 sensor.
- **Temperature Calculation:** Converting the ADC reading to a temperature value.
- **LED Indicators:** Using LEDs to indicate the temperature status (high, low, on target).

Objectives:

- Understand how to configure and use the ADC2 module for analog-to-digital conversion.
- Learn to calculate temperature values from ADC readings.
- Gain practical experience in writing assembly programs that interact with sensors and indicators.

These topics will be discussed in detail later in the module, providing students with both theoretical knowledge and practical skills in advanced assembly programming for the PIC16(L)F18855/75 microcontroller.

PART 1: INTRODUCTION TO PIC16(L)F18855/75 ADC2 MODULEC

The PIC16(L)F18855/75 microcontroller features an advanced Analog-to-Digital Converter with Computation (ADC2) module. This module not only converts analog input signals into 10-bit digital representations but also includes additional features for enhanced functionality. This reading material will introduce the key components and configuration options of the ADC2 module.

1. Overview of the ADC2 Module

The ADC2 module is designed to convert analog input signals into 10-bit digital values. It includes a sample and hold circuit, a successive approximation converter, and result registers to store the conversion results.

Key Components:

- **Sample and Hold Circuit:** Captures and holds the analog input voltage stable during conversion.
- **Successive Approximation Converter:** Performs the analog-to-digital conversion and generates a 10-bit binary result.
- **Result Registers (ADRESH:ADRESL):** Store the conversion results for further processing.

2. Advanced Features

The ADC2 module includes several advanced features to enhance its functionality:

2.1 8-bit Acquisition Timer

- Allows precise control of the sampling time.
- Ensures accurate conversion of high-impedance sources.

2.2 Capacitive Voltage Divider (CVD) Support

- **8-bit Precharge Timer:** Controls the precharge duration.
- **Adjustable Sample and Hold Capacitor Array:** Enhances the accuracy of capacitive measurements.
- **Guard Ring Digital Output Drive:** Reduces noise and interference.

2.3 Automatic Repeat and Sequencing

- **Automated Double Sample Conversion for CVD:** Facilitates capacitive measurements.
- **Two Sets of Result Registers:** Stores current and previous results for comparison.
- **Auto-Conversion Trigger:** Initiates conversions automatically based on predefined conditions.
- **Internal Retrigger:** Allows continuous sampling without software intervention.

2.4 Computation Features

- **Averaging and Low-Pass Filter Functions:** Smooths out noise and fluctuations in the input signal.
- **Reference Comparison:** Compares the conversion result with a reference value.
- **2-Level Threshold Comparison:** Triggers actions based on predefined threshold levels.
- **Selectable Interrupts:** Generates interrupts based on conversion completion or threshold comparison.

3. Voltage Reference Options

The ADC voltage reference is software-selectable and can be either internally generated or externally supplied. The positive and negative voltage references are controlled by the ADREF register.

Positive Voltage Reference Options:

- **VREF+ Pin:** External voltage reference.
- **VDD:** Supply voltage.
- **FVR 1.024V, 2.048V, 4.096V:** Fixed Voltage Reference levels.

Negative Voltage Reference Options:

- **VREF- Pin:** External voltage reference.
- **VSS:** Ground.

4. Interrupt Capabilities

The ADC2 module can generate interrupts upon:

- Completion of a conversion.
- Threshold comparison results.

These interrupts can wake the device from Sleep mode, enabling power-efficient operation in battery-powered applications.

5. Configuration Considerations

When configuring and using the ADC2 module, several factors must be considered:

5.1 Port Configuration

- Configure the I/O pin for analog input by setting the appropriate TRIS and ANSEL bits.

5.2 Channel Selection

- Select the desired analog input channel using the ADPCH register.
- Available channels include PORTA, PORTB, PORTC, PORTD (PIC16(L)F18875 only), PORTE (PIC16(L)F18875 only), Temperature Indicator, DAC output, and Fixed Voltage Reference.

5.3 Conversion Clock Source

- Select the conversion clock source via the ADCLK register and the ADCS bit of the ADCON0 register.
- Options include FOSC/(2*(n+1)) and FRC (dedicated RC oscillator).

5.4 Result Formatting

- The 10-bit conversion result can be left or right justified, controlled by the ADFRM0 bit of the ADCON0 register.

6. Starting and Completing a Conversion

6.1 Starting a Conversion

- Enable the ADC module by setting the ADON bit of the ADCON0 register.
- Start a conversion by setting the ADGO bit of the ADCON0 register, using an external trigger, or enabling continuous-mode retrigger.

6.2 Completing a Conversion

- Upon completion, the ADC module:
  - Clears the ADGO bit.
  - Sets the ADIF interrupt flag bit.
  - Updates the ADRESH:ADRESL result registers.

Summary

The ADC2 module in the PIC16(L)F18855/75 microcontroller provides advanced features for precise and efficient analog-to-digital conversion. Understanding its key components, configuration options, and advanced features is important for effectively utilizing its capabilities in analog signal processing and embedded system design.In the next part, we will delve into the detailed configuration and operation procedures of the ADC2 module, including practical examples and best practices for achieving accurate and reliable conversions.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=160#oembed-1

PART 2: DETAILED CONFIGURATION AND OPERATION OF THE PIC16(L)F18855/75 ADC2 MODULE

Detailed Configuration of the ADC2 Module

When configuring and using the ADC2 module, several functions must be considered to ensure accurate and efficient analog-to-digital conversions. These include port configuration, channel selection, voltage reference selection, conversion clock source, interrupt control, result formatting, and more.

1.1 Port Configuration

To use the ADC2 module for converting analog signals, the I/O pins must be configured appropriately:

- Set the associated TRIS bit to configure the pin as an input.
- Set the associated ANSEL bit to configure the pin as an analog input.

