CAO Slides

Eduard Fekete

Character Count: 16797
Word Count: 2494

December 23, 2025

1 Comprehensive Course Notes: Computer Architecture and Organization (ATmega2560)

1.1 1. Internal Representation of Data

1.1.1 Units of Memory

To define the size of computer memory (registers, RAM, disk), specific units are used. It is important to distinguish between "old school" binary prefixes and modern metric standards.

Binary Prefixes (IEC Standard): Based on powers of 2.

- 1 kibibyte (KiB) = 2^{10} bytes = 1024 bytes
- 1 mebibyte (MiB) = 2^{20} bytes = 1,048,576 bytes
- 1 gibibyte (GiB) = 2^{30} bytes = 1,073,741,824 bytes
- 1 tebibyte (TiB) = 2^{40} bytes
- 1 pebibyte (PiB) = 2^{50} bytes

Decimal Prefixes (SI Standard): Based on powers of 10.

- 1 kilobyte (KB) = 10^3 bytes = 1000 bytes
- 1 megabyte (MB) = 10^6 bytes = 1,000,000 bytes
- 1 gigabyte (GB) = 10^9 bytes
- 1 terabyte (TB) = 10^{12} bytes
- 1 petabyte (PB) = 10^{15} bytes

1.1.2 Word Size and Addressing

Computer storage is organized into "words". The size of a word depends on the architecture:

- **ATmega2560:** An 8-bit computer, operating on 8-bit words (bytes).
- **Modern CPUs (x86-64):** Typically 64-bit computers operating on 64-bit words (8 bytes).

Overflow: When a calculation results in a value larger than the word size can hold (e.g., adding 1 to a maxed-out 16-bit number like 65,535), an overflow occurs. Detection and handling of overflows are critical in low-level programming.

1.1.3 Hexadecimal Notation

Hexadecimal (base 16) is used as a compact way to write binary.

- It uses symbols 0-9 and a-f.
- One hex digit corresponds exactly to 4 bits (a nibble).
- The prefix `0x` is used to denote hex numbers (e.g., `0xffff`).
- Example conversion: Binary `0111 1010 0010 1111` -> Hex `0x7a2f`.

1.1.4 Two's Complement (Signed Integers)

Two's complement is the standard method for representing negative integers in binary.

- **Range (8-bit):** -128 to +127.
- **MSB (Most Significant Bit):** Indicates the sign (0 = positive, 1 = negative).
- **Conversion Algorithm (Positive to Negative):**
  1. Invert all bits of the positive number ($0 \to 1, 1 \to 0$).
  2. Add 1 to the result.
  3. Ignore any final carry (overflow) beyond the MSB.
- **Example:** To find -27 (8-bit):
  - 27 = `0001 1011`
  - Invert: `1110 0100`

  - Add 1: `1110 0101` (This is -27).
- **Arithmetic:** Subtraction is performed as addition: $A - B = A + (-B)$.
- **Overflow Rule:**
  - If sum of two positive numbers yields a negative result (MSB=1), overflow occurred.
  - If sum of two negative numbers yields a positive result (MSB=0), overflow occurred.

1.1.5 Floating Point Numbers (IEEE 754)

Binary floating-point numbers follow the format: $1.fraction \times 2^{exponent}$.

- **Normalization:** Numbers are formatted so there is exactly one non-zero digit (1) before the binary point. In binary, this leading 1 is implied and not stored, gaining one bit of precision.
- **IEEE 754 Formats:**
  - Half precision (16-bit).
  - Single precision (32-bit): 1 sign bit, 8 exponent bits, 23 fraction bits.
  - Double precision (64-bit): 1 sign bit, 11 exponent bits, 52 fraction bits.
- **Core Issues:**
  - **Approximation:** Numbers like 0.1 cannot be represented exactly in binary.
  - **Equality Checks:** Never use == for floats; use a tolerance comparison ($|a - b| < \epsilon$).
  - **Accumulation:** Adding small numbers to large ones can result in the small numbers vanishing.
  - **Associativity:** $(a + b) + c$ is not necessarily equal to $a + (b + c)$.

1.1.6 Text Representation

- **ASCII (1963):** Uses 7 bits to represent English characters. Limited range.
- **Unicode:** Designed to support all world alphabets (scripts). Assigns a unique "codepoint" to every symbol (e.g., U+03A8).
  - **UTF-8:** A variable-length encoding (1-4 bytes) for Unicode, widely used on the web. It is backward compatible with ASCII.
  - **UTF-16 / UTF-32:** Use 16 or 32 bits respectively per codepoint.

1.2 2. Boolean Algebra and Digital Logic

1.2.1 Boolean Algebra

Based on variables that are either true (1) or false (0).

