# Computer Architecture and Organization

3rd Semester Software Technology Engineering

Eduard Fekete

December 8, 2025

# Contents

# 1 Introduction

Computer Architecture and Organization (CAO) bridges the gap between high-level software and the physical hardware that executes it.

**Computer Architecture** refers to the attributes of a system visible to the programmer. It defines the **Instruction Set Architecture (ISA)**:

- **Instruction Set**: The vocabulary of the machine (e.g., ADD, SUB, JMP).
- **Data Types**: How integers, characters, and floats are represented.
- **Addressing Modes**: Mechanisms to access memory operands.
- **Registers**: The state visible to the programmer.

**Computer Organization** (or Microarchitecture) refers to the operational units and their interconnections that realize the architectural specifications.

- **Control Signals**: How the hardware is told what to do.
- **Interfaces**: How memory and I/O connect to the CPU.
- **Memory Technology**: SRAM vs DRAM vs Flash.

Here, we focus on the **Harvard Architecture** (separate memory for code and data) as implemented by the **Atmel AVR** family, specifically the **ATmega2560**.

## 1.1 Microcontroller vs Microprocessor

It is important to distinguish between the two:

- **Microprocessor (MPU)**: A CPU core only (e.g., Intel Core i7). Requires external RAM, Flash, and I/O chips on a motherboard to function. Optimized for raw processing power.
- **Microcontroller (MCU)**: A "System on a Chip". Contains the CPU, RAM, Flash, and Peripherals (Timers, ADC) all on a single silicon die. Optimized for control, low power, and cost. The ATmega2560 is an MCU.

## 1.2 System Bus Architecture

Data moves between components via buses:

- **Data Bus**: Carries the actual information (8-bit wide in AVR).
- **Address Bus**: Specifies the location of the data (16-bit wide).
- **Control Bus**: Carries signals like Read, Write, and Clock.

# 2 Arduino Atmega2560

The ATmega2560 is an 8-bit RISC (Reduced Instruction Set Computer) microcontroller. Unlike general-purpose CPUs (like x86 in PCs) which use a Von Neumann architecture (unified memory), the AVR uses a **Modified Harvard Architecture**.

