Integer and Floating-Point Representation

Algorithms, Programming, and Data Representation · Module 92313 · Chapter 34a

From Fundamentals to AI Apocalypse · personal study guide

Contents

Start from memory 2

How to use this guide 2

How the chapters fit together 3

3.1 Number systems and positional notation 5

3.2 Integer representations 7

3.3 Character data 9

3.4 Unsigned binary arithmetic 10

3.5 Two's-complement arithmetic and overflow 12

4.1 The floating-point model 14

4.2 Binary fractions 17

4.3 Floating-point arithmetic and error 19

4.4 IEEE 754 binary formats 21

4.5 Zero, subnormal numbers, infinity, and NaN 23

5.1 C memory and pointers 28

5.2 The Python memory model 31

Formula map 33

Derive it cold 34

Explain it 35

Mixed practice 36

Exam-style problems 38

Slips in the notes 40

Self-check 41

Answers 42

Made by Thanate Amornjiraporn (Focus)

This chapter builds from bit patterns and fixed-width integer rules to IEEE floating point, shows how finite precision affects calculations, and ends with how C and Python treat memory. Sections 3.x and 4.x keep the numbers of the Chapter 3 and Chapter 4 notes they draw on; sections 5.x cover the memory part of "APDR lecture 2 extended".

Start from memory

Answer before reading. Mark each response sure, unsure, or guessed.

1. Why must a signed bit pattern be interpreted with an encoding and width?
2. What is the n-bit two's-complement range?
3. Which floating-point field controls precision, and which controls range?
4. Why can subtracting close floating-point values produce a poor relative result?
5. What is the main accuracy advantage of fused multiply-add?
6. If p is an int pointer holding 0x1000 and an int takes 4 bytes, what address is p+3?
7. After b = a in Python, with a a list, how many list objects exist?

Answers: see "Start from memory" under Answers at the end.

How to use this guide

Try each question cold before reading the worked explanation. Rebuild the rule, then solve interleaved problems so you practise choosing a method.

1. **Day 0:** Recall §3.1–§3.5 and work the odd practice problems 1–23.
2. **Day 1:** Rebuild §4.1–§4.3 and work the even practice problems 2–24.
3. **Day 3:** Recall §4.4–§4.5 and work practice 25–36.
4. **About day 7:** Rebuild §5.1–§5.2 and work practice 37–59.
5. **About day 14:** Complete all five exam-style problems without notes.
6. **Before the exam:** Tick only skills you can demonstrate cold and revisit missed-answer references.

When stuck: write down the representation, width, and assumptions first. Then show each conversion or arithmetic step and check the result against its range or error bound.

Made by Thanate Amornjirapong (Focus)

How the chapters fit together

Integer representation explains exact bit-level meaning. Floating point extends this to a wide range of approximate real values. Numerical analysis then asks how representation and operation order affect the final answer.

Represent · assign meaning to bits

- **Positional bases** §3.1
  $\sum_{i=0}^{k} d_i b^i$
  Rests on: each digit's place has a power-of-base weight.
- **Integer encodings** §3.2
  bit pattern + width + rule
  Rests on: positional values and a fixed number of bits.

Calculate · preserve exact meaning

- **Fixed-width arithmetic** §3.5
  exact result → range check
  Rests on: the encoding's representable range.
- **Floating point** §4.1
  $(-1)^S \times \text{significand} \times 2^E$
  Rests on: sign, biased exponent, and finite precision.
- **C pointers** §5.1
  new = old + n × sizeof(type)
  Rests on: addresses are integers, and memory is a line of bytes.

Evaluate · bound numerical error

- **Rounding and spacing** §4.3
  $fl(x)=x(1+\delta)$
  Rests on: a finite set of representable values.
- **Numerical stability** §4.5
  reformulate before precision is lost
  Rests on: tracking where rounding error enters and how operations propagate it.
- **Python names** §5.2
  name → object
  Rests on: references, the safe form of the pointer idea.

Made by Thanate Ahornjiraporn (Focus)

Question	Method
What does a bit pattern represent?	State width, encoding, and format.
Is a fixed-width result valid?	Compute exactly, then check its representable range.
Why did a float change?	Inspect spacing, rounding, range, and operation order.
Can cancellation be avoided?	Reformulate the calculation or use one-rounding operations.
What does this pointer or name refer to?	Draw the memory: boxes and addresses in C, names and objects in Python.

Made by Thanate Amornjiraporn (Focus)

3.1 Number systems and positional notation

A numeral is a compact way to write a sum of place values. Binary, octal, decimal, and hexadecimal use the same positional rule with different bases.

> **In plain words:** a numeral is shorthand for a sum: each digit is multiplied by a power of the base, set by its position. In decimal, 352 means $3 \cdot 100 + 5 \cdot 10 + 2 \cdot 1$; in binary the place values are 1, 2, 4, 8, … instead. Octal and hexadecimal are short ways to write long binary strings, because one octal digit is exactly 3 bits and one hexadecimal digit exactly 4.

Expand and convert

In base b, each digit d_i contributes $d_i b^i$, where the rightmost digit has i=0. Binary digits are 0 or 1. Decimal-to-binary repeated division records remainders; read them from last to first.

Positional value

$$(d_k \ldots d_1 d_0)_b = \sum_{i=0}^{k} d_i b^i$$

$$(1100010)_2 = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^1 = 98$$

Read it as: "the number with digits d_k down to d_0 in base b is the sum of d_i times b to the i": each digit times its place value.

The worked binary number is 98 in decimal. Subscripts identify the base.

64	32	16	8	4	2	1
1	1	0	0	0	1	0

$\mathbf{(1100010)_2}$: add the place values above each 1: $64 + 32 + 2 = 98$.

Worked conversion · decimal to binary

Convert 45 to binary by repeated division.

1. $45 \div 2$ gives quotient 22, remainder 1.
2. $22 \div 2$ gives quotient 11, remainder 0; then 11 gives 5 remainder 1.
3. Continue: 5 gives 2 remainder 1; 2 gives 1 remainder 0; 1 gives 0 remainder 1.
4. Read remainders upward: 101101_2. Check: 32+8+4+1=45. §3.1

Octal and hexadecimal

Because $8=2^3$, each octal digit corresponds to three binary digits. Because $16=2^4$, each hexadecimal digit corresponds to four bits. Group binary digits from the right and pad the leftmost group with zeroes if needed. In hexadecimal, A–F mean 10–15.

