SEMICONDUCTOR ENGINEERING: NUMERICAL ANALYSIS & COMPUTER SIMULATIONS

Lecture 1: Fundamentals of Computer Simulation & Error Analysis

1. Course Introduction & Logistics

- **Instructor**: [Professor’s Name]
- **Topics**: Numerical analysis, computer simulations (Ch 1-7 by me)
- **Today’s Focus**: Fundamentals of computer simulation, sources of computational errors
- **Recommended Texts**:
  - “Numerical Analysis”, “Numerical Simulation”
  - “Numerical Recipes” for algorithms & programming
- **Programming Tools**:
  - Text Editor: **Microsoft Visual Studio Code** recommended

2. Today's Assignment (Due: Midnight, June 11th)

Problem 1: Number Base Conversion (Manual Calculation Required) 1. Convert $(101001)_2$ to Base 10. 2. Convert $(4251)_{10}$ to Base 16. * *Please solve manually first; programs can be used for verification.*

Problem 2: Python Program Analysis 1. Choose one Python program from lecture materials. 2. Explain what each block/part of the source code does. 3. If unclear, list the parts you don't understand and explain *why*. * *Objective: Engage with code, even if not fully understood.*

3. Fundamentals of Computer Representation

- **Binary Nature**: Computers operate using binary (base 2) states (0 or 1).
  - CPU & memory are built with binary logic.
  - Primitive expression in computers is Base 2.
- **Human Convenience**: Base 2 is verbose. We often use:
  - **Base 8 (Octal)**: Digits 0-7.
  - **Base 16 (Hexadecimal)**: Digits 0-9, A-F (A=10, F=15).

3.1 Number Base Conversion: Base-r to Base-10

- **General Formula**: For a number $(a_n a_{n-1} \dots a_1 a_0)_r$:
- $(a_n a_{n-1} \dots a_1 a_0)_r = a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + \cdots + a_1 \cdot r^1 + a_0 \cdot r^0$
- **Example 1: Decimal (Base 10)** $(1975)_{10} = 1 \cdot 10^3 + 9 \cdot 10^2 + 7 \cdot 10^1 + 5 \cdot 10^0 = 1975$
- **Example 2: Binary (Base 2)** $(11011)_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 0 + 2 + 1 = 27_{10}$
- **Example 3: Octal (Base 8)** $(53)_8 = 5 \cdot 8^1 + 3 \cdot 8^0 = 40 + 3 = 43_{10}$
- **Example 4: Hexadecimal (Base 16)** $(2F)_{16} = 2 \cdot 16^1 + F \cdot$

3.2 Number Base Conversion: Base-10 to Base-r

- **Method**: Repeated division by `r` and collecting remainders in reverse order.
- **Example: Convert 39_{10} to Base 8**
  1. $39 \div 8 = 4$ remainder 7
  2. $4 \div 8 = 0$ remainder 4
  3. Reading remainders upwards: $(47)_8$
- *Verification*: $4 \cdot 8^1 + 7 \cdot 8^0 = 32 + 7 = 39_{10}$

3.3 Data Storage Units: Bits & Bytes

- **Bit (b)**: Smallest unit of data (0 or 1).
- **Byte (B)**: Fundamental group of 8 bits.
  - Can represent $2^8 = 256$ values (0-255).
- **Prefixes (Binary vs. Decimal)**:
  - **Kilobyte (KB)**: 2^{10} bytes $= 1024$ bytes
  - **Megabyte (MB)**: 2^{20} bytes $= 1{,}048{,}576$ bytes
  - **Gigabyte (GB)**: 2^{30} bytes $= 1{,}073{,}741{,}824$ bytes
  - **Terabyte (TB)**: 2^{40} bytes $= 1{,}099{,}511{,}627{,}776$ bytes
- **Note**: Capital ‘B’ for Byte, lowercase ‘b’ for bit (e.g., Mbps = Megabits per second).