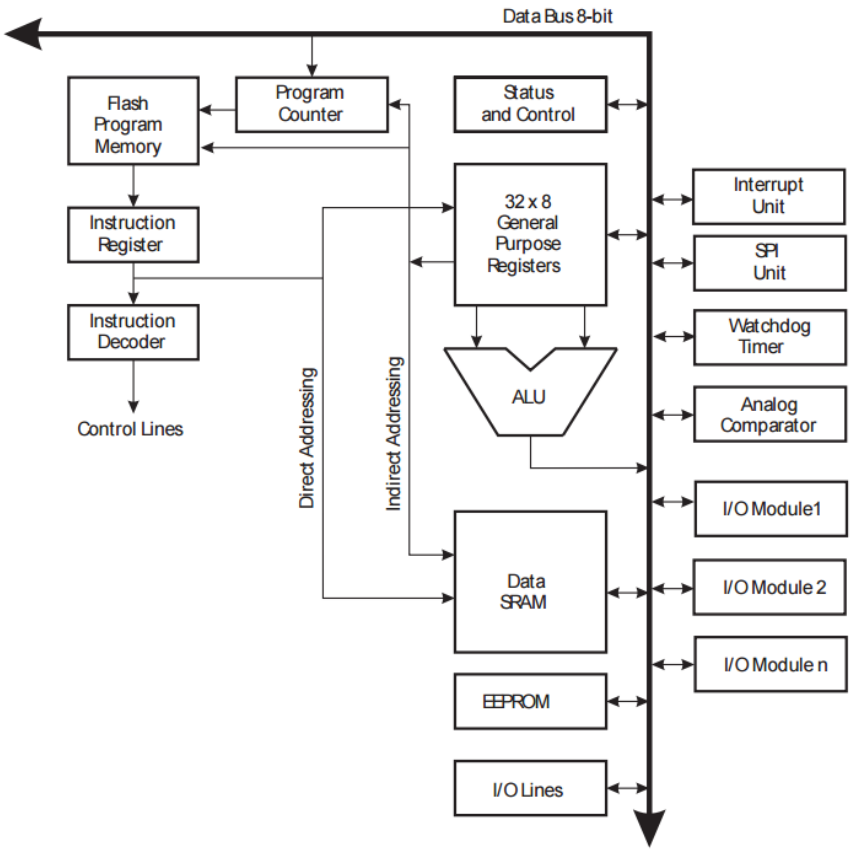


Figure 1: The AVR CPU Core

## 2.1 Core Components

| Part | Description | Functionality |
|---|---|---|
| **ALU** | Arithmetic Logic Unit | The computational brain. It performs arithmetic (add, sub) and logic (and, or, xor) operations. It operates directly on the Register File. |
| **CU** | Control Unit | The conductor. It decodes instructions fetched from Flash and generates control signals to coordinate the ALU, Registers, and Memory. |
| **Registers** | General Purpose Registers | The fastest storage in the CPU (access time: 1 clock cycle). Used to hold temporary data and operands for the ALU. |
| **SRAM** | Static Random Access Memory | Volatile data memory. Stores variables, the stack, and the heap. Fast access but loses data when power is lost. |
| **FLASH** | Program Memory | Non-volatile memory. Stores the program code (instructions) and constant data. Can be read by the CPU but not easily written to during runtime. |
| **EEPROM** | Electrically Erasable PROM | Non-volatile data memory. Used for storing persistent configuration data (like settings) that must survive power cycles. Slower than SRAM. |
| **I/O Ports** | Input/Output Ports | The interface to the outside world. Digital pins that can be read (Input) or written (Output) to control LEDs, motors, or read sensors. |

## 2.2 On-Chip Peripherals vs CPU Core

While the **CPU Core** executes instructions, the **Peripherals** do the heavy lifting of interfacing with the world. They are memory-mapped, meaning the CPU controls them by writing to specific addresses. - **Inside CPU**: ALU, Register File, Program Counter, Stack Pointer, Status Register, Instruction Decoder. - **Outside CPU (but on chip)**: SRAM, Flash, EEPROM, Timers/Counters, USART (Serial), SPI, I2C, ADC, Watchdog Timer, GPIO Ports.

## 2.3 Specifications (ATmega2560)

- **Architecture**: 8-bit AVR RISC
- **Clock Speed**: 16 MHz (Up to 16 MIPS throughput)
- **Flash Memory**: 256 KB (Code)
- **SRAM**: 8 KB (Runtime Data)
- **EEPROM**: 4 KB (Persistent Data)
- **I/O Pins**: 86 Programmable lines
- **Timers**: 2 x 8-bit, 4 x 16-bit
- **PWM Channels**: 15
- **ADC**: 16 channels, 10-bit resolution
- **Communication**: 4 UARTs, 1 SPI, 1 I2C

## 2.4 The Harvard Advantage

By having separate buses for **Program Memory** and **Data Memory**, the CPU can fetch the next instruction from Flash *while* simultaneously reading or writing data to SRAM. This allows for **single-cycle execution** for most instructions, a hallmark of the AVR RISC design.

# 3 Registers

The AVR architecture features a **Register File** containing 32 x 8-bit General Purpose Working Registers. These are directly connected to the ALU, allowing two independent registers to be accessed in one single instruction executed in one clock cycle.

| Register | Description | Usage Notes |
|---|---|---|
| **r0 - r31** | General Purpose | Used for arithmetic and logic. `r0` is often used as a scratch register by compilers. |
| **r16 - r31** | High Registers | Only these registers can be used with immediate operations (e.g., `LDI` - Load Immediate). `r0-r15` cannot be loaded directly with a constant. |
| **r26 - r31** | Pointer Registers (X, Y, Z) | 16-bit address pointers formed by combining two 8-bit registers. Essential for addressing the 64KB+ memory space. |
| **SPH, SPL** | Stack Pointer | A 16-bit register pointing to the top of the Stack in SRAM. Used for subroutine calls and interrupts. |
| **SREG** | Status Register | Contains flags indicating the result of the most recent arithmetic operation. |
| **PC** | Program Counter | A 16-bit (or larger) counter holding the address of the *next* instruction to be executed. |

## 3.1 Special Function Registers (SFRs)

Beyond the 32 general-purpose registers, the AVR has I/O registers (SFRs) that control peripherals.

- Addresses `$00` to `$3F` are directly accessible using `IN` and `OUT` instructions.
- Examples: `PORTA`, `DDRA`, `TCNT0` (Timer Count).
- These act as the "dashboard" for the microcontroller hardware.

