Semiconductor Engineering: Numerical Analysis and Computer Simulations

Chapter 1: Course Introduction and Fundamentals of Computer Simulation

1.1 Introduction to the Course

Welcome to this course on Numerical Analysis and Computer Simulations in Semiconductor Engineering. My name is [Professor's Name, or 'I' as in the first person narrative, since I'm acting as the professor], and I will be guiding you through the first half of this semester, covering topics from Chapter 1 to Chapter 7. Our primary focus will be on the essential numerical analysis techniques required to understand, develop, and implement programs for sophisticated analyses and computer simulations pertinent to semiconductor devices and materials.

Today's lecture, the first in this series, will lay the groundwork by introducing the fundamental principles of computer simulation. We will also delve into the various sources of errors that are inherent in computational calculations, a critical aspect for ensuring the reliability and accuracy of our simulations.

1.2 Course Logistics and Resources

1.2.1 Textbooks and References For those who prefer an English textbook, a simple search for "numerical analysis" or "numerical simulation" will yield numerous excellent resources. In particular, for practical algorithms and programming techniques, I highly recommend the "Numerical Recipes" series. These books provide a comprehensive collection of algorithms and their implementations, serving as an invaluable reference for advanced study.

1.2.2 Programming Environment While word processors like Microsoft Word are suitable for document creation, they are generally not ideal for software development. For writing and editing program code, a dedicated text editor is essential. I recommend using **Microsoft Visual Studio Code** if you do not already have a preferred text editor. It offers robust features for code development across various programming languages.

1.2.3 Generative AI Tools The field of generative AI, such as ChatGPT, has advanced rapidly in recent years and can be a powerful learning aid. You are permitted to use generative AI for your assignments. However, it is **crucial** that your submissions include your own critical thought, analysis, and improvements derived from the AI's output. Merely copying and pasting AI-generated content will result in a non-evaluated submission. The goal is to leverage AI as a tool for learning and problem-solving, not as a substitute for your own intellectual effort.

1.2.4 Assignments and Grading There will be **no final examination** for this course. Instead, your performance will be evaluated based on a **term-end assignment**. Specific details regarding this assignment will be provided later in the semester.

1.2.5 Class Recordings and Communication Each lecture will be recorded. If you are unable to attend a class or wish to review the material, please inform me via email or Slack, and I will ensure you have access to the recording. For any questions or clarifications during or after the lecture, please use the Q&A box or contact me directly.

1.3 Today's Assignment

To help you focus on the key concepts, I am providing today's assignment at the beginning of the lecture. Please submit your answers via the Learning Management System (LMS) within approximately two days, by midnight on June 11th. If you encounter issues with LMS access, you may email your answer file, ensuring the filename includes your student ID and full name.

Problem 1: Number Base Conversion You are required to perform the following number base conversions: 1. Convert the binary number `101001_2` to its decimal (base 10) equivalent. 2. Convert the decimal number `4251_10` to its hexadecimal (base 16) equivalent.

We will cover the necessary methods for these conversions during today's lecture. Please solve these problems manually without relying on any programming tools, though you are welcome to use a program to verify your answers afterward.

Problem 2: Python Program Analysis In the provided lecture materials (typically a zip file), you will find several Python programs. Choose one of these programs and provide an explanation of what each block or significant part of the source code does. If you cannot fully understand a specific part, you should list the problematic code sections and articulate why you find them difficult to understand or what their purpose seems to be. Even if your answer states, "I couldn't understand anything," it will be accepted, provided you demonstrate a genuine attempt to engage with the code. The aim of this problem is to encourage you to start interacting with and analyzing computational code.

1.4 Fundamentals of Computer Representation

1.4.1 Binary Nature of Computers At its most fundamental level, a computer represents all numerical data using electronic hardware components that have two distinct states, typically denoted as 0 and 1. This is known as **binary representation**. The core building blocks of a computer, such as the Central Processing Unit (CPU) and memory, are constructed from binary logic gates and memory cells. Consequently, the most primitive form of data expression in a computer is based on base 2.

While binary is fundamental, it often requires a large number of digits to represent even small values, making it inconvenient for human comprehension. For this reason, other number bases, such as octal (base 8) and hexadecimal (base 16), are commonly used as more compact representations of binary data, especially in programming and hardware contexts.

1.4.2 Number Systems and Base Conversion

A number in any base `r` can be generally represented as a sequence of digits $(a_n a_{n-1} \dots a_1 a_0)_r$.