Compact binary groups

$$10101_2 = 010\,101 = 25_8$$
$$10010100_2 = 1001\,0100 = 94_{16}$$
$$1\,\text{byte} = 8\,\text{bits} = 2\,\text{hexadecimal digits}$$

The lecture uses 16-bit words and 32-bit long words as examples; the sizes named by these terms can depend on a computer architecture.

Worked conversion · hexadecimal to binary and decimal

Convert $2F_{16}$.

1. Replace each digit by four bits: 2 is 0010 and F (15) is 1111, so $2F_{16}=0010\ 1111_2$.
2. Expand by base 16: $2\times16+15=47$.
3. Check by the bits: $32+8+4+2+1=47$. §3.1

Trap: do not read a binary string as an ordinary decimal integer. Attach its base, expand it, and only then compare values.

Near miss: 10_2 looks like ten, but in binary it is $1 \cdot 2 + 0 \cdot 1 = 2$. Read the base before the digits.

Recall check · answer without looking

1. What power of the base multiplies the rightmost digit?
2. Why can binary be grouped in threes to convert to octal?
3. What is A in hexadecimal worth in decimal?

Answers: see "3.1 Number systems and positional notation" under Answers at the end.

Made by

3.2 Integer representations

A computer stores a fixed number of bits, not a plus or minus sign unless the encoding defines one. The same pattern can mean different values under different encodings.

> **In plain words:** a computer stores only a row of bits; whether the row means a positive number, a negative number or something else depends on the agreed encoding. Unsigned uses every bit for size. The signed encodings give part of the range to negatives, and two's complement, the one computers use, gives the top bit a negative weight.

Unsigned and signed ranges

An unsigned n-bit pattern represents 0 through 2^n-1. Signed encodings trade positive range for negative values. The source compares sign-and-magnitude, one's complement, two's complement, excess (biased) notation, and binary-coded decimal (BCD). The lecture judges a signed encoding by four wishes: one zero, a balanced range, simple fast addition hardware, and no gaps in the number line.

Ranges for n bits

$$\begin{aligned}\text{Unsigned}: &\ 0 \dots 2^n - 1\\ \text{Sign and magnitude}: &\ -(2^{n-1}-1) \dots +(2^{n-1}-1)\\ \text{One's complement}: &\ -(2^{n-1}-1) \dots +(2^{n-1}-1)\\ \text{Two's complement}: &\ -2^{n-1} \dots 2^{n-1}-1\end{aligned}$$

Read it as: "with n bits, unsigned runs from 0 to $2^n - 1$; two's complement runs from -2^{n-1} to $2^{n-1} - 1$". Sign-and-magnitude and one's complement have two zero encodings. Two's complement has one zero and one extra negative value.

Worked comparison · decode 1111 in four encodings

1. Unsigned: 15.
2. Sign-and-magnitude: sign bit 1 and magnitude 111, so −7.
3. One's complement: invert to 0000, so −0, which equals zero but has a distinct pattern.
4. Two's complement: the unsigned value 15 represents 15−16=−1. §3.2

Why the older encodings lost

Sign-and-magnitude has two zeros, and its adder must compare signs and then choose whether to add or subtract magnitudes. One's complement still has two zeros, and its addition needs an end-around carry: a carry out of the top bit is added back into the lowest bit. For example, 0101 (+5) + 1110 (−1) gives 1 0011; adding the carry back gives 0100 (+4). Two's complement has one zero and needs no such correction, at the price of a range that is not quite balanced.

Two's complement and bias

To form −x in n-bit two's complement, invert all n bits of x and add 1; discard a carry beyond the width. The minimum value -2^{n-1} is the exception: its positive counterpart is outside the signed range, so fixed-width negation leaves the same pattern and overflows. Why inverting and adding one works: inverting x gives $(2^n-1)-x$, because each bit becomes 1 minus itself. Adding 1 gives 2^n-x, which is exactly the pattern for −x modulo 2^n. Equivalently, interpret a pattern with top bit 1 by subtracting 2^n from its unsigned value. In excess-B notation, actual value = stored unsigned value − B. The lecture's three-bit excess-4 example therefore maps 000 to −4 and 111 to 3. Bias has a practical reason: the order of the true values matches the unsigned order of the stored patterns, so comparing two excess values is a plain unsigned comparison.

Decode rules

$$\text{Two's}-\text{complement value} \;=\; \text{unsigned(pattern)} \;-\; 2^n \text{ when the top bit is 1}$$

$$\text{Negate } x: \text{ invert } n \text{ bits, then add 1}$$

$$\text{Excess}-\text{B value} \;=\; \text{stored value} \;-\; B$$

Read it as: "when the top bit is 1, the two's-complement value is the unsigned value minus 2^n"; to negate, flip every bit and add 1.

Keep the width fixed throughout. The same bit pattern has no signed meaning until an encoding is specified.

−8	4	2	1
1	0	1	1

Four-bit two's complement: the top bit weighs −8, so 1011 = −8 + 2 + 1 = −5.

BCD stores decimal digits separately

In four-bit BCD, each decimal digit has its own four-bit code. For example, decimal 59 is 0101 1001 in BCD, while the ordinary binary value 59 is 00111011. BCD uses more bits but keeps each decimal digit directly readable.

Trap: do not convert all BCD bits as one binary number. Decode each four-bit group as a decimal digit.

Near miss: +8 in four-bit two's complement looks as if it should be 1000, but 1000 is −8: the range stops at +7, so +8 cannot be stored in four bits.

Recall check · answer without looking

1. What is the four-bit two's-complement range?
2. Why is the two's-complement range asymmetric?
3. What is the BCD encoding of decimal 42?

Answers: see "3.2 Integer representations" under Answers at the end.

3.3 Character data

Numbers are not the only data stored as bits. A character encoding assigns numeric codes to letters, digits, punctuation, and control characters.

> **In plain words:** text is stored as numbers too: a character encoding is a table that gives every letter, digit and symbol a numeric code. 7-bit ASCII has 128 codes, enough for basic English; Unicode extends the table to the world's scripts, and a byte encoding says how each code is written as bytes.

ASCII and Unicode

ASCII uses seven bits, giving 128 code values. The lecture describes eight-bit extensions that use the additional 128 patterns, but those upper values depend on the particular extension. Unicode assigns code points to characters across many writing systems; its first 128 code points match ASCII. A code point and its byte encoding are related but are not the same concept.