4. Numerical Representation in Computer Programs

4.1 Integer Data Types (Whole Numbers)

- **Signed vs. Unsigned**:
  - **Unsigned**: Non-negative only (0 to $2^n - 1$).
  - **Signed**: Positive and negative (typically $-(2^{n-1})$ to $2^{n-1} - 1$).
- **Common Sizes**:
  - **16-bit**:
    - Unsigned: 0 to $65{,}535$
    - Signed: $-32{,}768$ to $32{,}767$
  - **32-bit**:
    - Unsigned: 0 to 4.29×10^9
    - Signed: -2.14×10^9 to 2.14×10^9

4.2 Floating-Point Data Types (Real Numbers)

- **Purpose**: Represent numbers with fractional parts (real numbers).
- **Standard: IEEE 754** standard for consistent representation.
- **Types**:
  - **Binary32 (Single Precision)**: 32 bits
  - **Binary64 (Double Precision)**: 64 bits (common default)
  - **Binary128 (Quad Precision)**: 128 bits
- **Structure**: $\pm(1.M)_2 \times 2^E$
  - **Sign bit**: 1 bit ($\pm$)

4.2 Floating-Point Data Types (Cont.)

- **Binary64 (Double Precision)**:
  - 1 bit sign
  - 11 bits exponent
  - 52 bits mantissa (fraction)
  - **Precision**: $\approx 15 - 17$ decimal digits.
  - **Range**: $\approx 10^{-308}$ to 10^{308}.
- **Precision in Semiconductor Physics**:
  - Energy scales: meV to MeV (e.g., $k_B T \approx 26$ meV at 300K, core electron energies can be keV).

5. Sources of Numerical Errors in Computation

- **Computers use finite precision**: Real numbers (infinite digits) must be approximated.
- **Machine Epsilon**: Smallest number such that $1 + \epsilon \neq 1$. Fundamental limit of floating-point precision.

5.1 Round-off Error

- **Definition**: Errors from inexact representation of real numbers in finite binary digits.
- **Example**: 0.1_{10} cannot be exactly represented in binary.
  - $(0.1)_{10} = (0.0001100110011\ldots)_2$ (repeating)

5.2 Overflow and Underflow

- **Overflow**: Result is too large for the data type.
  - Ex: Product of two large `double`s exceeds 10^{308}.
- **Underflow**: Result is too small (too close to zero) to be represented accurately, often rounded to zero.
  - Ex: A `double` cannot represent numbers smaller than $\approx 10^{-308}$.
- **Physical Example (Boltzmann Factor $\exp(-E/k_B T)$)**:
  - $E_g = 1.1$ eV (Silicon), $k_B T = 62$ meV (specific context): $\exp(-1.1/0.062) \approx \exp(-17.7) \approx 10^{-7.7}$ (No issue).
  - $E_g = 4$ eV (Oxide), $k_B T = 62$ meV: $\exp(-4/0.062) \approx$

5.3 Truncation Error

- **Definition**: Error from approximating an infinite mathematical process with a finite one.
- **Example**: Using a finite number of terms in a Taylor series expansion:
- $f(x) = \sum_{n=0}^{N} \frac{f^{(n)}(a)}{n!}(x-a)^n + R_N(x)$
  - $R_N(x)$ is the truncation error.

5.4 Convergence Error

6. Practical Implications & Avoiding Errors

6.1 Floating-Point Comparisons (`if (x == y)`)

- **Problem**: Direct equality comparison of floats is unreliable due to round-off error.
  - `if (3.0 * 10.0 == 30.0)` might be `False`!
- **Solution**: Compare absolute difference with a small tolerance (epsilon).
- if $|\mathrm{val}_1 - \mathrm{val}_2| < \mathrm{EPSILON}$
  - `EPSILON` (e.g., 10^{-9} or 10^{-12}) accounts for small inaccuracies.

6.2 Converting Floating-Point to Integer

- **Problem**: `int(9.999999999999999)` might yield `9` instead of `10`.
- **Solution**: Add a small epsilon before conversion.
- int_value = int(floating_value + EPSILON)
  - This “nudges” values slightly below an integer threshold up.

6.3 Information Buried (Catastrophic Cancellation)

- **Problem**: Subtracting large, nearly equal numbers leads to significant digit loss.
- **Example**: Calculating $\exp(-x)$ for large positive x using its Taylor series:
- $\exp(-x) = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$
  - For $x = 40$, intermediate terms are very large, leading to significant cancellation and an incorrect result (e.g., 5.88 instead of 4.25×10^{-18}).
- **Solution**: Reformulate the expression to avoid cancellation

7. Conclusion & Assignment Review

- **Key Takeaways**:
  - Computers use binary; other bases are for human convenience.
  - Data types (int, float) have finite precision and range.
  - Numerical errors (round-off, overflow, underflow, truncation, cancellation) are inherent.
  - **Crucial**: Understand and mitigate these errors for reliable simulations.
- **Assignment Reminder**:
  - Problem 1: Base conversion (manual).
  - Problem 2: Python code analysis.