1.4.2.1 Converting from Base-r to Base-10 To convert a number from base `r` to base 10, we use a positional notation where each digit a_i is multiplied by the base `r` raised to the power of its position `i` (starting from 0 for the rightmost digit). The sum of these products gives the base-10 equivalent.

Formula: $(a_n a_{n-1} \dots a_1 a_0)_r = a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + \cdots + a_1 \cdot r^1 + a_0 \cdot r^0$

- **Example 1: Base 10 (Decimal)** A decimal number like $(1975)_{10}$ is inherently understood this way: $1 \cdot 10^3 + 9 \cdot 10^2 + 7 \cdot 10^1 + 5 \cdot 10^0 = 1000 + 900 + 70 + 5 = 1975$

- **Example 2: Base 2 (Binary)** Consider the binary number $(11011)_2$: $1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 0 + 2 + 1 = 27_{10}$ Thus, $(11011)_2$ is equivalent to 27_{10}.

- **Example 3: Base 8 (Octal)** Octal numbers use digits from 0 to 7. For $(53)_8$: $5 \cdot 8^1 + 3 \cdot 8^0 = 40 + 3 = 43_{10}$

- **Example 4: Base 16 (Hexadecimal)** Hexadecimal numbers use digits 0-9 and letters A-F, where A=10, B=11, C=12, D=13, E=14, F=15. For $(2F)_{16}$: $2 \cdot 16^1 + F \cdot 16^0 = 2 \cdot 16 + 15 \cdot 1 = 32 + 15 = 47_{10}$ The maximum value for a two-digit hexadecimal number is $(FF)_{16}$, which is $15 \cdot 16^1 + 15 \cdot 16^0 = 240 + 15 =$

255_{10}. This range (0-255) is important as it corresponds to the range representable by one byte (8 bits).

1.4.2.2 Converting from Base-10 to Base-r Converting a decimal number to another base `r` is typically done using the method of **repeated division and remainder collection**. You repeatedly divide the decimal number by `r`, recording the remainders. The base-r number is then formed by reading the remainders from bottom to top (last remainder first).

- **Algorithm:**
  1. Divide the decimal number `N` by `r`.
  2. The remainder is the rightmost digit of the base-r number.
  3. Take the quotient and repeat steps 1 and 2 until the quotient becomes 0.
  4. Collect the remainders in reverse order.
- **Example: Convert 39_{10} to Base 8**
  1. $39 \div 8 = 4$ remainder 7 (rightmost digit)
  2. $4 \div 8 = 0$ remainder 4 (next digit)
  3. The quotient is 0, so we stop.
  4. Reading remainders from bottom to top: 47_8. Thus, 39_{10} is equivalent to $(47)_8$.

1.4.3 Data Storage Units: Bits and Bytes

In computer hardware, the most fundamental unit of data is a **bit**, representing a binary digit (0 or 1). However, data is frequently processed and stored in groups of bits.

- **Byte (B)**: A standard unit, comprising 8 bits. One byte can represent $2^8 = 256$ different values (from 0 to 255 for unsigned integers).
- **Larger Units**: For larger data volumes, prefixes like kilo, mega, giga, and tera are used. It's crucial to understand the distinction between decimal (base 10) prefixes and binary (base 2) prefixes in computing:
  - **Kilobyte (KB)**: Traditionally 1024 bytes (2^{10} bytes). Note the uppercase 'B' for byte. If you see 'kb' (lowercase 'b'), it typically means kilobits.
    - 1 KB $= 2^{10}$ bytes $= 1024$ bytes
  - **Megabyte (MB)**: 2^{10} KB $= 2^{20}$ bytes $= 1{,}048{,}576$ bytes.
  - **Gigabyte (GB)**: 2^{10} MB $= 2^{30}$ bytes $= 1{,}073{,}741{,}824$ bytes.
  - **Terabyte (TB)**: 2^{10} GB $= 2^{40}$ bytes $= 1{,}099{,}511{,}627{,}776$ bytes.

This distinction is important because hard drive manufacturers often use decimal prefixes (e.g., 1 TB = 10^{12} bytes), while operating systems typically report capacities using binary prefixes (e.g., 1 TB = 2^{40} bytes), leading to perceived discrepancies.