## 3.2 Stack Pointer Details

The Stack is a LIFO (Last In, First Out) buffer growing from high memory addresses to low.

- **Initialization**: The SP must be set to the end of SRAM (RAMEND) at the start of the program.
- **Operation**: `PUSH` decrements SP; `POP` increments SP.
- **Corruption**: If the Stack grows too large (recursion or too many interrupts), it can overwrite global variables (Stack Overflow), causing erratic behavior.

## 3.3 The Pointer Registers (X, Y, Z)

Since the ATmega2560 has more than 256 bytes of memory, a single 8-bit register cannot address all of it. The architecture pairs the last six registers to form 16-bit pointers:

- **X Pointer**: `r27` (High byte) : `r26` (Low byte)
- **Y Pointer**: `r29` (High byte) : `r28` (Low byte)
- **Z Pointer**: `r31` (High byte) : `r30` (Low byte) - *Special use: Accessing Flash memory (LPM instruction).*

## 3.4 SREG Flags (The Decision Makers)

The Status Register (SREG) is the mechanism for conditional branching. If you compare two numbers, the CPU subtracts them and updates these flags. A subsequent "Branch if Equal" (`BREQ`) instruction simply checks the Z-flag.

| Bit | Name | Description | Mathematical Meaning |
|---|---|---|---|
| **0** | **C** | Carry Flag | Indicates an unsigned overflow. Example: `255 + 1 = 0` (Carry set). |
| **1** | **Z** | Zero Flag | Set if the result is zero. Used for equality checks (`a == b`). |
| **2** | **N** | Negative Flag | Simply a copy of the Most Significant Bit (MSB, bit 7) of the result. |
| **3** | **V** | Two's Complement Overflow | Set if a signed operation overflows (e.g., `127 + 1 = -128` in signed 8-bit). |
| **4** | **S** | Sign Flag | `S = N XOR V`. The *true* sign of the number. If `S=1`, the number is effectively negative, correcting for overflow errors. |
| **5** | **H** | Half Carry Flag | Carry from bit 3 to 4. Used for BCD (Binary Coded Decimal) arithmetic. |
| **6** | **T** | Transfer Bit | A storage bit for bit-copy instructions (`BLD`, `BST`). |
| **7** | **I** | Global Interrupt Enable | The master switch for interrupts. Must be set (1) for any ISR to run. |

# 4 CAOLibrary

Here, a simplified library to abstract the complex setup of hardware peripherals (like the UART baud rate calculation) is used. This allows us to focus on assembly logic and register manipulation.

| Task you want | Use routine | Input you must prepare |
|---|---|---|
| Initialize | `cao_usart0_init` | **None**. Call this once at the start. Sets up Serial communication at 115200 baud. |
| Print Number | `cao_print_r31` | **r31**. Load the 8-bit value (0-255) you want to display into register `r31`. |
| Print SRAM String | `cao_print_data_x` | **X Pointer (r27:r26)**. Load the address of the string in SRAM into X. The string must be null-terminated. |
| Print Flash String | `cao_print_code_z` | **Z Pointer (r31:r30)**. Load the address of the string in Flash into Z. **Important**: Flash is word-addressed in hardware but byte-addressed in software tools, so you often need `address * 2`. |
| Delay | `cao_delay_r16` | **r16**. A blocking delay. The value in `r16` determines the duration (approx 8.2ms per count). |

## 4.1 Calling Convention

To ensure modular code, we follow a convention:

- **Arguments**: Passed in specific registers (e.g., `r31` for values, `X/Z` for pointers).
- **Return Values**: Usually returned in `r24` or `r25` (standard AVR-GCC convention), though this library relies on side effects (printing).

## 4.2 Register Preservation

When calling a subroutine, who is responsible for saving registers?

- **Caller-Saved**: Registers the subroutine is allowed to trash. The caller must save them if needed.
- **Callee-Saved**: Registers the subroutine must preserve. If the subroutine uses them, it must `PUSH` them at the start and `POP` them at the end.

