Embedded_intro_sessions

Embedded Systems Introduction Sessions

Created: 2025-07-13 & 2025-07-14
Author: Fares Hesham Mahmoud
Tags: AVR, Embedded system, Introduction
Status: #Formatted

اللهمَّ علِّمنا ما ينفعنا، وانفعنا بما علمتنا، وزدنا علمًا. وافتح علينا فتحًا عظيمًا.

Embedded Systems Overview

Embedded Systems: Building specialized blocks that control actions and processes within devices.

Core Components

1. Processor - Central processing unit
2. I/O Interfaces - Input/output connections
3. Memory - Data and program storage

Design Requirements

Requirement	Description	Trade-offs
Cost	Manufacturing and component costs	Performance vs. budget
Size	Physical dimensions and weight	Functionality vs. compactness
Performance	Processing speed and capability	Power vs. speed
Power Consumption	Energy efficiency	Performance vs. battery life
Configurability	Flexibility and adaptability	Simplicity vs. versatility

Processor Architecture

Terminology Clarification

Term	Definition	Context
Processor	Generic term for processing unit	Started with vacuum tube processors
Microprocessor	Modern integrated processor	Current standard terminology
CPU	Complete chip with processor + support	Processor + additional tools

Processor Internal Components

Control Unit

- Decoder: Translates instructions
- Fetch Unit: Retrieves instructions from memory

ALU (Arithmetic Logic Unit)

Performs mathematical and logical operations.

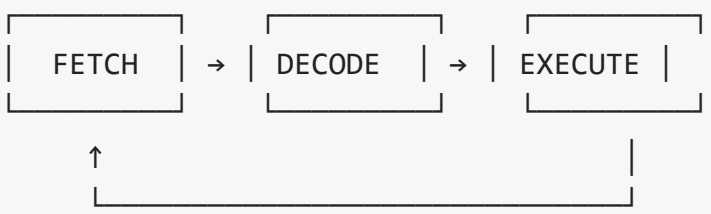
Register Bank/Register File

Register	Purpose	Example Content
Program Counter (PC)	Address of next instruction	0x1A2B
Instruction Register (IR)	Current instruction being executed	ADD R1, R2
Accumulator	Stores ALU results	0x45
PSW (Processor Status Word)	Status flags	Z=1, C=0, N=0
GPR (General Purpose Register)	Temporary storage for frequently used data	Variables, constants

Status Flags (PSW Examples)

Flag	Name	Condition
Z	Zero Flag	Result equals zero
C	Carry Flag	Arithmetic overflow/carry
N	Negative Flag	Result is negative
O	Overflow Flag	Signed arithmetic overflow

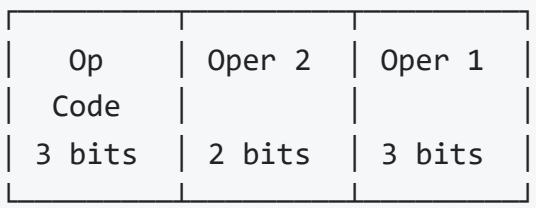
Processor Execution Cycle



1. FETCH: Locate and read instruction from memory
2. DECODE: Translate and understand the instruction
3. EXECUTE: Perform the instruction operation

Instruction Set Architecture

Instruction Format Example (8-bit)




```
Example: 001 10 011
         ADD  2   3  → Add register 2 to register 3
```

Instruction Set Examples

Operation	Op Code	Description
ADD	001	Addition
SUB	010	Subtraction
SHL	011	Shift Left
SHR	100	Shift Right

RISC vs CISC Architecture

RISC (Reduced Instruction Set Computing)

Characteristics

- **Small instruction set** (limited operations)
- **Hardware implementation** preferred for performance
- **Complex compiler** required to optimize code

Example: Multiplication in RISC

```
// No direct multiply instruction
// Multiplication of 2 × 3 implemented as:
result = 2 + 2 + 2;  // Three additions
```