1.5 Numerical Representation in Computer Programs

Computer programs need to handle various types of numbers, not just integers. The way these numbers are stored and manipulated dictates the precision and range available for calculations.

1.5.1 Integer Data Types Integer types represent whole numbers without fractional components. Their range and memory footprint depend on the number of bits allocated to them.

- **Signed vs. Unsigned Integers**:
  - **Unsigned integers** can only represent non-negative values (0 and positive integers). An n-bit unsigned integer can represent values from 0 to $2^n - 1$.
  - **Signed integers** can represent both positive and negative values. Typically, one bit is used for the sign (e.g., 0 for positive, 1 for negative), reducing the range for the magnitude. An n-bit signed integer typically represents values from $-(2^{n-1})$ to $2^{n-1} - 1$.
- **Common Integer Bit Lengths**:
  - **16-bit Integer**:
    - Unsigned: 0 to $65{,}535$
    - Signed: $-32{,}768$ to $32{,}767$
  - **32-bit Integer**:
    - Unsigned: 0 to $4{,}294{,}967{,}295$
    - Signed: $-2{,}147{,}483{,}648$ to $2{,}147{,}483{,}647$
  - **64-bit Integer (Long Long Integer in C++)**:
    - Unsigned: 0 to 1.84×10^{19}
    - Signed: -9.22×10^{18} to 9.22×10^{18}

The standard length for integer types is often determined by the underlying CPU architecture. Modern CPUs are predominantly 64-bit, making 64-bit integers a common default. However, for calculations involving extremely large integers (e.g., numbers with trillions of digits like π), specialized software implementations (often called "multi-precision arithmetic") are required, as standard hardware integer types cannot accommodate such magnitudes.

1.5.2 Floating-Point Data Types To represent real numbers, which include fractional parts and can span a much wider range than integers, computers use **floating-point data types**. These types approximate real numbers using a fixed number of bits to represent the sign, exponent, and mantissa (fractional part).

- **IEEE 754 Standard**: The representation of floating-point numbers is standardized by the IEEE 754 standard, ensuring consistency across different hardware and software platforms. This standard defines several formats, commonly referred to as:
  - **Binary32 (Single Precision)**: 32 bits
  - **Binary64 (Double Precision)**: 64 bits
  - **Binary128 (Quad Precision)**: 128 bits
- **Structure of an IEEE 754 Floating-Point Number**: A floating-point number is typically stored in the format $\pm(1.M)_2 \times 2^E$, where:
  - **Sign bit**: 1 bit (0 for positive, 1 for negative).
  - **Exponent**: Represents the power of 2, often with a bias.
  - **Mantissa (or Fraction/Significand)**: Represents the fractional part (the $(1.M)_2$ part, where the leading '1' is implicit for normalized numbers).
- **Binary64 (Double Precision) Details**: A 64-bit double-precision floating-point number is allocated as follows:
  - 1 bit for the sign.
  - 11 bits for the exponent.
  - 52 bits for the mantissa (fractional part after the implicit leading '1'). This configuration allows it to represent approximately 15-17 decimal digits of precision and a very wide range of magnitudes (from about 10^{-308} to 10^{308}).
- **Precision Requirements in Semiconductor Physics**: In semiconductor engineering, calculations often require very high precision. For instance, consider the energy scales in quantum mechanics:
  - The 1s orbital energy of a hydrogen atom is -13.6 eV. Heavier atoms exhibit even larger energy scales (e.g., hundreds of keV for core electrons).
  - Thermal energy ($k_B T$) at room temperature (approximately 300K) is about 26 meV (or ~62 meV in some contexts, as mentioned in the original transcript, which corresponds to a higher temperature or specific energy range).
  - Magnetic interaction energies can be in the range of milli-electron volts ($\sim$ several meV).

  To accurately simulate systems that involve a broad spectrum of energy scales, from milli-electron volts to mega-electron volts (a range of over nine orders of magnitude), we need numbers with sufficient decimal precision. A 64-bit floating-point type, offering about 15-17 decimal digits of precision, is generally the minimum requirement. For even greater precision, 128-bit (quad-precision) or higher is used. However, quad-precision is often implemented in software rather than hardware, leading to significantly slower computations.

1.6 Sources of Numerical Errors in Computation

Numerical calculations performed by computers are subject to various types of errors due to the finite nature of digital representation and approximation methods. Understanding these errors is crucial for designing robust and accurate simulation programs.