Character-code facts

$$7 - \text{bit ASCII} : 2^7 = 128 \text{ code values}$$
$$\text{ASCII code points} : 0 \ldots 127$$

Read it as: "7-bit ASCII has $2^7 = 128$ code values, numbered 0 to 127".
The first 128 Unicode code points correspond to ASCII. Byte encodings represent code points as bytes by their own rules.

128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1

'A' in ASCII: 0100 0001 = 64 + 1 = 65.

> **Worked encoding · read a short ASCII string**
> The lecture's example encodes "Fred" in hexadecimal as 46 72 65 64.
>
> 1. Split the bytes: 46, 72, 65, 64.
> 2. Read each as a hexadecimal ASCII code: 46 is F, 72 is r, 65 is e, and 64 is d.
> 3. The four codes decode to "Fred". §3.3

Trap: an encoding is a mapping, not a property of the bit pattern by itself. Identify the character set and any byte encoding before decoding text.

Near miss: the character '7' looks like the number 7, but its ASCII code is 55. Arithmetic on character codes goes wrong unless you subtract the code of '0', which is 48, first.

> **Recall check · answer without looking**
>
> 1. How many code values are available in seven-bit ASCII?
> 2. What is the relation between ASCII and the first 128 Unicode code points?
>
> *Answers: see "3.3 Character data" under Answers at the end.*

3.4 Unsigned binary arithmetic

Binary arithmetic follows the same place-value logic as decimal arithmetic. Carries and borrows move between columns, while the fixed bit width limits which values can be stored.

> **In plain words:** binary arithmetic works column by column like the decimal method from school, only the columns are worth 1, 2, 4, 8, …. 1 + 1 makes 2, which is written 0 with 1 carried. The difference is that a computer has a fixed number of columns, so a carry out of the last column is lost.

Add and subtract

Add each column and carry 1 when a column totals 2. For subtraction, borrow from a higher column when the current minuend bit is too small; a borrowed binary 1 is worth 2 in the current column. Subtraction can also be checked by adding the difference to the subtrahend.

Worked addition · include the carry

Add 111011_2 and 101010_2.

1. From right to left: 1+0=1.
2. 1+1=10, so write 0 and carry 1. Continue each column with its carry.
3. The result is 1100101_2. Check: 59+42=101, and 1100101_2=101. §3.4

Worked subtraction · borrow across a zero

Calculate $1010_2 - 0011_2$.

1. Rightmost column: 0−1 needs a borrow from the next column, so 2−1=1, and that column's 1 becomes 0.
2. Next column: 0−1 needs a borrow, but the column to its left is 0, so take from the top bit. The top bit becomes 0, the third column keeps 1, and this column gets 2: 2−1=1.
3. Third column: 1−0=1. Top column: 0−0=0.
4. The result is 0111_2. Check: 10−3=7. §3.4

Made by

Multiply and divide

Binary multiplication uses partial products, like decimal long multiplication, but each multiplier bit is only 0 or 1. Binary division uses quotient and remainder. For divisor $d \neq 0$, dividend = quotient×divisor + remainder, where $0 \leq \text{remainder} < \text{divisor}$.

Division check

$$\text{dividend} = \text{quotient} \times \text{divisor} + \text{remainder}$$
$$0 \leq \text{remainder} < \text{divisor}$$

Read it as: "dividend equals quotient times divisor plus remainder, with the remainder at least 0 and smaller than the divisor".

Substitute the quotient, divisor, and remainder to check a division result.

Worked multiplication · partial products

Calculate $1101_2 \times 101_2$.

1. Multiplier bit 1 (rightmost): write 1101.
2. Multiplier bit 0: the partial product is zero.
3. Multiplier bit 1 (two places left): write 110100.
4. Add: $1101+110100=1000001_2$. Check: $13 \times 5 = 65$. §3.4

Worked division · long division

Calculate $10111_2 \div 11_2$.

1. 10 is smaller than 11, so take 101: quotient bit 1, remainder $101-11=10$.
2. Bring down 1 to get 101: quotient bit 1, remainder 10.
3. Bring down 1 to get 101: quotient bit 1, remainder 10.
4. Quotient 111_2, remainder 10_2. Check: $7 \times 3+2=23$. §3.4

Memory hook: in binary addition, 1+1 writes 0 and carries 1. In subtraction, a borrow makes the current column worth 2 before subtracting.

Near miss: in four bits, 1100 + 0101 gives the correct sum 10001, but only four bits are kept: 0001. The column arithmetic was right; the result just does not fit.

Recall check · answer without looking

1. What bit do you write for 1+1, and what do you carry?
2. What inequality must the remainder satisfy?

Answers: see "3.4 Unsigned binary arithmetic" under Answers at the end.

3.5 Two's-complement arithmetic and overflow

Two's complement lets the same adder perform signed addition and subtraction. Correct bit operations still need a range check: a fixed-width result can wrap to a pattern with a different signed value.

> **In plain words:** two's complement lets one adder handle both signs: to subtract, add the negation. The catch is the fixed range. Adding two positive numbers can go past the largest value and wrap round to a pattern that reads as negative. That is **overflow**, and it is not the same thing as a carry out of the top bit.

Addition, subtraction, and overflow

Add the bit patterns and discard any carry beyond the most significant bit. For subtraction, negate the subtrahend and add it to the minuend. Signed overflow occurs when the exact mathematical result lies outside the representation's range. For addition, it occurs when two operands with the same sign produce a result with the opposite sign; operands of unlike signs cannot overflow.

n-bit two's-complement checks

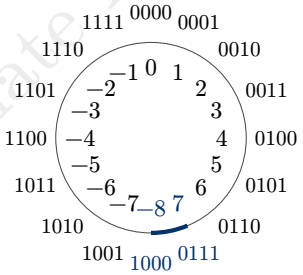
$$\text{Range}: -2^{n-1} \ldots 2^{n-1} - 1$$

$$A - B = A + (\text{two's} - \text{complement negation of } B)$$

$$\text{Same} - \text{sign addition with opposite} - \text{sign result} \Rightarrow \text{overflow}$$

Read it as: "A minus B is A plus the negation of B"; overflow on addition means two operands of the same sign gave a result of the other sign.

Discarding carry-out is part of fixed-width addition. Carry-out alone does not signal signed overflow.



The 4-bit two's-complement wheel: adding 1 to 0111 (7) steps clockwise to 1000, which reads as −8. That jump at the bottom of the wheel is overflow.