- **Basic Operators:** AND ($*$), OR ($+$), NOT (!).
- **Precedence:** NOT $\rightarrow$ AND $\rightarrow$ OR.
- **Key Laws:**
  - **Identity:** $A + 0 = A$, $A * 1 = A$.
  - **Null:** $A + 1 = 1$, $A * 0 = 0$.
  - **Idempotent:** $A + A = A$, $A * A = A$.
  - **Complement:** $A + !A = 1$, $A * !A = 0$.
  - **Double Negation:** $!!A = A$.
  - **De Morgan's:** $!(A * B) = !A + !B$ and $!(A + B) = !A * !B$.
  - **Distributive:** $A * (B + C) = (A * B) + (A * C)$.
  - **Absorption:** $A + (A * B) = A$ and $A * (A + B) = A$.

1.2.2 Logic Gates

Hardware implementations of Boolean expressions. Transistors are the physical building blocks.

- **Basic Gates:** AND, OR, NOT.
- **Universal Gates:** NAND and NOR (can build any other gate from these).
- **XOR (Exclusive OR):** Output is 1 if inputs are different.
- **XNOR:** Inverse of XOR.

1.2.3 Combinatorial Logic

Circuits where output depends only on the current input (stateless).

- **Half Adder:** Adds two bits, produces a Sum and a Carry.
- **Full Adder:** Adds two bits plus an incoming Carry.
- **Multiplexer (MUX):** Selects one input from many to send to the output based on selector bits.
- **Demultiplexer (DEMUX):** Routes one input to one of many outputs.

Truth Table to Boolean Expression:

1. Identify rows in the truth table where the output is 1 (True).
2. Create an AND expression for each row (e.g., if A=0, B=1, use $!A * B$).
3. Combine all AND expressions with OR.
4. Simplify the resulting expression using Boolean laws or Karnaugh maps.

1.3 3. Sequential Logic and CPU Basics

1.3.1 Sequential Logic

Circuits that possess state/memory. Output depends on input *and* current state.

- **Latch:** Basic storage element, often level-triggered.
  - **SR Latch:** Set/Reset. Has an undefined state when S=1 and R=1.
- **Flip-flops:** Edge-triggered storage elements synced to a clock.
  - **SR Flip-flop:** Adds clock synchronization to the latch.
  - **D Flip-flop (Data):** Captures input D on the rising clock edge. Used for basic registers.
  - **JK Flip-flop:** Improved SR flip-flop. Toggles output when J=1 and K=1. Can act as a frequency divider.
- **Applications:**
  - **Frequency Divider:** Using Toggling JK flip-flops.
  - **Counters:** Chaining flip-flops to count clock pulses.

1.3.2 CPU Architecture (Reference Model)

- **ALU (Arithmetic Logic Unit):** Performs math (+, -) and logic (AND, OR). Operates on registers.
- **Control Unit (CU):** Decodes instructions and orchestrates data flow between ALU, registers, and memory.
- **Registers:** Small, fast internal storage (e.g., General Purpose Registers, Instruction Register).
- **Buses:**
  - **Address Bus:** Carries the location address. Width determines maximum memory capacity.
  - **Data Bus:** Carries the actual information. Width determines word size.
- **Clocks:** Provide the heartbeat for synchronization. Higher frequency = more operations per second (generally).

1.3.3 Memory Hierarchy

Trade-off between speed (bandwidth/latency) and capacity/cost.

1. **Level 0: Registers:** Inside CPU, immediate access.
2. **Level 1: Cache (SRAM):** Very fast, stores copies of frequently used data.
3. **Level 2: Main Memory (DRAM):** Slower, larger capacity, volatile.
4. **Level 3: Storage (Flash/Disk):** Slowest, non-volatile, huge capacity.

1.4 4. Introduction to ATmega2560

1.4.1 Specifications

- **Type:** 8-bit AVR Microcontroller (SoC - System on Chip).

- **Performance:** ~16 MIPS at 16MHz (most instructions take 1 clock cycle).
- **Memory:**
  - **Flash:** 256KB (Program Code).
  - **SRAM:** 8KB (Data).
  - **EEPROM:** 4KB (Persistent Data).
- **Registers:** 32 General Purpose Registers (R0-R31).
- **Peripherals:** GPIO, Timers, ADC, USART, SPI, I2C.

1.4.2 Memory Layout (Harvard Architecture)

The ATmega2560 uses separate memory spaces/buses for code and data.

1. Program Memory (Flash):

- Stores machine code.
- Addresses `0x00000` to `0x1FFFF`.
- Addressed as 16-bit words (128k words total).

2. Data Memory (SRAM):

- Stores data. Addresses `0x0000` to `0xFFFF`.
- **Layout:**
  - `0x0000 - 0x001F`: General Purpose Registers (R0-R31).
  - `0x0020 - 0x005F`: Standard I/O Registers.
  - `0x0060 - 0x01FF`: Extended I/O Registers.
  - `0x0200 - 0x21FF`: Internal SRAM.
  - `0x2200 - 0xFFFF`: External SRAM (if attached).