CISC (Complex Instruction Set Computing)

Characteristics

- **Large instruction set** (many operations)
- **Software implementation** preferred for cost
- **Simpler compiler** due to rich instruction set

Example: Multiplication in CISC

```
MUL R1, R2  ; Direct multiplication instruction
```

RISC vs CISC Comparison

Aspect	RISC	CISC
Instruction Count	~100 instructions	~1000+ instructions
Implementation	Hardware preferred	Software preferred
Performance	Higher (specialized)	Lower (general purpose)
Cost	Higher hardware cost	Lower hardware cost
Compiler Complexity	High	Medium
Power Consumption	Lower	Higher

Key Design Principle >

> "The slowest hardware is faster than the fastest software!"
> This drives the decision of whether to implement functionality in hardware (RISC) or software (CISC).

Memory Technologies

ROM (Read-Only Memory) Types

Type	Programming	Reprogramming	Erasure Method	Cost	Use Case
Masked ROM (MROM)	Factory	No	N/A	Lowest	High-volu productio
OTP/PROM	User (once)	No	N/A	Low	Small runs developm
EPROM	User	Yes	UV light	Moderate	Prototypir developm
EEPROM	User	Yes	Electrical	High	Configura data, logs
Flash Memory	User	Yes	Electrical	Moderate	Mass stor firmware

Memory Characteristics Detail

Masked ROM (MROM)

- Programming: Done by manufacturer during fabrication
- Reprogramming: Not possible (permanent)
- Cost: Lowest per bit for high volumes
- Applications: Mass-produced firmware, boot code

OTP/PROM (One-Time Programmable)

- Programming: User programmable, one time only
- Reprogramming: Not possible after programming
- Cost: Low for moderate volumes
- Applications: Small production runs, final firmware versions

EPROM (Erasable Programmable ROM)

- **Programming:** Electrical programming via programmer
- **Reprogramming:** UV light erases entire chip (20-40 minutes)
- **Cycles:** Few thousand erase/write cycles
- **Applications:** Development, prototyping (now largely obsolete)

EEPROM (Electrically Erasable PROM)

- **Programming:** In-circuit electrical programming
- **Reprogramming:** Byte-level electrical erasure
- **Cycles:** 100k to 1M+ erase/write cycles
- **Applications:** Configuration parameters, calibration data

Flash Memory

- **Programming:** In-circuit electrical programming
- **Reprogramming:** Block/page-level electrical erasure
- **Cycles:** 10k to 100k+ erase/write cycles
- **Applications:** Program storage, data logging, USB drives, SSDs

Bus Architecture

Bus Types

Bus Type	Function	Example
Address Bus	Carries memory/peripheral addresses	PC content
Data Bus	Carries actual data	Instructions, variables
Control Bus	Carries control signals	Read/Write, Enable