## 4.3 Examples

### 4.3.1 Print "Hello World!" from FLASH

Storing strings in Flash saves valuable SRAM. We use the `Z` pointer and the `LPM` (Load Program Memory) capable routine.

```
    ; Load the address of the label 'hello_world' into Z pointer (r31:r30)
    ; We multiply by 2 because the assembler addresses bytes, but Flash is organized in words.
    ldi     r31, lo8(hello_world*2) ; Load Low byte of address
    ldi     r30, hi8(hello_world*2) ; Load High byte of address
    rcall   cao_print_code_z        ; Call library routine
    rjmp    main_loop               ; Infinite loop
hello_world:
    .db     "Hello World!", 0       ; Define Byte (.db) string, null terminated
```

### 4.3.2 Print "Hello World!" from SRAM

If a string is generated dynamically (e.g., user input), it lives in SRAM. We use the `X` pointer.

```
    ; Load the address of 'hello_sram' into X pointer (r27:r26)
    ldi     r26, lo8(hello_sram)    ; X Low
    ldi     r27, hi8(hello_sram)    ; X High
```

```
    rcall   cao_print_data_x        ; Call library routine
    rjmp    main_loop
hello_sram:
    .db     "Hello World!", 0       ; Note: In a real app, this needs to be copied to SRAM first
```

#### 4.3.3 Print a number from 0 to 255

Visualizing register values is crucial for debugging.

```
    ldi     r31, 123          ; Load immediate value 123 into r31
    rcall   cao_print_r31     ; Print the decimal representation "123" to UART
    rjmp    main_loop
```

#### 4.3.4 Wait some milliseconds

A simple software delay loop.

```
    ldi     r16, 10           ; Load counter. 10 * 8.2ms ~= 82ms
    rcall   cao_delay_r16     ; Block CPU for 82ms
    rjmp    main_loop
```

# 5 Representation of data

Computers only understand binary (0s and 1s). All data types are abstractions built upon this.

## 5.1 Integers

- **Unsigned Integers**: Direct binary mapping. 8-bit range: 0 to 255 ($2^8 - 1$).
- **Signed Integers (Two's Complement)**: The standard for representing negative numbers.
  - To negate a number: Invert all bits and add 1.
  - Example: `1` is `00000001`. Invert: `11111110`. Add 1: `11111111` (-1).
  - 8-bit range: -128 to +127.
  - The MSB (Most Significant Bit) acts as the sign bit (1=negative, 0=positive).

## 5.2 Floating point numbers (IEEE 754)

Representing real numbers (with fractions) requires a scientific notation format: $Value = (-1)^S \times 1.M \times 2^{E-Bias}$.

- **Single Precision (32-bit)**:
  - **Sign (1 bit)**: Positive or Negative.
  - **Exponent (8 bits)**: The magnitude. Stored with a bias of 127 (so an exponent of 0 is stored as 127).
  - **Mantissa/Fraction (23 bits)**: The precision. The leading "1." is implicit.
- **AVR Implementation**: The Atmega2560 has **no hardware FPU** (Floating Point Unit). All float operations are emulated in software (libgcc), which is very slow and consumes lots of Flash. Avoid floats in 8-bit embedded systems if possible.

## 5.3 Text

- **ASCII**: A 7-bit encoding mapping numbers 0-127 to characters (e.g., 'A' = 65, '0' = 48).
- **Strings**: In C and Assembly, a string is a contiguous array of characters terminated by a **Null Byte** (`0x00`).
  - "Hi" is stored as `0x48, 0x69, 0x00`.
  - Without the null terminator, functions like `printf` will keep reading memory until they crash or find a random zero.

## 5.4 Fixed Point Arithmetic

Since floating point is slow, embedded systems often use Fixed Point.

- **Concept**: Reserve a specific number of bits for the fractional part.
- **Example (Q4.4)**: 8 bits total. 4 bits integer, 4 bits fraction.
  - `0001.1000` $= 1 + 0.5 = 1.5$.
  - Math is done using standard integer instructions (ADD, SUB), which is extremely fast.

## 5.5 Endianness

Describes the order of bytes in multi-byte words.

- **Little Endian**: Least Significant Byte (LSB) stored at the lowest address. (Used by **AVR** and x86).
  - Value `0x1234` at address `0x100`: `0x100 = 0x34`, `0x101 = 0x12`.
- **Big Endian**: Most Significant Byte (MSB) stored at the lowest address. (Used by Network protocols).

# 6 Boolean Logic and Combinational Logic

Digital logic is the foundation of the CPU.