> **Worked example · detect overflow**
>
> Use four-bit two's complement to calculate $-7+(-6)$.
>
> 1. Encode −7 as 1001 and −6 as 1010.
> 2. Add: 1001+1010=1 0011. Discard the carry-out, leaving 0011.
> 3. The stored result 0011 is +3, but the exact sum is −13, outside the four-bit range −8…+7. Both inputs were negative and the result is positive, so overflow occurred. §3.5

Made by Thanate A… (Focus)

Multiplication and division

The lecture notes that ordinary unsigned multiplication cannot simply be applied to signed two's-complement patterns and read at the original width. A full product may require up to twice as many bits as the operands; retain a sufficient width and interpret the result consistently. For a simple signed multiplication, multiply magnitudes and make the result negative exactly when the operand signs differ, then check the available width. The supplied slides introduce this idea but do not give a complete signed division algorithm.

Worked multiplication · signed values

Calculate $(-3)\times(+5)$ and store it in eight-bit two's complement.

1. Multiply magnitudes: $3\times5=15=00001111_2$.
2. The signs differ, so the result is negative.
3. Negate in eight bits: invert to 11110000 and add 1, giving 11110001.
4. Check: $241-256=-15$, inside $-128\ldots+127$. Four bits could not hold it. §3.5

Overflow also matters for safety. The lecture warns that an integer overflow in a memory-size calculation can become a security hole, such as a buffer overflow, because too little memory gets allocated.

Sign check: do not treat discarded carry as overflow. Decode the fixed-width result and compare it with the exact mathematical result and allowed range.

Near miss: in four bits, 0111 + 0001 = 1000 has no carry out of the top bit, yet it is wrong: 7 + 1 = 8 is out of range, and 1000 reads as −8. Overflow is judged by the signs, not by the carry.

Recall check · answer without looking

1. How is A−B implemented in two's-complement arithmetic?
2. Can two's-complement addition overflow when the operands have unlike signs?
3. Why can a signed product need more bits than either input?

Answers: see "3.5 Two's-complement arithmetic and overflow" under Answers at the end.

Made by

4.1 The floating-point model

Fixed-width integers from §3.5 cannot hold fractions or very large and very small values. Floating point uses a sign, a significand, and an exponent to cover those magnitudes, at the cost of exactness.

> **In plain words:** a floating-point number is scientific notation in binary: a sign, a short run of significant digits (the **significand**) and an **exponent** that slides the point left or right. A fixed number of bits holds only so many significant digits, so most real numbers are stored as the nearest value the format can hold, and the gaps between neighbouring values grow as the numbers get larger.

Made by Thanate Amornjiraporn (Focus)

Range and precision

For base β, a normalised number has the form sign × significand × β^{exponent}. The exponent controls scale and therefore the approximate range. The number of significand bits controls precision, or how finely nearby values can be distinguished.

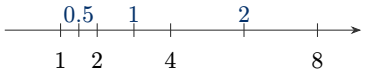
Floating-point values are finite in number, and their spacing changes with magnitude.

Scientific notation model

$$x = s \times M \times \beta^{E}$$

Read it as: "x is s times M times β to the E": the sign s, the significand M with the digits, and the exponent E that says where the point goes.

s is the sign, M is the significand, β is the base or radix, and E is the exponent. In binary floating point, β=2.



Gaps double with 2 significant bits: the values 1, 1.5, then 2, 3, then 4, 6, 8 lie 0.5, then 1, then 2 apart (the numbers above each zone).

Local gap between neighbours

$$\text{gap within } [2^{E},\ 2^{E+1}) \ = \ 2^{E-p+1}$$

Read it as: "the gap between neighbouring values in $[2^{E}, 2^{E+1})$ is 2^{E-p+1}": it doubles every time the values cross a power of 2.

p is the number of significand bits. (This follows directly from the notes' model: the last of p bits has weight $2^{-(p-1)}$, scaled by 2^{E}. The notes only say the gap grows.) Between 8 and 16, E=3, so p=24 gives 2^{-20}.

Worked example · normalise a decimal value

Write 343000×10^{-4} in normalised decimal form with $1 \le M < 10$.

1. Move the decimal point in 343000 five places left to make 3.43.
2. Increase the exponent by five to preserve the value: 3.43×10^{1}.
3. Check: $343000 \times 10^{-4} = 34.3$, and $3.43 \times 10^{1} = 34.3$. §4.1

Trap: moving the radix point changes the exponent in the opposite direction if the represented value is to stay the same.

Near miss: 12.5×10^{0} and 1.25×10^{1} are the same number, but only the second is normalised, with one non-zero digit before the point. A format that allowed both would spend bit patterns on duplicate values.

Recall check · answer without looking

1. Which field chiefly determines range?
2. Which field chiefly determines precision?
3. Why are floating-point values not evenly spaced over the number line?

Answers: see "4.1 The floating-point model" under Answers at the end.

Made by Thanate Amornjiraporn (Focus)

4.2 Binary fractions

The significand from §4.1 is a binary fraction, so its place values explain both conversion and approximation.

> **In plain words:** binary fractions work like decimal ones with halves instead of tenths: after the binary point the places are worth 1/2, 1/4, 1/8, …. Some fractions that are short in decimal never end in binary, just as 1/3 never ends in decimal.

The first bit after the binary point has weight 2^{-1}, and each following bit has half the previous weight.

Convert binary fractions to decimal

Binary fractional place values

$$0.b_1b_2...b_k = b_1 2^{-1} + b_2 2^{-2} + ... + b_k 2^{-k}$$

Read it as: "$0.b_1b_2…b_k$ is b_1 halves plus b_2 quarters plus …": each bit times its power of one half.
Each b_i is 0 or 1. A leading integer part is evaluated with non-negative powers of 2.

	1/2	1/4	1/8
0.	1	0	1

$\mathbf{0.101_2} = 1/2 + 1/8 = 0.625$: each place after the point is worth half the one before.

Worked conversion · binary fraction to decimal
Convert 0.01101_2 to decimal.

1. Use the five place values: 1/2, 1/4, 1/8, 1/16, and 1/32.
2. Keep the places where the bit is 1: 1/4 + 1/8 + 1/32.
3. Add: 0.25+0.125+0.03125=0.40625. §4.2

Made by Tha… rnjrapo… s)

Convert decimal fractions to binary

Repeatedly multiply the fractional part by 2.

At each step, the integer part is the next binary digit, and the new fractional part is carried into the next step.

Worked conversion · terminating binary fraction

Convert 0.6875 to binary.