1.4.3 Machine Code vs. Assembly

- **Machine Code:** Raw binary/hex instructions executed by the CPU (e.g., `01 e2`).
- **Assembly:** Human-readable mnemonics (e.g., `LDI R16, 0x21`). The assembler translates this 1:1 into machine code, handling labels and constants.
- **Assembler Directives:** Commands for the assembler (e.g., `.EQU`, `.ORG`) not executed by the CPU.

1.5 5. Assembly Programming Constructs

1.5.1 Instruction Format

Format: `[Opcode Operand(s)]`.

- Can be 16-bit or 32-bit (2 or 4 bytes).
- **Operands:** Can be registers, immediate values, or memory addresses.

1.5.2 Special Registers

- **Program Counter (PC):** Holds the address of the next instruction to execute.
- **Stack Pointer (SP):** Points to the top of the stack in SRAM. Grows from high addresses (`RAMEND`) to low addresses.
  - Use `PUSH` to store, `POP` to retrieve.
  - Used for subroutines and interrupts to store return addresses.
- **Status Register (SREG):** Updated after ALU operations.
  - **C (Carry):** Carry out occurred.
  - **Z (Zero):** Result was zero.
  - **N (Negative):** Result was negative (MSB=1).
  - **V (Two's Complement Overflow):** Arithmetic overflow.
  - **I (Global Interrupt Enable):** Must be 1 for interrupts to work.

1.5.3 Pseudo-registers

Pairs of 8-bit registers treated as 16-bit pointers for addressing memory:

- **X Register:** R27:R26
- **Y Register:** R29:R28
- **Z Register:** R31:R30

1.6 6. Instruction Set Details

1.6.1 Data Transfer

- `LDI Rd, K`: Load Immediate (load constant K into register Rd).
- `MOV Rd, Rr`: Copy value from Rr to Rd.
- `IN Rd, Port`: Read from I/O port (0x00-0x3F) into Rd.
- `OUT Port, Rr`: Write to I/O port (0x00-0x3F) from Rr.
- `LD Rd, X`: Load indirect from data space using pointer X.
- `ST X, Rr`: Store indirect to data space using pointer X.
- `LDS Rd, k` / `STS k, Rr`: Load/Store direct from/to SRAM address k.

1.6.2 Arithmetic

- `ADD Rd, Rr`: Add without carry.
- `ADC Rd, Rr`: Add with carry (essential for multi-byte arithmetic).
- `SUB Rd, Rr`: Subtract without carry.
- `SBC Rd, Rr`: Subtract with carry.
- `INC Rd` / `DEC Rd`: Increment / Decrement.
- **Note:** `INC` and `DEC` do not update the Carry flag.

1.6.3 Control Flow

Unconditional:

- `RJMP k`: Relative Jump (PC + offset).
- `JMP k`: Absolute Jump (full address space).
- `CALL k`: Call subroutine (pushes PC+2 to stack).
- `RET`: Return from subroutine (pops PC from stack).

Comparison:

- `CP Rd, Rr`: Compare registers (performs subtraction, updates flags, discards result).
- `CPI Rd, K`: Compare register with immediate.

Conditional Branching (relies on SREG):

- `BREQ`: Branch if Equal (Z=1).
- `BRNE`: Branch if Not Equal (Z=0).
- `BRSH`: Branch if Same or Higher (Unsigned, C=0).
- `BRLO`: Branch if Lower (Unsigned, C=1).
- `BRMI`: Branch if Minus (N=1).
- `BRPL`: Branch if Plus (N=0).
- `SBRC Rr, b` / `SBRS Rr, b`: Skip next instruction if bit `b` in `Rr` is Cleared/Set.

1.6.4 Subroutines

1. **Call:** `CALL label` pushes return address to stack.
2. **Save Context:** Subroutine should `PUSH` any registers it plans to modify.
3. **Body:** Execute task.
4. **Restore Context:** `POP` registers in reverse order.

5. **Return:** `RET` pops return address to PC.

1.7 7. I/O Ports and Advanced Configuration

1.7.1 Digital I/O Architecture

Each port (A, B, C…) is controlled by three 8-bit registers:

1. **DDRx (Data Direction Register):** Configures pins.
   - 0 = Input.
   - 1 = Output.
2. **PORTx (Data Register):**
   - If Output (DDR=1): Sets pin level (1=High, 0=Low).
   - If Input (DDR=0): Activates internal Pull-up resistor (1=Pull-up On, 0=Floating).
3. **PINx (Input Pins Address):** Reads the actual logic level on the physical pins.