## 6.1 Boolean Algebra & Gates

- **Basic Gates**: AND (Intersection), OR (Union), NOT (Inversion).
- **Universal Gates**: NAND and NOR. Any logic function can be built using only NAND or only NOR gates.
- **XOR (Exclusive OR)**: True if inputs are different. Crucial for adders and parity checks.
- **De Morgan's Laws**:
  - $\overline{A \cdot B} = \overline{A} + \overline{B}$ (NAND is equivalent to OR with inverted inputs).
  - $\overline{A + B} = \overline{A} \cdot \overline{B}$ (NOR is equivalent to AND with inverted inputs).

## 6.2 Logic Gates Reference

| Gate | Symbol | Function | Truth Table (A, B -> Q) |
|---|---|---|---|
| **AND** | $A \cdot B$ | True only if ALL inputs are True. | 0,0->0; 0,1->0; 1,0->0; 1,1->1 |
| **OR** | $A + B$ | True if ANY input is True. | 0,0->0; 0,1->1; 1,0->1; 1,1->1 |
| **NOT** | $\overline{A}$ | Inverts the input. | 0->1; 1->0 |
| **NAND** | $\overline{A \cdot B}$ | False only if ALL inputs are True. | 0,0->1; 0,1->1; 1,0->1; 1,1->0 |
| **NOR** | $\overline{A + B}$ | True only if ALL inputs are False. | 0,0->1; 0,1->0; 1,0->0; 1,1->0 |
| **XOR** | $A \oplus B$ | True if inputs are DIFFERENT. | 0,0->0; 0,1->1; 1,0->1; 1,1->0 |

## 6.3 Boolean Identities

Simplifying logic saves transistors.

- **Identity**: $A \cdot 1 = A$, $A + 0 = A$
- **Null**: $A \cdot 0 = 0$, $A + 1 = 1$
- **Idempotent**: $A \cdot A = A$, $A + A = A$
- **Inverse**: $A \cdot \overline{A} = 0$, $A + \overline{A} = 1$
- **Distributive**: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

## 6.4 Combinational Circuits

Outputs depend *only* on the current inputs. There is no memory of the past.

- **Multiplexer (MUX)**: A digital switch. Selects one of $N$ inputs to forward to the output based on control lines. Used inside the CPU to select which register feeds the ALU.
- **Decoder**: Converts binary code to a "one-hot" output. Example: Address decoding (selecting a specific RAM chip based on address bits).
- **Adder**:
  - **Half Adder**: Adds 2 bits. Outputs Sum and Carry.
  - **Full Adder**: Adds 3 bits (A, B, Carry-in). Outputs Sum and Carry-out.
  - **Ripple Carry Adder**: Chained Full Adders. Slow because carry must propagate through the chain.

## 6.5 Minimization

- **Karnaugh Maps (K-Map)**: A graphical method to simplify boolean expressions. By grouping adjacent 1s in a grid (ordered by Gray code), we find the minimal logic gates required.
- **Timing**: Real gates have **Propagation Delay**. The **Critical Path** is the longest path from input to output and determines the maximum clock speed of the CPU.

# 7 Sequential Logic

Outputs depend on current inputs **AND** the current state (memory). This allows the system to perform sequences of operations.

## 7.1 Memory Elements

- **Latch**: Level-sensitive. Transparent while clock is high. Prone to glitches.
- **Flip-Flop (D-FF)**: Edge-triggered (usually rising edge). Samples input only at the exact moment the clock goes from 0 to 1. This is the building block of stable synchronous systems.
- **Register**: A collection of N Flip-Flops sharing a clock. Stores an N-bit word.

## 7.2 Types of Flip-Flops

- **SR Latch (Set-Reset)**: Basic memory. Invalid state if S=1 and R=1.
- **D Flip-Flop (Data)**: Captures the value of input D on the clock edge. Most common in CPUs.
- **JK Flip-Flop**: Like SR, but J=1, K=1 toggles the output. No invalid states.
- **T Flip-Flop (Toggle)**: Toggles output if T=1. Used in counters.

## 7.3 Clocking and Timing

- **Clock Period**: Time between two rising edges. Frequency = 1/Period.
- **Setup Time**: Data must be stable *before* the clock edge.
- **Hold Time**: Data must remain stable *after* the clock edge.
- **Violation**: If setup/hold times are violated, the flip-flop enters a **Metastable State** (neither 0 nor 1), potentially crashing the system.

## 7.4 Finite State Machines (FSM)*

A model of computation used to design control units.