Example:

```
BANKSEL TRISA
BSF TRISA, 0 ; Set RA0 as input
BANKSEL ANSELA
BSF ANSELA, 0 ; Set RA0 as analog input
```

1.2 Channel Selection

The ADC2 module supports multiple input channels, which can be selected using the ADPCH register. Available channels include:

- Eight PORTA pins (RA<7:0>)
- Eight PORTB pins (RB<7:0>)
- Eight PORTC pins (RC<7:0>)
- Eight PORTD pins (RD<7:0>, PIC16(L)F18875 only)

- Three PORTE pins (RE<2:0>, PIC16(L)F18875 only)
- Temperature Indicator
- DAC output
- Fixed Voltage Reference (FVR)
- AVSS (ground)

Example:

```
  MOVLW 0x00 ; Select AN0 (RA0) as input channel
  MOVWF ADPCH
```

1.3 Voltage Reference Selection

The ADPREF bits of the ADREF register control the positive voltage reference, while the ADNREF bit controls the negative voltage reference. Options include:

- **Positive Voltage Reference:**
    - VREF+ pin
    - VDD
    - FVR 1.024V
    - FVR 2.048V
    - FVR 4.096V
- **Negative Voltage Reference:**
    - VREF- pin
    - VSS

Example:

```
MOVLW 0x03 ; Set positive reference to FVR 4.096V
MOVWF ADREF
```

1.4 Conversion Clock Source

The conversion clock source is selected via the ADCLK register and the ADCS bit of the ADCON0 register. Options include:

- FOSC/(2*(n+1)) (where n is from 0 to 63)
- FRC (dedicated RC oscillator)

Example:

```
MOVLW 0x03 ; Set ADC clock source to FOSC/8
MOVWF ADCLK
```

2. Starting and Completing a Conversion

2.1 Starting a Conversion

To start a conversion, the ADC module must be enabled, and the ADGO bit must be set. Conversions can be initiated by:

- Software setting the ADGO bit
- An external trigger
- Continuous-mode retrigger

Example:

```
BSF ADCON0, ADON ; Enable ADC module
BSF ADCON0, ADGO ; Start conversion
```

2.2 Completion of a Conversion

Upon completion of a conversion:

- The ADGO bit is cleared.
- The ADIF interrupt flag bit is set.
- The conversion result is stored in the ADRESH:ADRESL registers.

Example:

```
BTFSC ADCON0, ADGO ; Wait for conversion to complete
GOTO $-1 ; Loop until conversion is done
MOVF ADRESH, W ; Read upper 2 bits of result
MOVWF RESULT_HIGH
MOVF ADRESL, W ; Read lower 8 bits of result
MOVWF RESULT_LOW
```

3. Handling ADC Results

3.1 Result Formatting

The 10-bit ADC conversion result can be formatted as either left justified or right justified, controlled by the ADFRM0 bit of the ADCON0 register.

Example:

```
BANKSEL ADCON0
BSF ADCON0, ADFRM0 ; Set result format to right justified
```

3.2 Interrupt Handling

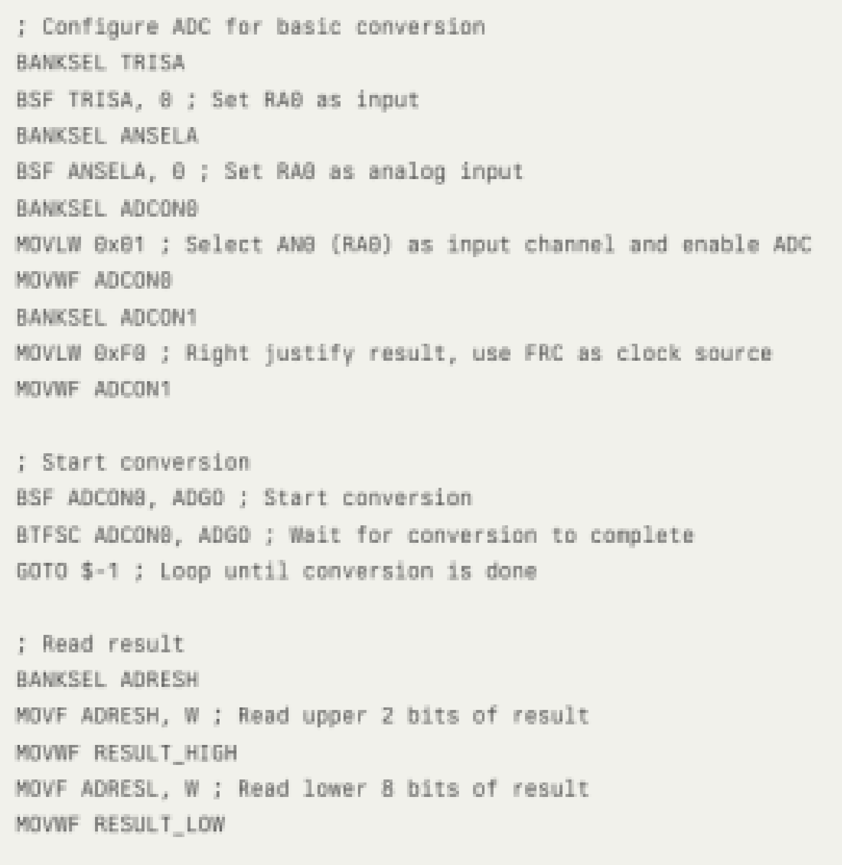
The ADC module can generate interrupts upon completion of a conversion. The ADIF bit in the PIR1 register indicates the interrupt flag, and the ADIE bit in the PIE1 register enables the interrupt.Example:

text

```
BANKSEL PIR1
BCF PIR1, ADIF ; Clear ADC interrupt flag
BANKSEL PIE1
BSF PIE1, ADIE ; Enable ADC interrupt
BANKSEL INTCON
BSF INTCON, PEIE ; Enable peripheral interrupts
BSF INTCON, GIE ; Enable global interrupts
```