1. $0.6875 \times 2 = 1.375$, so the first bit is 1; carry 0.375.
2. $0.375 \times 2 = 0.75$, so the next bit is 0; carry 0.75.
3. $0.75 \times 2 = 1.5$, so the next bit is 1; carry 0.5.
4. $0.5 \times 2 = 1.0$, so the next bit is 1; the fraction is now zero.
5. Therefore $0.6875 = 0.1011_2$. Check: $1/2 + 1/8 + 1/16 = 0.6875$. §4.2

Some fractions, such as 0.1 in decimal, repeat in binary.

A finite significand stores only an approximation of a repeating expansion.

Trap: a decimal fraction that looks short is not necessarily exact in binary.

Near miss: 0.1 has a single digit in decimal, but in binary it is 0.000110011… with the block 0011 repeating forever, so a computer can only store an approximation. $0.5 = 0.1_2$, by contrast, is exact.

Recall check · answer without looking

1. What is the weight of the fourth binary digit after the point?
2. What is 0.101_2 in decimal?
3. Why can a computer store 0.1 only approximately in binary floating point?

Answers: see "4.2 Binary fractions" under Answers at the end.

Made by T

4.3 Floating-point arithmetic and error

The exponent and significand must be managed during each operation.

> **In plain words:** to multiply two floating-point numbers, multiply the significands and add the exponents. To add them, first shift the one with the smaller exponent until the exponents match, then add. Either way the result is normalised and rounded, and that rounding is where small errors creep in.

After the exact operation, the result may need normalisation and rounding to fit the available precision.

Multiplication

Multiply normalised values

$$(M_1 \times \beta^{E_1})(M_2 \times \beta^{E_2}) = (M_1 M_2) \times \beta^{E_1+E_2}$$

Multiply the significands and add the exponents. Then normalise and round if needed.

Worked multiplication · normalise the result

Calculate $(2.6\times10^{3})(5.4\times10^{-2})$.

1. Multiply significands: $2.6\times5.4=14.04$.
2. Add exponents: $3+(-2)=1$, giving 14.04×10^{1}.
3. Normalise: $1.404\times10^{2}=140.4$. §4.3

Addition

Before adding, express both numbers with the same exponent.

Shift the significand of the value with the smaller exponent to the right by the exponent difference.

This alignment can discard low-order digits before the addition is even performed.

Worked addition · align exponents

Use two significant decimal digits to add 4.5×10^{3} and 6.7×10^{2}.

1. Rewrite the second value at exponent 3: $6.7\times10^{2}=0.67\times10^{3}$.
2. Add significands: $(4.5+0.67)\times10^{3}=5.17\times10^{3}$.
3. Round to two significant digits: 5.2×10^{3}. The exact sum is 5170, while this two-digit result is 5200. §4.3

Rounding, overflow, and comparison

Truncation drops extra digits and tends to bias results in one direction.

Rounding chooses a nearby representable value and usually gives a better approximation.

Exponent overflow occurs when the result is too large for the format.

Exponent underflow occurs when the result is too small for the normalised range.

The slides describe infinity or an exception for overflow, and zero or an exception for underflow in their simplified model.

IEEE formats also provide subnormal numbers near zero.

For approximate comparisons, choose a tolerance ε that fits the scale and precision of the problem.

Absolute tolerance example

$$\text{Treat } a \text{ as close to } b \text{ when } |a - b| < \varepsilon$$

$$\text{For } b = 1 \text{ and } \varepsilon = 0.000005 : 0.999995 < a < 1.000005$$

Read it as: "treat a as close to b when the distance between them is less than ϵ".
A fixed absolute tolerance may be unsuitable when magnitudes vary greatly. The slides also introduce relative closeness.

Sign check: estimate the order of magnitude before accepting a rounded result.

Near miss: 0.1 + 0.2 looks as if it should equal 0.3, but in double precision the computed sum is 0.30000000000000004, so the test 0.1 + 0.2 == 0.3 is false. Compare floating-point results with a tolerance instead.

Recall check · answer without looking

1. What do you do to the exponents when multiplying?
2. Which operand's significand is shifted to align an addition?
3. How do overflow and underflow differ?
4. Why can direct equality comparison be unreliable for computed floating-point values?

Answers: see "4.3 Floating-point arithmetic and error" under Answers at the end.

4.4 IEEE 754 binary formats

A shared format defines how floating-point fields are stored and how arithmetic behaves.

> **In plain words:** IEEE 754 is the shared recipe for storing a floating-point number: one sign bit, an exponent field stored with an offset (the **bias**) so it needs no sign of its own, and the fraction bits after the binary point. A normalised number always starts with 1., so that 1 is not stored at all: it is the **hidden bit**.

The slides introduce IEEE 754 as a widely adopted standard for predictable binary floating-point arithmetic.

Single and double precision

Format	Total	Sign	Exponent	Fraction field	Significand precision	Bias
Single	32 bits	1 bit	8 bits	23 bits	24 bits for normal values	127
Double	64 bits	1 bit	11 bits	52 bits	53 bits for normal values	1023

For a normalised binary value, the leading significand bit is 1 and is omitted from the stored fraction field.

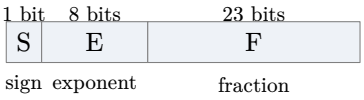
The exponent field stores a biased exponent rather than a two's-complement exponent. Stored order then matches true order, so hardware can compare two exponents as plain unsigned integers.

Normalised value rule

$$\text{value} = (-1)^S \times 1.F \times 2^{E-\text{bias}}$$
$$\text{Single bias} = 127$$
$$\text{double bias} = 1023$$

Read it as: "value is −1 to the S, times 1.F, times 2 to the (E minus bias)": the sign, the stored fraction with its hidden 1, and the exponent once the bias is removed.

S is the sign bit, E is the unsigned exponent field, and F is the stored fraction interpreted after the binary point. This rule is for normalised values.



IEEE 754 single precision: 1 sign bit, 8 exponent bits stored with bias 127, and 23 fraction bits after the hidden 1, 32 bits in all.