- **State Register**: Holds the current state.
- **Next State Logic**: Combinational logic determining the next state based on current state and inputs.
- **Output Logic**: Determines outputs.
  - **Moore Machine**: Output depends *only* on current state. (Safer, synchronous outputs).
  - **Mealy Machine**: Output depends on current state *and* current inputs. (Faster response, but outputs can glitch).

# 8 Basic CPU Knowledge

## 8.1 The Instruction Cycle

The CPU processes instructions in a continuous loop:

1. **Fetch**: Retrieve the instruction from Flash memory at the address in the PC.
2. **Decode**: The Control Unit interprets the opcode. What is the operation? Where are the operands?
3. **Execute**: The ALU performs the calculation, or memory is accessed.
4. **Writeback**: The result is written back to the Register File.

## 8.2 Pipelining

To increase speed, the AVR overlaps these steps. While one instruction is being **Executed**, the next one is being **Fetched**.

- **Throughput**: Increases (1 instruction completes every clock cycle).
- **Latency**: Stays the same (each instruction still takes time).
- **Hazards**:
  - **Control Hazard**: A branch (JMP) changes the PC, invalidating the instruction already fetched in the pipeline. This wastes cycles (flush).
  - **Data Hazard**: An instruction needs the result of the previous one before it's written back.

## 8.3 Data Path vs Control Path

A CPU consists of two main sections:

1. **Data Path**: The "Muscle". Includes the ALU, Registers, and Buses. It performs the actual operations on data.
2. **Control Path**: The "Brain". The Control Unit (CU). It receives the Opcode and outputs control signals (Select lines for MUXes, Write Enable for registers) to direct the Data Path.

## 8.4 Bus Arbitration

Since the CPU and peripherals (like DMA or USB) might want to access memory simultaneously, an arbiter decides who gets the bus. In simple AVRs, the CPU is the master, but in complex systems, "Cycle Stealing" allows peripherals to access RAM without stopping the CPU.

# 9 Assembly

Assembly language is a human-readable representation of machine code. It provides a 1-to-1 mapping to hardware instructions.

## 9.1 Addressing Modes

How do we specify where the data comes from?

1. **Register Direct**: Operands are in registers.
   - `ADD r16, r17` (r16 = r16 + r17)
2. **Immediate**: Operand is a constant value in the instruction itself.
   - `LDI r16, 50` (Load 50 into r16)
3. **Data Direct**: Operand is at a fixed memory address.
   - `LDS r16, 0x0100` (Load from SRAM address 0x100)
4. **Data Indirect**: Address is held in a Pointer Register (X, Y, Z).
   - `LD r16, X` (Load from address pointed to by X)
5. **Indirect with Displacement**: Address is Pointer + Constant.
   - `LDD r16, Y+5` (Useful for accessing struct fields or stack variables).
6. **Pre-decrement / Post-increment**: Automatically update the pointer.
   - `LD r16, X+` (Load, then increment X. Great for array traversal).

## 9.2 Branching Instructions

Branching controls the flow based on the SREG flags.

| One Form Mnemonic | Common Test | Status Register | Complement Form Mnemonic | Common Test | Status Register | Comment |
|---|---|---|---|---|---|---|
| BRGE | Rd $\geq$ Rr | S == 0 | BRLT | Rd < Rr | S == 1 | Signed |
| BRSH | Rd $\geq$ Rr | C == 0 | BRLO | Rd < Rr | C == 1 | Unsigned |
| BRNE | Rd $\neq$ Rr | Z == 0 | BREQ | Rd == Rr | Z == 1 | Unsigned / Signed |
| BRBC | - | SREG(s) == 0 | BRBS | - | SREG(s) == 1 | - |
| BRCC | - | C == 0 | BRCS | - | C == 1 | Simple |
| BRPL | - | N == 0 | BRMI | - | N == 1 | Simple |
| BRVC | - | V == 0 | BRVS | - | V == 1 | Simple |

## 9.3 Bit Manipulation

Crucial for controlling hardware (setting specific pins).

- `SBI P, b` (Set Bit in I/O Register): Sets bit `b` in Port `P` to 1.
- `CBI P, b` (Clear Bit in I/O Register): Clears bit `b` in Port `P` to 0.
- `SBR r16, K` (Set Bits in Register): Logical OR with constant.
- `CBR r16, K` (Clear Bits in Register): Logical AND with inverse of constant.