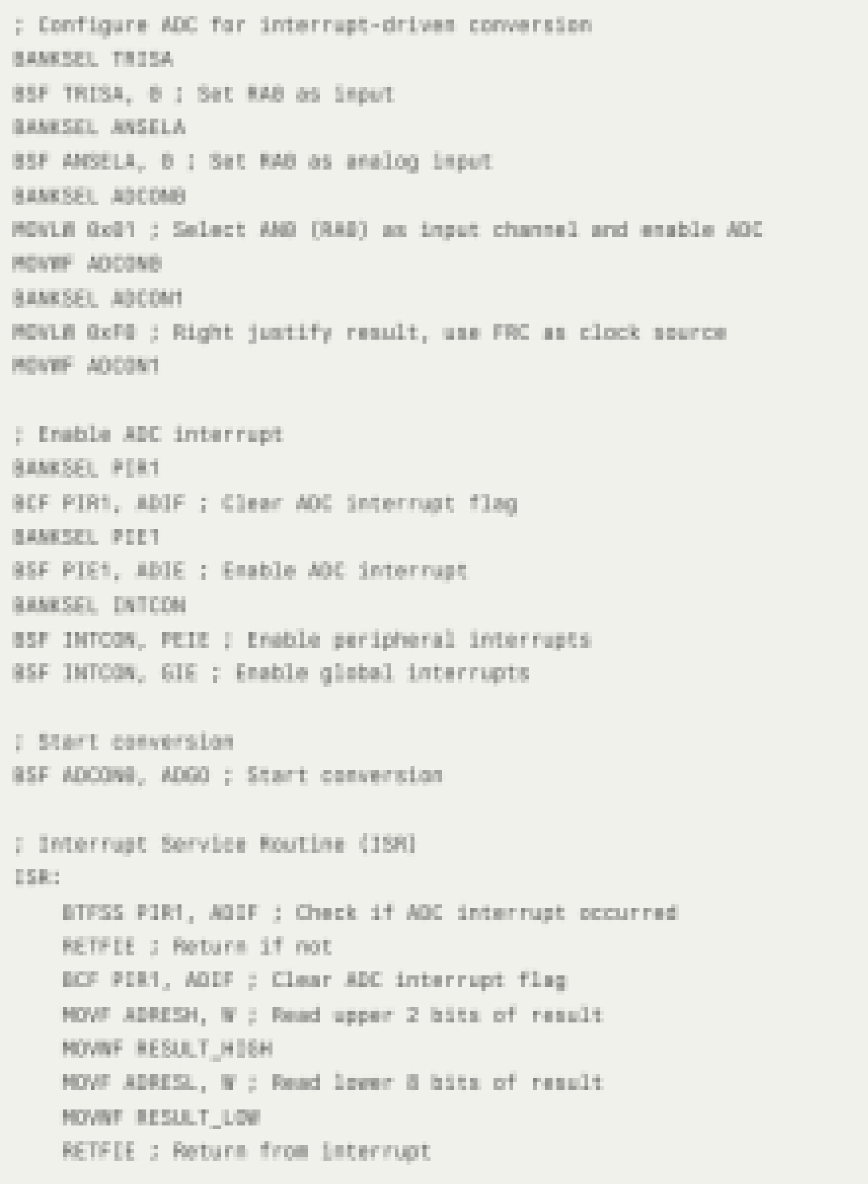
4. Practical Examples

4.1 Basic ADC Conversion



```
; Configure ADC for basic conversion
BANKSEL TRISA
BSF TRISA, 0 ; Set RA0 as input
BANKSEL ANSELA
BSF ANSELA, 0 ; Set RA0 as analog input
BANKSEL ADCON0
MOVLW 0x01 ; Select AN0 (RA0) as input channel and enable ADC
MOVWF ADCON0
BANKSEL ADCON1
MOVLW 0xF0 ; Right justify result, use FRC as clock source
MOVWF ADCON1

; Start conversion
BSF ADCON0, ADGO ; Start conversion
BTFSC ADCON0, ADGO ; Wait for conversion to complete
GOTO $-1 ; Loop until conversion is done

; Read result
BANKSEL ADRESH
MOVF ADRESH, W ; Read upper 2 bits of result
MOVWF RESULT_HIGH
MOVF ADRESL, W ; Read lower 8 bits of result
MOVWF RESULT_LOW
```

4.2 Using Interrupts for ADC Conversion



```
; Configure ADC for interrupt-driven conversion
BANKSEL TRISA
BSF TRISA, 0 ; Set RA0 as input
BANKSEL ANSELA
BSF ANSELA, 0 ; Set RA0 as analog input
BANKSEL ADCON0
MOVLW 0x01 ; Select AN0 (RA0) as input channel and enable ADC
MOVWF ADCON0
BANKSEL ADCON1
MOVLW 0xF0 ; Right justify result, use FRC as clock source
MOVWF ADCON1

; Enable ADC interrupt
BANKSEL PIR1
BCF PIR1, ADIF ; Clear ADC interrupt flag
BANKSEL PIE1
BSF PIE1, ADIE ; Enable ADC interrupt
BANKSEL INTCON
BSF INTCON, PEIE ; Enable peripheral interrupts
BSF INTCON, GIE ; Enable global interrupts

; Start conversion
BSF ADCON0, ADGO ; Start conversion

; Interrupt Service Routine (ISR)
ISR:
    BTFSS PIR1, ADIF ; Check if ADC interrupt occurred
    RETFIE ; Return if not
    BCF PIR1, ADIF ; Clear ADC interrupt flag
    MOVF ADRESH, W ; Read upper 2 bits of result
    MOVWF RESULT_HIGH
    MOVF ADRESL, W ; Read lower 8 bits of result
    MOVWF RESULT_LOW
    RETFIE ; Return from interrupt
```

Summary

In this part, we have explored the detailed configuration and operation of the ADC2 module in the PIC16(L)F18855/75 microcontroller. By understanding how to configure ports, select channels, choose voltage references, set conversion clock sources, and handle interrupts, you can effectively utilize the ADC2 module for accurate and efficient analog-to-digital conversions.Regular practice and reference to the official Microchip documentation will help in becoming proficient in using the ADC2 module. Always consult the latest datasheet and programming manual for the most up-to-date and detailed information on the ADC2 module and its usage.