1.6.1 Machine Epsilon and Floating-Point Representation Issues Real numbers often have an infinite number of digits (e.g., π, $1/3$). Computers, however, must store these numbers using a finite number of bits. This inherent limitation leads to **round-off error**.

- **Exact Representation**: Some numbers can be represented exactly in binary floating-point. For example:
  - $1.0_{10} = (1.0)_2 \times 2^0$
  - $0.5_{10} = (1.0)_2 \times 2^{-1}$
  - $0.125_{10} = (1.0)_2 \times 2^{-3}$
  - $100.0_{10} = (1.1001)_2 \times 2^6$ (This is $1 \cdot 2^2 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$ is wrong, $100_{10} = (1100100)_2 = 1.100100 \times 2^6$. It has a finite binary representation.)
- **Inexact Representation**: Many common decimal numbers cannot be represented exactly in binary.
  - 0.1_{10}: This number has an infinite, repeating binary representation: $(0.0001100110011\ldots)_2$. As such, it can only be approximated when stored as a floating-point number. This approximation introduces a small round-off error, which might be in the order of 2×10^{-5} for single-precision or much smaller for double-precision, but it is always present.

1.6.2 Types of Numerical Errors

1. **Round-off Error**:
   - **Definition**: Errors arising from the inability to represent real numbers exactly with a finite number of bits. Occurs during storage, arithmetic operations, and conversion between data types.
   - **Example**: As noted, 0.1_{10} cannot be precisely stored. Repeated addition of such numbers can accumulate these small errors, leading to a significant deviation from the mathematically exact result. For instance, summing 0.1 one hundred times might not yield exactly 10.0 in floating-point arithmetic.

- **Loss of Trailing Digits**: A related issue occurs when very similar numbers are subtracted, or very different magnitude numbers are added. If we have $X - Y$ where $X \approx Y$, the most significant digits cancel out, leaving only the less significant (and potentially error-prone) digits. Similarly, adding a very small number to a very large number might result in the small number being "swallowed" by the large number's magnitude if the combined precision is insufficient.
    - Example: $(1000.0)_{10} + (1.456)_{10}$ in a system with only 4 significant decimal digits would result in 1001.0, effectively losing the 0.456 part.
- **Mitigation**: For addition of many small numbers, using Kahan summation algorithm or summing from smallest to largest can improve accuracy. For subtraction, reformulation of the equation might be necessary (e.g., using Taylor series approximations or alternative algebraic forms).

2. **Overflow and Underflow**:
    - **Definition**: These errors occur when a numerical calculation produces a result that is outside the representable range of the data type.
    - **Overflow**: The result is too large to be stored.
        - Example: Multiplying two very large numbers that individually fit into a `double` but whose product exceeds `double`'s maximum value (10^{308}).
    - **Underflow**: The result is too small (too close to zero) to be represented as a normalized floating-point number, and is typically rounded to zero.
        - Example: A `double` cannot represent numbers smaller than approximately 10^{-308}. If a calculation yields 10^{-500}, it would underflow to zero.
    - **Impact**: Underflow can be particularly problematic in calculations involving ratios or exponents, as a result becoming zero when it should be a small non-zero value can lead to division by zero or incorrect logical paths.
    - **Example from Physical Phenomena**: Consider the Boltzmann factor $\exp(-E/k_BT)$. For a wide bandgap semiconductor like an oxide ($E_g = 4$ eV) at room temperature ($k_BT \approx 62$ meV as per original text's context), the argument is $4/0.062 \approx 64.5$. So $\exp(-64.5) \approx 10^{-28}$. This value is well within the range of a double-precision float. However, if we consider a material at very low temperature, say $T = 3$ Kelvin, $k_BT \approx 0.26$ meV. For a band gap of 1.1 eV (e.g., silicon), $E/k_BT = 1.1/0.00026 \approx 4230$. The Boltzmann factor would be $\exp(-4230) \approx 10^{-1837}$. This value is far smaller than the smallest representable

number for a 64-bit floating point (10^{-308}), leading to an underflow error where the result would be computed as zero. Such scenarios necessitate the use of specialized arbitrary-precision arithmetic libraries.