Bus Width Impact

Address Bus Width

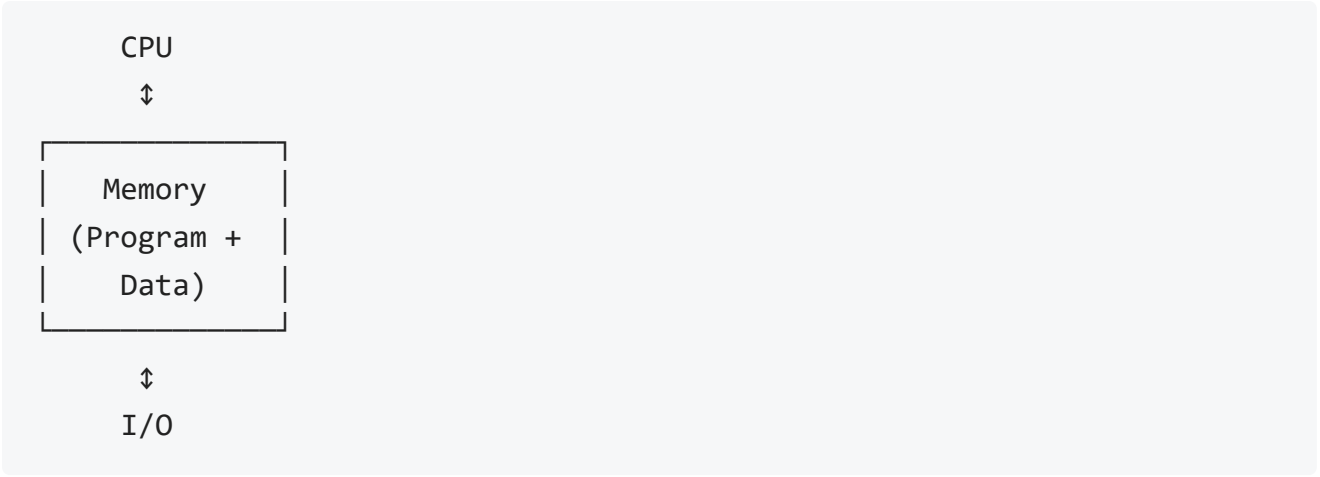
- **8-bit:** Can address 2^8 = 256 locations
- **16-bit:** Can address 2^{16} = 64K locations
- **32-bit:** Can address 2^{32} = 4GB locations

Data Bus Width

- **8-bit:** Transfers 1 byte at a time
- **16-bit:** Transfers 2 bytes at a time
- **32-bit:** Transfers 4 bytes at a time