One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=162#oembed-1

READINGS SPECIFIC TO LAB ASSIGNMENTS

Lab 1: Home Security Alarm System

Introduction to Microcontroller-Based Security Systems

Microcontroller-based security systems have become increasingly popular for home and business applications due to their flexibility, cost-effectiveness, and ability to integrate multiple sensors and outputs. The PIC16F18875 microcontroller is well-suited for this type of project due to its ample I/O pins, built-in timers, and analog-to-digital conversion capabilities.

Key Components and Concepts

1. Sensor Inputs:
   - Door/window sensors (magnetic reed switches)
   - Motion detectors (PIR sensors)
   - Glass break detectors
2. User Interface:
   - Keypad for arming/disarming (3-digit code)
   - LCD display for system status
3. Outputs:
   - LEDs for visual indicators
   - Piezo buzzer or siren for audible alarm
   - Relay for external siren control
4. Timing Considerations:
   - Entry/exit delays (30 seconds)

    - Alarm activation delay (15 seconds)

5. State Machine:

    - Disarmed state
    - Armed state
    - Alarm triggered state

Implementation Tips

1. Use interrupts for sensor inputs to ensure quick response times.
2. Implement debouncing for keypad inputs to avoid false triggering.
3. Use timer interrupts for managing delays and timing functions.
4. Store the security code in EEPROM for persistence across power cycles.
5. Use a state machine approach to manage system behavior in different modes.

Assembly Language Considerations

1. Use meaningful labels and comments to improve code readability.
2. Utilize macros for repeated code segments to improve maintainability.
3. Be mindful of register bank selection when accessing special function registers.
4. Use bit-wise operations for efficient handling of individual I/O pins.

Lab 2: Temperature Indicator

Introduction to Analog Temperature Sensing

The MCP9701 is a linear active thermistor integrated circuit that provides a voltage output proportional to temperature. By using the PIC16F18875's analog-to-digital converter (ADC), we can measure this voltage and convert it to a temperature reading.

Key Components and Concepts

1. MCP9701 Temperature Sensor:

- Linear output: 19.5mV/°C (scale factor)
- 500mV output at 0°C (offset voltage)

2. Analog-to-Digital Conversion:

   - 10-bit ADC on PIC16F18875
   - Voltage reference selection

3. Temperature Calculation:

   - Converting ADC reading to voltage
   - Applying scale factor and offset to calculate temperature

4. LED Indicators:

   - Red LED for high temperature
   - Blue LED for low temperature
   - Green LED for on-target temperature

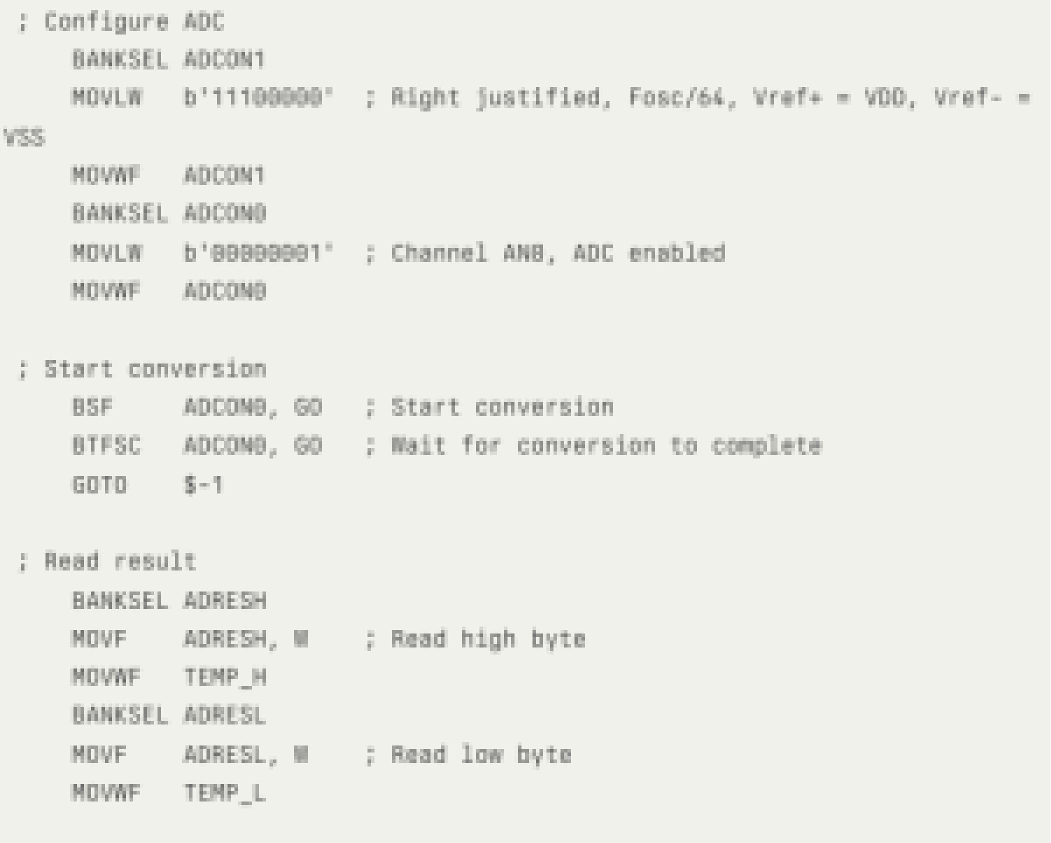
Implementation Tips

1. Configure the ADC for the appropriate voltage reference and acquisition time.
2. Use averaging of multiple ADC readings to reduce noise.
3. Implement hysteresis to prevent rapid switching between temperature states.
4. Use floating-point calculations for accurate temperature conversion.