Convert decimal to single precision

Worked conversion · 42.6875

1. Convert to binary: $42.6875 = 101010.1011_2$.
2. Normalise: 1.010101011×2^5, so the sign bit is 0.
3. Store the fraction after the hidden leading 1: 01010101100000000000000.
4. Bias the exponent: $5 + 127 = 132 = 10000100_2$.
5. The fields are 0 | 10000100 | 01010101100000000000000, or hexadecimal 422A C000. §4.4

Decode single precision

Worked decoding · BEC0 0000

1. The sign bit is 1, so the value is negative.
2. The exponent field is $01111101_2 = 125$; subtract the bias: $125 - 127 = -2$.
3. Restore the hidden bit: $1.1000\ldots_2 = 1.5$.
4. Scale: $1.5 \times 2^{-2} = 0.375$. Apply the sign: -0.375. §4.4

Trap: restore the hidden leading 1 when decoding a normalised value, but do not apply that rule to a subnormal value.

Near miss: the single-precision exponent field 10000000 looks like a large exponent, but it means 128 − 127 = 1. Subtract the bias before reading the exponent.

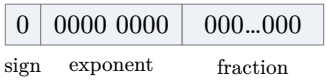
Recall check · answer without looking

1. How many bits are stored in the single-precision fraction field?
2. What is the single-precision exponent bias?
3. Why can a normalised value have 24 bits of significand precision while storing only 23 fraction bits?

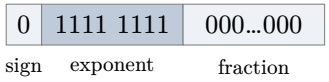
Answers: see "4.4 IEEE 754 binary formats" under Answers at the end.

4.5 Zero, subnormal numbers, infinity, and NaN

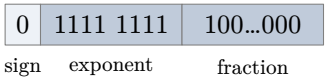
Reserved exponent patterns extend useful behaviour beyond normalised values.



+0: every bit is 0.



$+\infty$: exponent all 1s, fraction 0.



NaN: exponent all 1s, fraction not 0.

> **In plain words:** some exponent patterns are kept for special cases. An exponent of all zeros means zero or a tiny **subnormal** number that fills the gap just above zero; an exponent of all ones means infinity when the fraction is zero, and NaN, "not a number", for results such as 0/0.

In single precision, exponent field zero and exponent field all ones have special interpretations.

Exponent field	Fraction field	Meaning
All zeroes	All zeroes	Positive or negative zero
All zeroes	Non-zero	Subnormal number
Neither all zeroes nor all ones	Any pattern	Normalised number
All ones	All zeroes	Positive or negative infinity
All ones	Non-zero	NaN, or not-a-number

Subnormal value rule

$$\text{value} = (-1)^S \times 0.F \times 2^{-126}$$

$$\text{Smallest normal single}: 2^{-126} \approx 1.175 \times 10^{-38}$$

Read it as: "a subnormal value is −1 to the S times 0.F times 2^{-126}": no hidden 1, and the exponent stays at its smallest value.

For a subnormal single-precision number, the hidden bit is 0 and the exponent is fixed at −126. Precision falls as the value nears zero.

Subnormal values fill the gap between the smallest normal value and zero, so underflow is gradual. They also keep the property x−y==0 exactly when x==y, even for very small numbers.

They can be slow: many CPUs handle them through a microcode assist or a software trap, which the lecture calls orders of magnitude slower. Flush-to-zero (FTZ) is a non-compliant mode that treats subnormals as zero. It is faster but gives up that correctness, and high-performance programs often enable it.

Infinity represents overflow results and some divisions by zero. NaN represents an operation without a real-number result, such as 0/0, $\sqrt{-1}$ or infinity−infinity.

Before IEEE 754, such operations could trap and end the program. With these values the computation continues, and the status propagates to be checked later.

Operations on infinity and NaN follow format rules, not ordinary real-number algebra in every case.

Special arithmetic examples

Operation	Result
$N \div \pm\infty$	0
$\pm$ non-zero $\div 0$	$\pm\infty$
$+\infty + +\infty$	$+\infty$
$\infty + 5.0$	∞
$\pm\infty \times \pm\infty$	$\pm\infty$
$\pm 0 \div \pm 0$	NaN
$+\infty - +\infty$	NaN
$\pm\infty \div \pm\infty$	NaN
$\pm\infty \times 0$	NaN

The sign of the infinity in signed cases depends on the operand signs. An invalid operation normally returns a quiet NaN; a trap may be enabled instead.

Worked classification · single precision fields

Classify sign 0, exponent 11111111, fraction 000…000.

1. The exponent field is all ones.
2. The fraction field is all zeroes.
3. This encodes positive infinity. §4.5

Trap: an all-ones exponent does not always mean infinity; inspect the fraction field to distinguish infinity from NaN.

Numerical stability and cancellation

Cancellation occurs when subtracting close stored values. The subtraction may itself be exact, yet earlier rounding in the operands can dominate the small difference. A stable method avoids needlessly creating that sensitive subtraction.

Made by Thabate Academy (Focus)

Machine epsilon and relative error

Machine epsilon is the gap from 1 to the next larger representable value. For radix β and p significant digits, $\varepsilon_{mach}=\beta^{1-p}$. Here p counts the full significand digits, including the hidden leading bit in a normalised IEEE value.

Binary machine epsilon

$$\text{Single precision}: p = 24, \varepsilon_{\text{mach}} = 2^{-23} \approx 1.19 \times 10^{-7}$$
$$\text{Double precision}: p = 53, \varepsilon_{\text{mach}} = 2^{-52} \approx 2.22 \times 10^{-16}$$

Read it as: "ϵ_{mach} is the gap between 1 and the next larger value the format can store": 2^{-23} in single precision. Under round-to-nearest, the unit roundoff u is $\varepsilon_{mach}/2$ for normal results. "Unit roundoff" is not a term from the slides.

Relative rounding model

$$\text{fl}(x) = x(1+\delta), \text{ with } |\delta| \le u$$

Read it as: "the stored value fl(x) is x times (1 + δ), with |δ| at most u": rounding changes a number by a small relative amount.
For finite non-zero normal results, round-to-nearest, and no overflow. Relative error is measured against |x|, so the same absolute error matters more when the true answer is small.

The slides bound the error by ε; under round-to-nearest the tight bound is ε/2 (= u). Both statements are true; use the slide form if the exam asks for it.

Error propagation in a difference

$$x' = x(1+\delta_x),\ y' = y(1+\delta_y)$$
$$\text{relative error} = ((x'-y')-(x-y))/(x-y) = (x\delta_x - y\delta_y)/(x-y)$$

The numerator stays about the size of the input errors. As x approaches y the denominator approaches zero, so the relative error can explode.

Worked example · cancellation

Suppose five significant decimal digits store x=0.123456789 as 0.12346 and y=0.123451234 as 0.12345.