Architecture Comparison

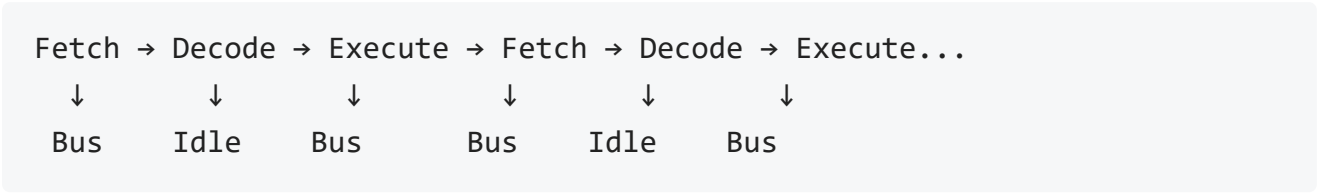
Von Neumann Architecture



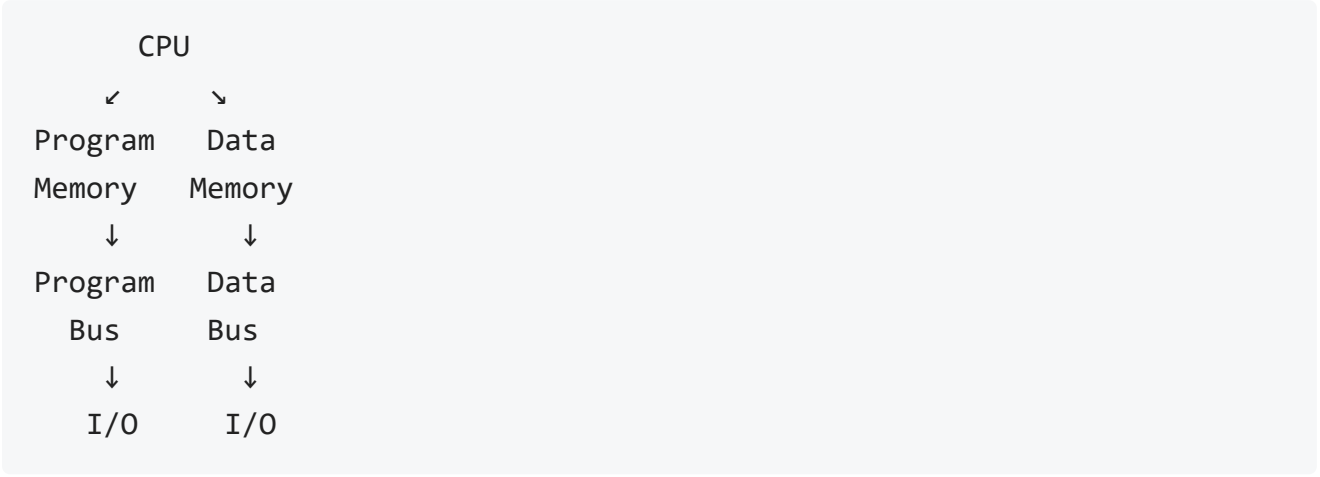
Characteristics

- **Single bus system** for program and data
- **Sequential operation:** Only one operation at a time
- **Bottleneck:** Bus shared between instruction fetch and data access
- **Simpler design:** Unified memory space

Execution Sequence



Harvard Architecture



Characteristics

- **Separate buses** for program and data

- **Parallel operations:** Can fetch and execute simultaneously
- **Higher performance:** No bus contention
- **More complex:** Requires separate memory spaces

Execution Sequence

```
Fetch₁ → Decode₁ → Execute₁
  ↓        ↓         ↓
Fetch₂ → Decode₂ → Execute₂ (Pipeline)
```

Architecture Comparison

Feature	Von Neumann	Harvard
Buses	1 (shared)	2+ (separate)
Performance	Lower	Higher
Complexity	Simple	Complex
Cost	Lower	Higher
Pipelining	Limited	Excellent
Memory Usage	Flexible	Fixed allocation

ATmega32 Microcontroller

Pin Configuration (40-pin DIP)

General Purpose I/O Pins (32 pins)

Organized in 4 groups of 8 pins each:

Port	Pins	Pin Numbers
Port A	PA0-PA7	33-40
Port B	PB0-PB7	1-8
Port C	PC0-PC7	21-29
Port D	PD0-PD7	14-20

Pin Control Registers

Each port has three associated registers:

Register	Function	Description
DDR	Data Direction Register	0=Input, 1=Output
PORT	Port Output Register	Output value when pin is output
PIN	Port Input Register	Read current pin state

Port Register Example

```
// Configure Port B
DDRB = 0xFF;    // All pins as output
PORTB = 0xAA;   // Set alternating pattern (10101010)

// Configure Port C
DDRC = 0x00;    // All pins as input
PORTC = 0xFF;   // Enable internal pull-ups
uint8_t input = PINC;  // Read input values
```

Specific Function Pins (8 pins)

Pin	Function	Description
VCC	Power Supply	Positive power (+5V)
GND	Ground	Negative power (0V)
AVCC	Analog VCC	Clean power for ADC
AREF	Analog Reference	ADC reference voltage
XTAL1	Crystal Input	External oscillator input
XTAL2	Crystal Output	External oscillator output
RESET	Reset Input	Active low reset signal

Clock System

Clock Sources

- Internal RC Oscillator: Built-in, less accurate
- External Crystal: High accuracy, stable frequency
- External Clock: From external source