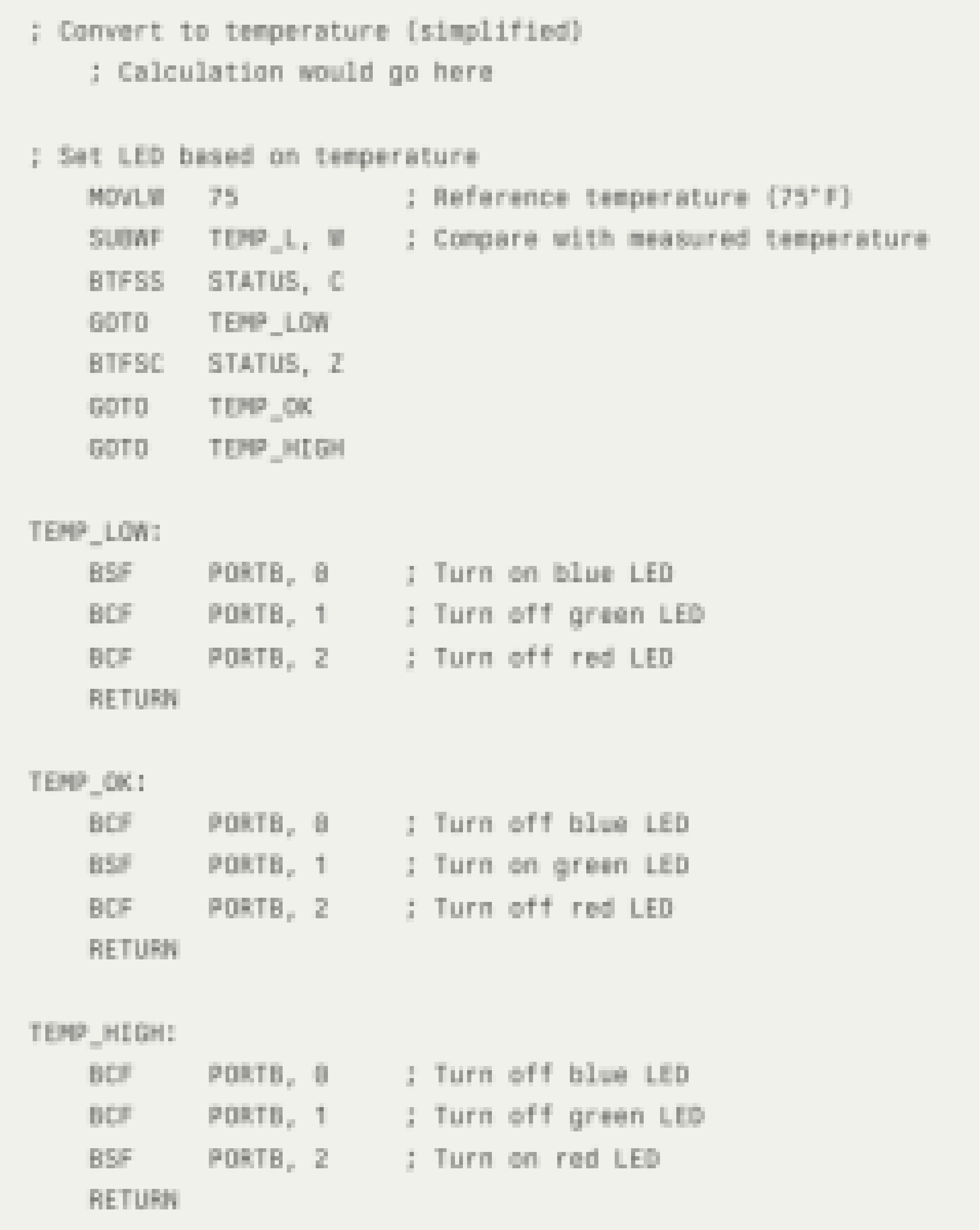
Assembly Language Considerations for ADC

1. Configure ADC control registers (ADCON0, ADCON1, etc.) appropriately.
2. Use bit-setting instructions to start conversions and check for completion.
3. Implement proper timing delays for ADC acquisition and conversion.
4. Handle 10-bit results correctly, considering result formatting (left/right justified).

Sample Code Snippet (PIC16F18875 Assembly)



```
 ; Configure ADC
    BANKSEL ADCON1
    MOVLW   b'11100000'  ; Right justified, Fosc/64, Vref+ = VDD, Vref- =
VSS
    MOVWF   ADCON1
    BANKSEL ADCON0
    MOVLW   b'00000001'  ; Channel AN0, ADC enabled
    MOVWF   ADCON0

 ; Start conversion
    BSF     ADCON0, GO   ; Start conversion
    BTFSC   ADCON0, GO   ; Wait for conversion to complete
    GOTO    $-1

 ; Read result
    BANKSEL ADRESH
    MOVF    ADRESH, W    ; Read high byte
    MOVWF   TEMP_H
    BANKSEL ADRESL
    MOVF    ADRESL, W    ; Read low byte
    MOVWF   TEMP_L
```



```
; Convert to temperature (simplified)
    ; Calculation would go here

; Set LED based on temperature
    MOVLW   75              ; Reference temperature (75°F)
    SUBWF   TEMP_L, W       ; Compare with measured temperature
    BTFSS   STATUS, C
    GOTO    TEMP_LOW
    BTFSC   STATUS, Z
    GOTO    TEMP_OK
    GOTO    TEMP_HIGH

TEMP_LOW:
    BSF     PORTB, 0        ; Turn on blue LED
    BCF     PORTB, 1        ; Turn off green LED
    BCF     PORTB, 2        ; Turn off red LED
    RETURN

TEMP_OK:
    BCF     PORTB, 0        ; Turn off blue LED
    BSF     PORTB, 1        ; Turn on green LED
    BCF     PORTB, 2        ; Turn off red LED
    RETURN

TEMP_HIGH:
    BCF     PORTB, 0        ; Turn off blue LED
    BCF     PORTB, 1        ; Turn off green LED
    BSF     PORTB, 2        ; Turn on red LED
    RETURN
```

Deepen your understanding: Watch the accompanying lecture video to delve deeper into the concepts covered in the reading.