1.7.2 Pull-Up Resistors

- **Floating Inputs:** Unconnected inputs fluctuate randomly (noise).
- **Pull-up:** Connects input to Vcc via a resistor. Default state is High. Closing a switch to Ground makes it Low.
- ATmega2560 has built-in internal pull-ups configurable via software (set DDR=0, PORT=1).

1.7.3 Bit Manipulation Instructions

Special instructions for I/O registers in range 0x00-0x1F:

- `SBI Port, Bit`: Set Bit in I/O register (e.g., set LED on).
- `CBI Port, Bit`: Clear Bit in I/O register (e.g., turn LED off).
- `SBIC Port, Bit` / `SBIS Port, Bit`: Skip next instruction if bit in I/O register is Cleared/Set.

1.7.4 m2560def.inc

A standard include file in Microchip Studio projects. It defines symbolic names for all registers (e.g., `PORTA`, `DDRB`, `SPL`) and bit names, mapping them to their hex addresses.

1.8 8. Advanced Addressing and Compilers

1.8.1 Addressing Modes

1. **Register Direct:** `ADD R16, R17`.
2. **I/O Direct:** `OUT PORTB, R16`.
3. **Data Direct:** `LDS R16, 0x1234`.
4. **Data Indirect:** Uses X, Y, or Z as pointers.
   - `LD R16, X`: Load from address in X.
   - **Post-increment:** `LD R16, X+`. Load, then increment X (X = X + 1). Efficient for looping through arrays.
   - **Pre-decrement:** `LD R16, -X`. Decrement X (X = X - 1), then load. Efficient for stack operations.

1.8.2 Compilers (C to Assembly)

- Compilers translate high-level code (C/C++) into assembly.
- **Pipeline:** Source Code → Preprocessing → Syntax/Semantic Analysis → Intermediate Code → Optimization → Assembly Code → Binary Object File → Linker → Executable.
- The compiler manages complex tasks like register allocation, stack frame management, and optimizations.

1.9 9. Persistent Storage

1.9.1 Flash Memory

- **Technology:** Uses floating-gate MOSFETs to trap electrons.
- **Characteristics:**
  - High voltage required to erase/write (~20V internal charge pump).
  - **Wear:** The insulator degrades with write cycles. Wear leveling is used to distribute writes.
  - **NAND Flash:** Organized in Pages and Blocks. Must erase a whole Block before writing a Page. Writing changes 1s to 0s; erasing resets 0s to 1s.

1.9.2 EEPROM (Electrically Erasable Programmable Read-Only Memory)

- **Use Case:** Storing configuration/calibration data.
- **Specs:** 4KB on ATmega2560. Byte-addressable.
- **Lifespan:** ~100,000 write/erase cycles.
- **Access:** Slower than RAM. Accessed via I/O registers:
  - `EEAR` (Address Register).
  - `EEDR` (Data Register).
  - `EECR` (Control Register).
- **Procedure:** Writing requires a specific timed sequence to `EECR` (Master Write Enable followed by Write Enable) to prevent accidental writes.

1.10 10. Interrupts

1.10.1 Definition and Purpose

An interrupt is a hardware signal that temporarily halts the main program flow to handle an urgent event.

- **vs. Polling:** Polling wastes CPU cycles checking flags. Interrupts are event-driven and efficient.
- **Latency:** Interrupt response is fast and deterministic.

1.10.2 Interrupt Vector Table

A reserved section of code memory (starting at 0x0000) containing `JMP` instructions to specific Interrupt Service Routines (ISRs).

- **Priority:** Lower address = Higher priority.
- **Reset:** Vector 1 (0x0000).
- **External Interrupts:** INT0, INT1, etc.
- **Timers/Serial:** Have their own vectors.

1.10.3 Interrupt Service Routine (ISR) Workflow

1. **Trigger:** Event occurs (e.g., pin change, timer overflow).
2. **Finish:** CPU completes current instruction.
3. **Context Save (Hardware):** CPU pushes PC (return address) to stack. Global interrupts are disabled (I-bit cleared).
4. **Jump:** CPU executes instruction at the vector address (usually a Jump to ISR).
5. **Context Save (Software):** ISR *must* push `SREG` and any modified registers to the stack.
6. **Execute:** Perform the urgent task.
7. **Restore Context:** Pop registers and `SREG` from stack.
8. **Return:** `RETI` instruction pops PC from stack and re-enables global interrupts.

1.10.4 Enabling Interrupts

Requires three conditions:

1. **Global Enable:** Set I-bit in SREG (`SEI` instruction).
2. **Local Enable:** Set the specific interrupt enable bit (e.g., `EIMSK` for external interrupts).
3. **Event Configuration:** Configure the trigger condition (e.g., `EICRA` for rising/falling edge).