Clock Timing Example

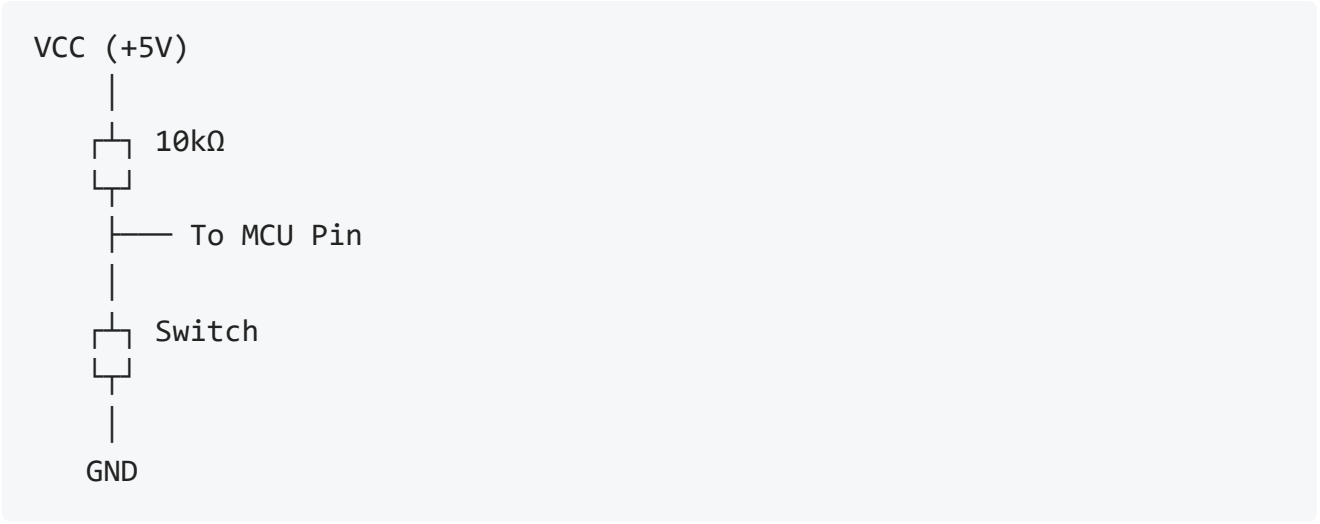
```
Frequency = 1 MHz
Period = 1/1MHz = 1 µs per cycle

Processor Operations:
- Fetch: Rising edge
- Decode: High level
- Execute: Falling edge
```

Hardware Interfacing Concepts

Pull-up and Pull-down Resistors

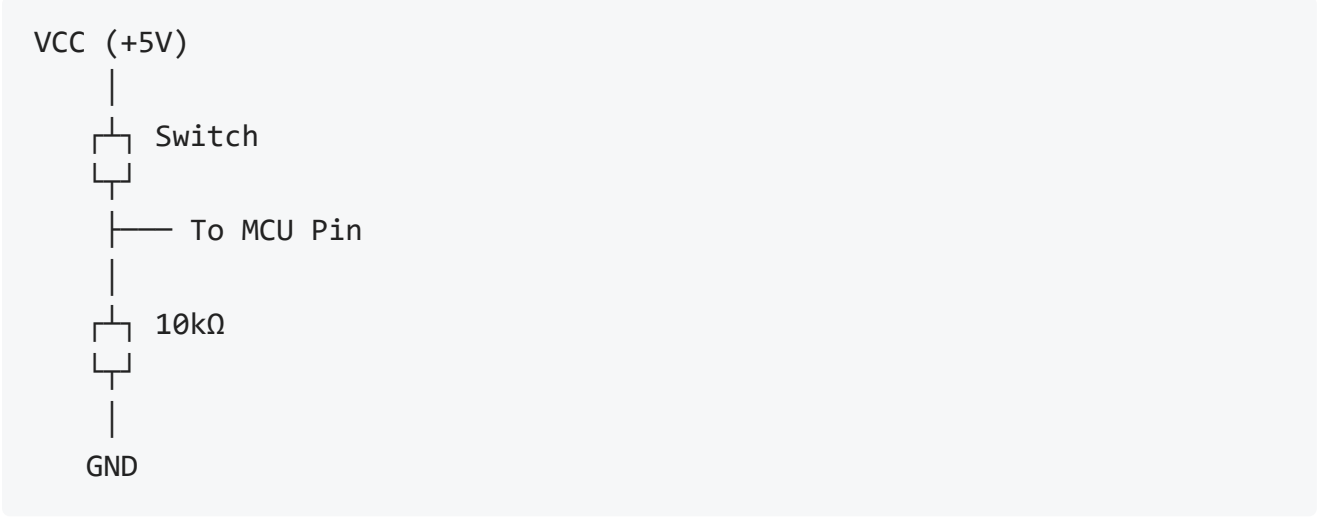
Pull-up Resistor Configuration



Operation:

- Switch Open: Pin reads HIGH (1)
- Switch Closed: Pin reads LOW (0)

Pull-down Resistor Configuration



Operation:

- **Switch Open:** Pin reads LOW (0)
- **Switch Closed:** Pin reads HIGH (1)

Internal Pull-up in ATmega32

```
// Enable internal pull-up on input pin
DDRD &= ~(1<<PD2);   // Set PD2 as input
PORTD |= (1<<PD2);   // Enable internal pull-up
```

Memory-Mapped I/O

Register Access Methods

Direct Memory Access (Not Recommended)

```
// Magic number - unclear purpose
*((volatile uint8_t*)0x37) = 0x01;
```

Defined Register Names (Recommended)

```
// Clear, readable code
#define DDRB_REG  *((volatile uint8_t*)0x37)
DDRB_REG = 0x01;  // Configure Port B direction
```

> ⚠ **Magic Numbers** >
>
> **Magic Numbers** are unrecognized numeric values with unclear meaning in code.
> **Always use named constants or** `#define` **macros** for better code readability and maintainability.

Volatile Keyword Importance

```
// Without volatile - compiler may optimize away
uint8_t *register_ptr = (uint8_t*)0x37;

// With volatile - ensures actual hardware access
volatile uint8_t *register_ptr = (volatile uint8_t*)0x37;
```

Why volatile is needed:

- Prevents compiler optimization
- Ensures actual hardware register access
- Required for memory-mapped I/O operations

Practical Examples

LED Control Example

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

#define LED_PORT PORTB
#define LED_DDR  DDRB
#define LED_PIN  PB0

int main(void) {
    // Configure LED pin as output
    LED_DDR |= (1 << LED_PIN);

    while(1) {
        // Turn LED on
        LED_PORT |= (1 << LED_PIN);
        _delay_ms(500);

        // Turn LED off
        LED_PORT &= ~(1 << LED_PIN);
        _delay_ms(500);
    }

    return 0;
}
```

Button Reading Example

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

#define BUTTON_PORT PORTD
#define BUTTON_DDR  DDRD
#define BUTTON_PIN  PIND
#define BUTTON_BIT  PD2
```

```c
int main(void) {
    // Configure button pin as input with pull-up
    BUTTON_DDR &= ~(1 << BUTTON_BIT);  // Input
    BUTTON_PORT |= (1 << BUTTON_BIT);   // Pull-up

    // Configure LED as output
    DDRB |= (1 << PB0);

    while(1) {
        // Read button (active low due to pull-up)
        if (!(BUTTON_PIN & (1 << BUTTON_BIT))) {
            PORTB |= (1 << PB0);   // LED on
        } else {
            PORTB &= ~(1 << PB0);  // LED off
        }
    }

    return 0;
}
```

Development Environment

Driver Development Approach

- **Ready-made drivers:** Available but limited learning
- **Custom drivers:** Built from scratch for better understanding
- **Header files (.h):** Contain definitions and declarations
- **Implementation files (.c):** Contain actual function implementations

Code Organization Best Practices

```c
// config.h - System configuration
#define F_CPU 8000000UL
#define LED_PIN PB0

// led.h - LED driver interface
void LED_Init(void);
void LED_On(void);
void LED_Off(void);
void LED_Toggle(void);

// led.c - LED driver implementation
#include "led.h"
```

```
void LED_Init(void) {
    DDRB |= (1 << LED_PIN);
}
```

References

- ATmega32 Datasheet
- AVR Instruction Set Manual
- Computer Architecture Fundamentals
- Embedded Systems Design Principles