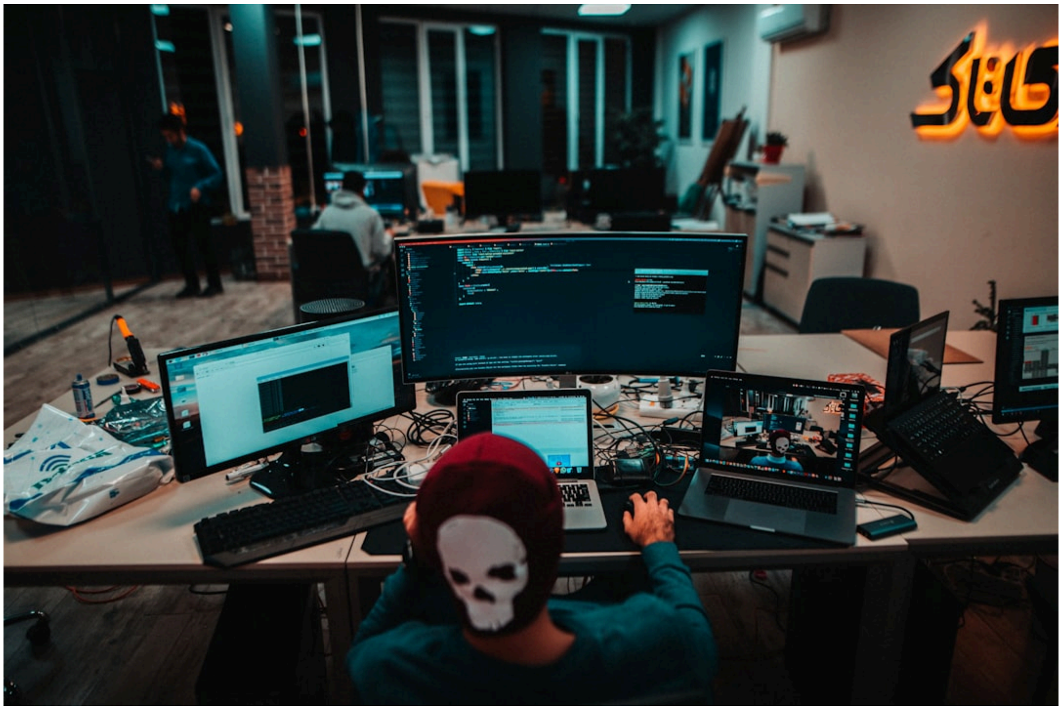
One or more interactive elements has been excluded from this version of the text. You can view them online here: https://openwa.pressbooks.pub/nehakardam10/?p=164#oembed-1

LAB #8 : MAKE THREE PROGRAM USING EXTERNAL CIRCUIT

In this Lab, you will be creating your own code for the 3 blinking LEDs program Download 3 blinking LEDs program, follow the instructions for this lab as mentioned below:

1. Perform all three programs using the separate circuit for LED. DO NOT use the built-in LED from the development board.
2. Write a report stating the whole process, the difficulty, and tips you found in that process.
3. Include the schematic of your circuit in the report.
4. Attach the report, asm file of your code for me to test, and a demo video of the running code (optional)

LAB #9: HOME SECURITY SYSTEM (ALARM SYSTEM)



Objective: Design and implement a microcontroller-based development board for a home security system using the PIC16F18875 microcontroller. The system should meet the specified performance requirements and be thoroughly documented.

Specifications

1. **System Scope:**
    - The system is designed to secure an average home.
    - It must monitor a minimum of 5 openings (doors or windows), with at least one door and one window.
2. **Code Input:**
    - The system will be enabled and disabled by inputting a 3-digit code sequentially.

3. **Arming Delay:**
   - A 30-second delay upon arming the system at a door opening to allow the occupant to exit without triggering the alarm.
4. **Disarming Delay:**
   - A 30-second delay upon entering the premises at a door opening to allow the occupant to enter and disarm the system without triggering the alarm.
5. **Alarm Outputs:**
   - Two distinct outputs:
     - A signal to activate an external siren or bell.
     - A second signal, delayed by 15 seconds, to notify a local monitoring station.
   - The external siren signal will have a silent feature.
6. **Opening Indication:**
   - The system will indicate which opening was entered when enabled and retain this information until reset.
7. **Inputs and Outputs:**
   - Inputs: Sensor outputs (5), code input, silent feature, and reset signal.
   - Outputs: Two system outputs for the siren and monitoring station, indicated by LEDs.
8. **Board Design:**
   - Ensure the development board has enough pins for all connections.

Requirements

1. **Assembly Language Program:**
   - Write the program in Assembly language with appropriate comments.
   - Include subroutines where possible to adhere to standard programming techniques.

2. **Building and Debugging:**
   - Build the program and produce the .asm file for review.
   - Download the .HEX file.
   - Debug the program thoroughly and include portions of the debug process in your report.
3. **Demonstration:**
   - Demonstrate the operational program to the instructor.
   - Post a demo video on Canvas.
4. **Technical Report:**
   - Prepare a fully documented technical report, including:
     - **Introduction:** Overview of the project and objectives.
     - **Discussion of Results:** Detailed explanation of the design process, including equations, graphs, schematics, oscilloscope pictures, and other relevant components.
     - **Understandability:** Ensure the report is understandable to another engineer or supervisor not working on the project.
     - **Conclusion:** Summarize the results, discuss interesting findings or unexpected outcomes.

Detailed Steps

1. **Circuit Design:**
   - Design the circuit to monitor 5 openings with sensors connected to the microcontroller.
   - Include input mechanisms for the 3-digit code, silent feature, and reset signal.
   - Design output connections for the siren and monitoring station, indicated by LEDs.
2. **Coding:**
   - Initialize the microcontroller's I/O ports and configure the ADC for sensor inputs.
   - Write subroutines for reading the sensors, inputting the code, and managing delays.
   - Implement logic for arming/disarming the system, triggering alarms, and indicating which opening was entered.
3. **Testing and Debugging:**

- Test the circuit and code to ensure all specifications are met.
- Debug any issues and document the debugging process.