1. The exact difference is 0.000005555.
2. The stored difference is 0.00001.
3. The stored difference is about 1.80 times the exact difference, so its relative error is about 80.0%. The subtraction exposed input rounding error. §4.5

Sterbenz lemma

If x and y are positive finite floating-point values in the same radix and precision, and $y/2 \le x \le 2y$, then x−y is exactly representable when gradual underflow is supported. For example, 1.5 and 1.0 meet the condition, and their difference 0.5 is exact. This does not undo rounding that happened while x or y was computed; cancellation can expose that earlier error.

Stable quadratic roots

For $ax^2+bx+c=0$, require $a\neq 0$ and $D=b^2-4ac\geq 0$. The usual formula can subtract nearly equal quantities for one root. For $D>0$, choose the branch that adds magnitudes: if $b\geq 0$, set $q=-\frac{1}{2}(b+\sqrt{D})$; if $b<0$, set $q=-\frac{1}{2}(b-\sqrt{D})$. Then the roots are q/a and c/q. This branch also handles $b=0$. For $D=0$, use the repeated root $-b/(2a)$, since q may be zero. These steps avoid cancellation while respecting the degenerate cases.

The notes write the same method as $x_1=(-b-\mathrm{sgn}(b)\sqrt{D})/(2a)$ and $x_2=c/(a\,x_1)$. Since $q=a\cdot x_1$, c/q is the slide's $c/(a\,x_1)$. The second form comes from expanding $a(x-x_1)(x-x_2)$: matching the constant term gives $x_1x_2=c/a$, which is Vieta's relation, so x_2 needs no subtraction.

Worked example · $x^2+10^8x+1=0$ in double precision

1. Here $a=1$, $b=10^8$, $c=1$, so $D=10^{16}-4$ and $\sqrt{D}\approx 10^8$.
2. Direct formula for the small root: $(-b+\sqrt{D})/2$ subtracts nearly equal values and gives about -7.45×10^{-9}.
3. Stable: $b\geq 0$, so $q=-\frac{1}{2}(b+\sqrt{D})\approx -10^8$. The roots are $q/a\approx -10^8$ and $c/q\approx -10^{-8}$.
4. Check: the product of the roots is $1=c/a$, and the sum is about $-10^8=-b/a$. The direct value was about 25% off. §4.5

Trap: mathematical equivalence does not guarantee equal numerical accuracy. Look for subtraction of nearly equal values and avoid rounding intermediate results unnecessarily.

Made by Thanate Amornpat

Fused multiply-add

A fused multiply-add evaluates a×b+c with one final rounding. Separate multiplication and addition can round the product first, then round again after addition. FMA can therefore preserve more information; its speed benefit depends on hardware. The lecture names dot products, matrix multiplication and polynomial evaluation as algorithms that gain from it.

Fused versus separate rounding

$$\mathrm{FMA}(a, b, c) = \mathrm{round}(a \times b + c)$$
$$\text{Separate} : \mathrm{round}(\mathrm{round}(a \times b) + c)$$

The fused operation avoids the intermediate rounding step.

Near miss: $+0$ and -0 have different bit patterns and compare as equal, yet they are not interchangeable: $1/(+0)$ gives $+\infty$ while $1/(-0)$ gives $-\infty$. The sign bit still matters at zero.

Recall check · answer without looking

1. How do you distinguish infinity from NaN?
2. What is the main purpose of subnormal numbers?
3. What does zero exponent and zero fraction encode?
4. What does machine epsilon measure, and how is it related to the unit roundoff?
5. Which quadratic root should you compute second, and how?

Answers: see "4.5 Zero, subnormal numbers, infinity, and NaN" under Answers at the end.

Made by Thanate

5.1 C memory and pointers

Everything so far gave meaning to bit patterns. C also lets you handle where those patterns live, because an address is itself a number you can store. That gives direct control and direct ways to fail.

> **In plain words:** every variable lives at an address in memory, and in C that address is a value you can store: a **pointer**. & gives the address of a variable, and * follows an address to the value stored there. That gives direct control over memory, and direct ways to break it.

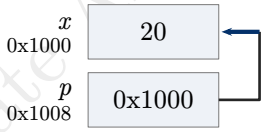
Addresses, & and *

C treats memory as a contiguous line of bytes, each with a numerical address, usually written in hexadecimal. A pointer is a variable that stores an address. The operator & gives the address of a variable, and * reads or writes the value at an address.

Core syntax

$$\text{int } * \text{p; declares a pointer to int}$$
$$\text{int x } = \text{ 10; p } = \text{ \&x; p now holds the address of x}$$
$$*p \;=\; 20$$
$$\text{writes through } p\text{, so } x \text{ becomes } 20$$

*Read it as: "int *p declares p as a pointer to an int"; "p = &x puts the address of x into p"; "*p = 20 writes 20 wherever p points".*
In a declaration, * builds a pointer type. In an expression, * dereferences.



A pointer: p stores the address of x, so *p = 20 writes into x. The addresses are illustrative.

Made by Thanate Ang (Focus)

Pointer arithmetic and arrays

Adding n to a pointer moves n elements, not n bytes. In most expressions an array name decays to a pointer to its first element, and subscripting is defined through that pointer.

Scaling and subscripts

$$\text{new address} = \text{old address} + \text{n} \times \text{sizeof(type)}$$
$$a[i] \text{ means } *(a + i)$$

*Read it as: "adding n to a pointer moves it n elements, that is n × sizeof(type) bytes, not n bytes"; a[i] is the same as *(a + i).*
Sizes depend on the platform; the lecture assumes 4 bytes for int and 8 for double.

Worked example · scaled steps
With int a[10]; int *p = &a[0]; the array starts at 0x1000, and sizeof(int) is 4.

1. p+1 is 0x1000+1×4=0x1004, the address of a[1].
2. p+3 is 0x1000+3×4=0x100C, because 12 is C in hexadecimal. This is the address of a[3].
3. For struct Point { double x; double y; } each element takes 2×8=16 bytes, so the same +1 step moves 16 bytes. Check: the step always equals the element size. §5.1

A function that receives an array actually receives a copy of the pointer to its first element. That is why arrays appear to be passed by reference.

Double pointers and function pointers

C passes every argument by value, so a function given a pointer gets a copy and cannot change the caller's pointer. To let it do so, as an allocating function must, pass the address of the pointer. A pointer to a pointer adds one level of indirection: with int **p2 = &p1; *p2 is p1 and **p2 is the int.