3. **Truncation Error**:
    - **Definition**: Errors introduced when an infinite mathematical process (like an infinite series, integral, or continuous derivative) is approximated by a finite one.
    - **Example**: Approximating a function using a Taylor series expansion involves summing only a finite number of terms. The neglected higher-order terms constitute the truncation error.
        - $f(x) = \sum_{n=0}^{N} \frac{f^{(n)}(a)}{n!}(x-a)^n + R_N(x)$ The remainder term $R_N(x)$ is the truncation error.
4. **Convergence Error**:
    - **Definition**: Arises in iterative numerical methods (e.g., solving systems of equations, self-consistent field calculations) when the iteration is stopped before the solution has fully converged to its exact value. The difference between the computed approximate solution and the true solution is the convergence error.
5. **Model/Approximation Error**:
    - **Definition**: Errors inherent in the physical or mathematical model itself, due to simplifications or approximations made to render the problem computationally tractable. These are distinct from numerical errors arising from the computation process.
    - **Example**: Using a simplified potential model in a quantum simulation, neglecting certain interactions, or using a classical model when quantum effects are significant.

1.6.3 Practical Implications and Best Practices

1. **Conditional Statements with Floating-Point Numbers**:
    - **Problem**: Due to round-off errors, two floating-point numbers that are mathematically equal might not be precisely equal in computer representation. Therefore, direct equality comparisons (`if (x == y)`) are unreliable and should be avoided.
    - **Solution**: Instead of direct equality, compare if the absolute difference between the two numbers is less than a very small “epsilon” value (a tolerance).
        - `if (fabs(x - y) < EPSILON)`

        - Here, `EPSILON` is a small positive value, e.g., 10^{-9} or 10^{-12}, depending on the required precision and data type (double precision can usually handle smaller epsilons).
    - **Example**: If you expect 3.0×10.0 to be 30.0, an `if (3.0 * 10.0 == 30.0)` statement might sometimes fail because 3.0 or 10.0 might have slight binary representation errors, making their product not exactly 30.0.
2. **Converting Floating-Point to Integer**:
    - **Problem**: When converting a floating-point number to an integer using functions like `int()`, if the floating-point value is slightly less than an integer (e.g., 9.999999999999999), it will be truncated to the lower integer (9 instead of 10).
    - **Solution**: Add a small epsilon before conversion to handle potential slight inaccuracies.
        - `int_value = int(floating_value + EPSILON)`
        - This ensures that values slightly below an integer threshold due to round-off are correctly rounded up before truncation.
3. **Information Buried (Catastrophic Cancellation)**:
    - **Problem**: Occurs when performing subtractions of nearly equal large numbers, leading to a loss of significant digits. This is a severe form of round-off error.
    - **Example**: Calculating $\exp(-x)$ for a large positive x using its Taylor series: $\exp(-x) = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$ If x is large (e.g., $x = 40$), some terms like $x^n/n!$ can become very large positive or negative numbers before the series eventually converges to a very small value. Subtracting these large, nearly equal terms leads to the cancellation of most significant digits, leaving only the less accurate lower-order digits. The computed result can be wildly inaccurate or even have the wrong sign. For $x = 40$, the exact $\exp(-40)$ is $\approx 4.25 \times 10^{-18}$. A direct Taylor series summation might yield a positive number like 5.88, completely wrong.
    - **Solution**: Reformulate the calculation to avoid subtracting nearly equal large numbers. For $\exp(-x)$ when $x > 0$, it is much more stable to calculate $\exp(x)$ and then take its reciprocal: $\exp(-x) = \frac{1}{\exp(x)} = \frac{1}{1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots}$ In this case, all terms in the denominator's series are positive, preventing catastrophic cancellation.

These considerations are fundamental to developing reliable and accurate numerical simulations, especially in fields like semiconductor engineering where precision can directly impact the validity of device models and material predictions.

1.7 Conclusion

Today, we've covered the foundational concepts of numerical representation in computers, including various number bases, data storage units, and the intricacies of integer and floating-point types. More importantly, we've begun to explore the critical topic of numerical errors—their sources, types, and practical implications. Understanding round-off, overflow, underflow, truncation, and loss of significance is paramount for anyone developing computational programs for scientific and engineering applications.

Remember to consider these error sources diligently as you embark on your own programming endeavors. It is not enough for a program to produce an answer; it must produce an accurate and reliable answer within the context of the problem.

For your assignment, please attempt Problem 1 (number base conversion) manually and Problem 2 (Python program analysis) by thoughtfully examining the provided code. These exercises are designed to reinforce today's lecture material and prepare you for more advanced topics in numerical simulation.

Thank you.