4. **Documentation:**

    - Create a detailed schematic diagram of the circuit.
    - Include high-quality pictures of the assembled circuit board.
    - Write and comment the assembly code.
    - Record a video demonstrating the working system.

Submission

- **Schematic Diagram:** Upload a PDF or image file of the schematic diagram.
- **Circuit Board Pictures:** Upload image files of the circuit board.
- **Code:** Include the commented assembly code in a text file or directly within the submission platform.
- **Video:** Upload the video file demonstrating the working security system.
- **ASM and HEX Files:** Upload the .asm and .HEX files containing the code.

Grading Criteria

1. **Report Quality:**

    - Clarity, spelling, grammar, and organization of the report.
    - The report should be understandable to another engineer or supervisor not involved in the project.

2. **System Functionality:**

    - Whether the system works according to the specifications.
    - How well the system was designed and implemented.

3. **Programming Standards:**

    - Adherence to standard programming techniques, including the use of subroutines.

LAB #10 : TEMPERATURE INDICATOR

Objective: Using the development board and a PIC16F18875 microcontroller, build and program a Temperature Indicator that measures temperature using the MCP9701 sensor and indicates the status with LEDs based on specific temperature thresholds.

Performance Specifications

1. **Temperature Measurement:**
    - Utilize the MCP9701 sensor to measure temperature.
    - Use the PIC16F18875's Analog-to-Digital Converter (ADC) to read the temperature data.
2. **Temperature Comparison:**
    - Compare the measured temperature with a reference value of 75°F (23.89°C).
3. **LED Indicators:**
    - Use three LEDs to indicate the temperature status:
        - **Red LED:** Lights up if the temperature is higher than 75°F.
        - **Blue LED:** Lights up if the temperature is lower than 75°F.
        - **Green LED:** Lights up if the temperature is exactly 75°F.
    - Note: The LEDs can be the same color if necessary.

Requirements

1. **Schematic Diagram:**
    - Provide a detailed schematic diagram of the circuit, including the connections between the PIC16F18875 microcontroller, MCP9701 sensor, and the LEDs.
    - Ensure all components are labeled clearly.
2. **Circuit Board Pictures:**

- Include high-quality pictures of the assembled circuit board.
- Ensure the pictures clearly show the connections and components.

3. **Code:**

    - Write the assembly code for the Temperature Indicator system.
    - Include comments in the code to explain the functionality of each section.
    - Ensure the code is well-organized and easy to read.

4. **Video Demonstration:**

    - Record a full video demonstrating the working Temperature Indicator.
    - The video should show the system measuring temperature and the LEDs indicating the status correctly.
    - Ensure the video is clear and all steps are visible.

5. **ASM File:**

    - Upload the assembly (.asm) file containing the code for the Temperature Indicator.

Detailed Steps

1. **Circuit Design:**

    - Design the circuit using the MCP9701 sensor connected to the ADC input of the PIC16F18875.
    - Connect the LEDs to the appropriate output pins of the microcontroller.
    - Ensure proper power supply and grounding for all components.

2. **Coding:**

    - Initialize the ADC module on the PIC16F18875.
    - Read the analog voltage from the MCP9701 sensor and convert it to a temperature value.
    - Compare the temperature value with the reference value of 75°F.
    - Control the LEDs based on the comparison result.

3. **Testing and Debugging:**

  - Test the circuit and code to ensure accurate temperature measurement and LED indication.
  - Debug any issues that arise during testing.

4. **Documentation:**

  - Create a detailed schematic diagram of the circuit.
  - Take clear pictures of the assembled circuit board.
  - Write and comment the assembly code.
  - Record a video demonstrating the working system.

Submission

- **Schematic Diagram:** Upload a PDF or image file of the schematic diagram.
- **Circuit Board Pictures:** Upload image files of the circuit board.
- **Code:** Include the commented assembly code in a text file or directly within the submission platform.
- **Video:** Upload the video file demonstrating the working Temperature Indicator.
- **ASM File:** Upload the .asm file containing the code.

External System Analysis

The assignment has been reformulated for clarity and detail, ensuring all steps and requirements are explicitly addressed. The enhancement includes specific tasks for using the MCP9701 sensor, coding, and documentation components related to the Temperature Indicator project.

This appendix provides brief information to support the main content of the module in this course.It includes summaries and key points regarding the 8085 instruction set and the PIC16(L)F18855/75 ADC2 module, along with practical code examples and configuration details.

8085 Instruction Set Overview

The 8085 microprocessor has a rich set of instructions that can be broadly categorized into several types:

1. **Data Transfer Instructions:**
   - **MOV:** Copy data between registers or between a register and memory.
   - **MVI:** Move immediate data to a register or memory.
   - **LDA/STA:** Load or store the accumulator from/to memory.
   - **LDAX/STAX:** Load or store the accumulator using indirect addressing.
2. **Arithmetic Instructions:**
   - **ADD/ADC:** Add register or memory to the accumulator, with or without carry.
   - **SUB/SBB:** Subtract register or memory from the accumulator, with or without borrow.
   - **INR/DCR:** Increment or decrement a register or memory.
   - **DAD:** Add register pair to HL pair.
3. **Logical Instructions:**
   - **ANA/ORA/XRA:** AND, OR, or XOR register or memory with the accumulator.
   - **CMA:** Complement the accumulator.
   - **CMP:** Compare register or memory with the accumulator.
4. **Branch Instructions:**
   - **JMP:** Unconditional jump to a specified address.
   - **JC/JNC, JZ/JNZ:** Conditional jumps based on the status flags.
   - **CALL/RET:** Call and return from subroutine.
   - **RST:** Restart from a specific address.

5. **Stack, I/O, and Machine Control Instructions:**
    - **PUSH/POP:** Push or pop data to/from the stack.
    - **IN/OUT:** Input or output data from/to an I/O port.
    - **NOP:** No operation.
    - **HLT:** Halt the processor.