## 9.4 Inputs and Outputs (Memory Mapped I/O)

In AVR, I/O pins are controlled by writing to specific memory addresses (Registers).

- **DDRx (Data Direction Register)**: Configures pin as Input (0) or Output (1).
- **PORTx (Data Register)**: Writes output value (High/Low) or enables internal pull-up resistor.
- **PINx (Input Pins Address)**: Reads the actual voltage state of the pin.

## 9.5 Control Flow and Subroutines

- **Branching**: `RJMP` (Relative Jump), `BREQ` (Branch if Equal). Changes the PC to execute code elsewhere.
- **Subroutines**:
  - `RCALL label`: Pushes the current PC (return address) onto the Stack and jumps to label.
  - `RET`: Pops the return address from the Stack into the PC.
  - **Stack Management**: You must ensure the Stack Pointer (SP) is initialized. If you push data inside a function, you must pop it before returning, or the CPU will return to a garbage address (Crash).

# 10 From Assembly to Machine Code

How does human-readable Assembly become binary Machine Code?

## 10.1 The Assembly Process

1. **Parsing**: The assembler reads the source file, stripping comments and whitespace.
2. **Opcode Lookup**: It matches mnemonics (e.g., `ADD`) to their binary Opcode (e.g., `0000 11...`).
3. **Operand Encoding**: It converts registers (`r16` -> `10000`) and constants into binary fields within the instruction word.
4. **Machine Code Generation**: It outputs the final hex/binary file.

## 10.2 The Two-Pass Assembler

A major challenge is **Forward Referencing** (jumping to a label defined later in the code).

- **Pass 1 (Symbol Table Creation)**: The assembler scans the code, counting bytes to determine the address of every label. It builds a **Symbol Table** (Label Name -> Address).
- **Pass 2 (Code Generation)**: It scans again. Now that it knows all addresses, it replaces labels with the actual relative offsets or absolute addresses calculated in Pass 1.

## 10.3 Instruction Encoding Example

Consider `ADD r16, r17`.

- AVR Opcode format for ADD: `0000 11rd dddd rrrr`
- `r` (Source) = 17 (`10001`), `d` (Destination) = 16 (`10000`).
- Result: `0000 1101 0000 0001` -> Hex `0D01`.
- This 16-bit word is what is actually stored in Flash.

# 11 EEPROM and FLASH Memory

## 11.1 Volatile vs Non-Volatile

- **Volatile (SRAM)**: Fast, infinite write cycles, but forgets everything when power is cut.
- **Non-Volatile (Flash, EEPROM)**: Retains data without power.

## 11.2 FLASH (Program Memory)

- **Technology**: NOR Flash.
- **Characteristics**: Read-fast, Write-slow. Must be erased in blocks (pages) before writing.
- **Endurance**: ~10,000 write cycles. Do not use for frequent data logging.
- **Bootloader**: A small program in a protected section of Flash that runs at startup. It allows the MCU to program its own Flash over Serial (how Arduino uploads sketches).

## 11.3 EEPROM (Data Memory)

- **Characteristics**: Byte-addressable (can erase/write single bytes).
- **Endurance**: ~100,000 write cycles. Better than Flash, but still finite.
- **Usage**: Store calibration data, serial numbers, or user settings.

## 11.4 Memory Map

The ATmega2560 has a unified address space for Data, but separate for Code.

- **Flash (Code)**: `0x0000` - `0x3FFFF` (256KB).
- **Data Space**:
  - `0x0000 - 0x001F`: 32 Registers.
  - `0x0020 - 0x005F`: 64 I/O Registers.
  - `0x0060 - 0x01FF`: Extended I/O Registers.
  - `0x0200 - 0x21FF`: Internal SRAM (8KB).

## 11.5 Fuse Bits

Special configuration bits that live outside normal memory. They control critical hardware settings.

- **Clock Source**: Internal RC vs External Crystal.
- **BOD (Brown-out Detection)**: Reset voltage level.
- **JTAG**: Enable/Disable debugging interface.
- *Warning*: Setting the wrong fuses (e.g., disabling the Reset pin or selecting a non-existent clock) can "brick" the chip.

# 12 Interrupts

An interrupt is a hardware mechanism that pauses the main program to handle an urgent event.

## 12.1 Polling vs Interrupts

- **Polling**: The CPU constantly checks a flag in a loop ("Are we there yet?"). Wastes CPU cycles.
- **Interrupt**: The hardware pokes the CPU only when the event happens. Efficient.