PIC16(L)F18855/75 ADC2 Module Configuration

The ADC2 module in the PIC16(L)F18855/75 microcontroller allows for the conversion of analog input signals to 10-bit binary representations. Key configuration steps include:

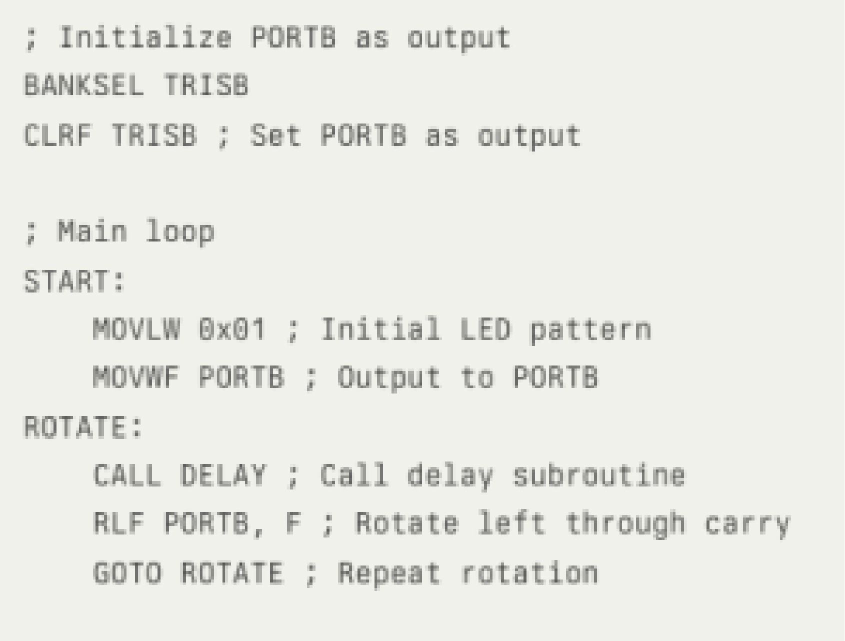
1. **Port Configuration:**
    - Set the I/O pin as an input by configuring the TRIS register.
    - Configure the pin as an analog input by setting the ANSEL register.
2. **Channel Selection:**
    - Use the ADPCH register to select the desired analog input channel.
3. **Voltage Reference Selection:**
    - Configure the ADREF register to select the positive and negative voltage references.
4. **Conversion Clock Source:**
    - Select the ADC conversion clock source via the ADCLK register and the ADCS bit of the ADCON0 register.
5. **Interrupt Control:**
    - Enable ADC interrupts by setting the ADIE bit in the PIE1 register and the PEIE and GIE bits in the INTCON register.
6. **Result Formatting:**

- Choose between left-justified and right-justified formats for the 10-bit conversion result using the ADFRM0 bit of the ADCON0 register.

7. **Starting and Completing a Conversion:**

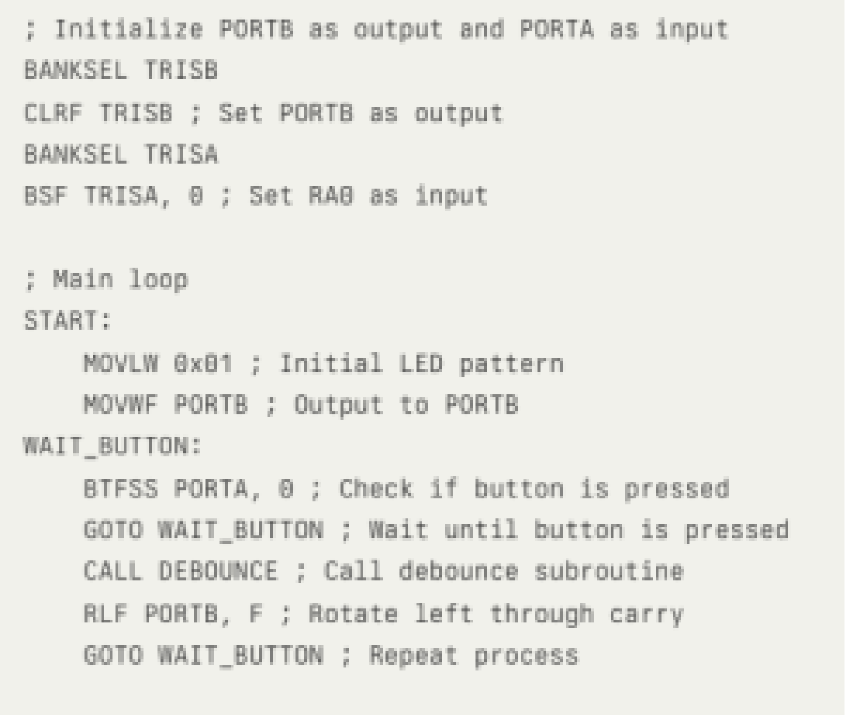
    - Enable the ADC module by setting the ADON bit of the ADCON0 register.
    - Start a conversion by setting the ADGO bit.
    - Upon completion, the ADGO bit is cleared, and the conversion result is stored in the ADRESH:ADRESL registers.

LED Control Example



```
; Initialize PORTB as output
BANKSEL TRISB
CLRF TRISB ; Set PORTB as output

; Main loop
START:
    MOVLW 0x01 ; Initial LED pattern
    MOVWF PORTB ; Output to PORTB
ROTATE:
    CALL DELAY ; Call delay subroutine
    RLF PORTB, F ; Rotate left through carry
    GOTO ROTATE ; Repeat rotation
```

Button Press Example



```
; Initialize PORTB as output and PORTA as input
BANKSEL TRISB
CLRF TRISB ; Set PORTB as output
BANKSEL TRISA
BSF TRISA, 0 ; Set RA0 as input

; Main loop
START:
    MOVLW 0x01 ; Initial LED pattern
    MOVWF PORTB ; Output to PORTB
WAIT_BUTTON:
    BTFSS PORTA, 0 ; Check if button is pressed
    GOTO WAIT_BUTTON ; Wait until button is pressed
    CALL DEBOUNCE ; Call debounce subroutine
    RLF PORTB, F ; Rotate left through carry
    GOTO WAIT_BUTTON ; Repeat process
```