## 12.2 The Interrupt Flow

1. **Event**: Timer overflow, Pin change, or Serial data arrival.
2. **Flag**: Hardware sets an Interrupt Flag.
3. **Vector**: If Global Interrupts (I-bit in SREG) are enabled, the CPU:
   - Finishes current instruction.
   - Clears the I-bit (disabling other interrupts).
   - Pushes PC to Stack.
   - Jumps to the **Interrupt Vector** (fixed address in Flash).
4. **ISR**: The Interrupt Service Routine executes.
   - **Context Save**: The ISR *must* save `SREG` and any registers it uses, or the main program will corrupt.
5. **RETI**: Return from Interrupt. Pops PC and re-enables the I-bit.

## 12.3 Interrupt Vector Table

The first locations in Flash memory are reserved for the Vector Table. Each address corresponds to a specific event. - `0x0000`: RESET (Power on) - `0x0002`: INT0 (External Interrupt 0) - `0x0004`: INT1 - ... - `0x002E`: USART0_RX (Serial Data Received)

## 12.4 Implementing Interrupts in Assembly

To use interrupts, you must: 1. **Setup the Vector Table**: Place `JMP` instructions at the start of Flash. 2. **Initialize Stack**: Interrupts push the PC, so SP must be valid. 3. **Enable Specific Interrupt**: Set the mask bit (e.g., `EIMSK` for external pins). 4. **Enable Global Interrupts**: `SEI` instruction.

```
.org 0x0000         ; Reset Vector
    jmp main
.org 0x0002         ; INT0 Vector
    jmp my_isr

main:
    ; ... Setup Stack ...
    sbi EIMSK, INT0 ; Enable INT0
    sei             ; Enable Global Interrupts
loop:
    rjmp loop

my_isr:
    push r16        ; Save context
    in r16, SREG
    push r16
    ; ... Handle Event ...
    pop r16         ; Restore context
    out SREG, r16
    pop r16
    reti
```

# 13 Additional Notes on Atmega2560

## 13.1 Timers

Hardware counters that run independent of the CPU.

- **Prescaler**: Divides the system clock (16MHz) to run the timer slower (e.g., /1024).
- **CTC Mode (Clear Timer on Compare)**: Resets the timer when it hits a specific value. Great for precise frequencies.
- **PWM (Pulse Width Modulation)**: Generates analog-like signals by toggling a pin fast. Used for motor speed or LED brightness.

## 13.2 UART (Serial)

- **Asynchronous**: No clock line. Both sides must agree on the **Baud Rate** (speed).
- **Frame**: Start bit, Data bits (8), Stop bit.
- **Buffer**: The hardware has a small FIFO buffer. If you don't read fast enough, data is lost (Overrun).

## 13.3 ADC (Analog to Digital Converter)

- **Resolution**: 10-bit (returns 0 to 1023).
- **Reference Voltage ($V_{ref}$)**: The ceiling. Input voltage equal to $V_{ref}$ gives 1023.
- **Formula**: $ADC = \frac{V_{in} \times 1024}{V_{ref}}$.
- **Multiplexer**: One ADC unit is shared across 16 pins. You must switch channels to read different pins.

## 13.4 SPI and I2C

- **SPI**: 4 wires (MOSI, MISO, SCK, SS). Full-duplex, very fast. Master-Slave.
- **I2C (TWI)**: 2 wires (SDA, SCL). Half-duplex, slower. Address-based (multiple devices on same wires).

## 13.5 Power Management

Mobile devices need to save battery.

- **Sleep Modes**: The CPU can be put to sleep (halting the clock).
- **Idle Mode**: CPU stops, but SPI, UART, and Timers keep running.
- **Power-down Mode**: Everything stops except external interrupts. Waking up takes time.

## 13.6 Glossary

- **Duplex**:
  - **Half-Duplex**: Data flows one way at a time (Walkie-Talkie).
  - **Full-Duplex**: Data flows both ways simultaneously (Telephone).
- **Baud Rate**: Symbols per second. In binary UART, roughly equals bits per second.
- **Throughput**: How much work the CPU completes in a given time.
- **Latency**: The delay between an input and the response.
- **Edge-Triggered**: Activated by the change in voltage (Low to High).
- **Level-Triggered**: Activated by the constant presence of voltage (High).