Changing the caller's pointer

$$\text{void allocate(int } **\text{ptr, int size) } \{ *\text{ptr} = \text{malloc(size} * \text{sizeof(int))}; \}$$
$$\text{int } * \text{ my} = \text{NULL; allocate(\&my, 10);}$$

Inside the function, *ptr is the caller's my itself.

Function pointers

$$\text{int } (*\text{op})(\text{int, int}) = \text{add; result} = (*\text{op})(5, 3);$$

A pointer can hold the address of code. Passing one to another function gives a callback, as in qsort or event handlers. With add returning a+b, result is 8.

Pitfalls

Problem	What goes wrong	Habit
Uninitialised pointer	Holds a garbage address; writing through it is undefined behaviour.	Initialise: p = NULL or p = &some_variable.
NULL dereference	Undefined behaviour; a crash such as a segmentation fault is typical.	Test p != NULL first.
Memory leak	The only pointer to a malloc block is lost, so it can never be freed.	Pair each malloc with a free.
Dangling pointer	The pointer still holds the address of freed memory, which may be reused.	free(p); p = NULL;

Trap: setting p to NULL after free only protects that one pointer. Any other copy of the address still dangles.

Near miss: after int *p = malloc(sizeof(int)); free(p); the pointer p still holds the old address, and *p may even still show the old value, but reading it is undefined behaviour: that memory no longer belongs to you.

Recall check · answer without looking

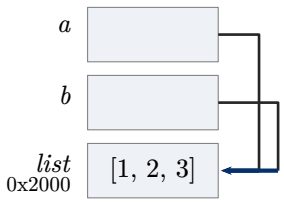
1. If q is a double pointer value 0x2000 with 8-byte double, what is q+2?
2. Why must an allocating function take int ** rather than int *?
3. What is a dangling pointer?

Answers: see "5.1 C memory and pointers" under Answers at the end.

Made by Thanat

5.2 The Python memory model

Python keeps the reference idea from §5.1 but hides the addresses and the freeing. Knowing the model tells you when a change through one name shows up through another.



Two names, one list: after b = a, both names refer to the same object, so a += [3] changes what b sees too. The address is illustrative.

> **In plain words:** in Python a variable is a name tag tied to an object, not a box that holds a value. b = a ties a second tag to the same object. If the object can change, as a list can, a change made through one name shows up through the other; if it cannot, as a number or a string cannot, "changing" it really ties the name to a new object.

Names, objects and rebinding

A C variable is a typed box that holds a value. A Python variable is a name bound to an object that lives elsewhere, and all data, including integers and functions, are objects. The built-in id() gives an object's identifier, which in CPython is its memory address, so you can see when two names share an object. In C, x = 20 overwrites the box. In Python, x = 20 rebinds the name to another object and leaves the old object unchanged.

> **Worked trace · shared reference then rebinding**
> Trace a = [1, 2, 3]; b = a; b.append(4); b = [0]; print(a).
>
> 1. a = [1, 2, 3] makes one list object and binds a to it.
> 2. b = a binds a second name to the same object, so id(a) equals id(b).
> 3. b.append(4) changes that object, so a now shows [1, 2, 3, 4].
> 4. b = [0] rebinds b only. print(a) gives [1, 2, 3, 4]. Check: exactly one statement changed the object a refers to. §5.2

Mutability and function arguments

Immutable objects (int, float, str, tuple) cannot change after creation, so operations make new objects. Mutable objects (list, dict, set) can change in place, and the change is visible through every name that refers to them.

A function receives a reference to the object, which is called pass-by-object-reference. It cannot change what the caller's name refers to, but it can change a mutable object's contents. With data = [1, 2, 3], a function calling some_list.append(4) leaves data as [1, 2, 3, 4].

Automatic memory management

Reference counting is the main mechanism: each object counts the references to it, and when the count reaches zero it is freed at once. These references include list slots and other containers, not only names. A generational garbage collector runs periodically to find reference cycles, such as two objects that refer to each other, which counting alone cannot free.

Feature	C	Python
A variable is	A typed memory location	A name bound to an object
Assignment	Copies a value into the location	Binds a name to an object
Memory management	Manual: malloc and free	Automatic: garbage collection
Pointer arithmetic	Scaled by type size	Not applicable
Main risk	Undefined behaviour	Unintended side effects through shared objects
Priority	Direct control and performance	Safety and developer productivity

Trap: b = a never copies a list. To get a separate list you must ask for one, for example with list(a).

Trap: id() is unique only among objects alive at the same time, so a freed object's identifier can come back.

Near miss: a = [1, 2]; b = a; a = a + [3] leaves b as [1, 2], but a = [1, 2]; b = a; a += [3] makes b [1, 2, 3] too. For lists, + builds a new list, while += changes the existing one in place.

Recall check · answer without looking

1. After x = 10 then x = 20, what happened to the object 10?
2. Why can a function change a list it receives but not the caller's integer?
3. What can the generational collector free that reference counting cannot?

Answers: see "5.2 The Python memory model" under Answers at the end.

Made by

Formula map

Idea	Relation	Use
Positional value	$\sum_{i=0}^{k} d_i b^i$	Decode a numeral.
Unsigned n-bit range	$0 \ldots 2^n-1$	Check fit.
Two's-complement range	$-2^{n-1} \ldots 2^{n-1}-1$	Check signed fit.
Biased exponent	actual exponent = stored exponent − bias	Decode a normal IEEE value.
Floating-point model	$(-1)^S \times \text{significand} \times 2^E$	Interpret sign, scale, and precision.
Machine epsilon	β^{1-p}	Spacing from 1 to the next value; p is the total number of significand digits.
Unit roundoff	$u=\varepsilon_{mach}/2$	Bound relative rounding error for normal results under round-to-nearest.
Relative error model	$fl(x)=x(1+\delta)$, $\lvert\delta\rvert\le u$	Model rounding under stated assumptions.
Sterbenz condition	$y/2 \le x \le 2y$	Subtraction of such floating-point x and y is exact under the lemma's assumptions.
Local gap	2^{E-p+1}	Spacing within $[2^E, 2^{E+1})$.
Error of a difference	$(x\delta_x-y\delta_y)/(x-y)$	See how cancellation amplifies input error.
Pointer step	new = old + n × sizeof(type)	Find an address, in bytes.
Subscript	a[i] = *(a + i)	Move between array and pointer notation.

Check every answer

State width and encoding for integer problems

Track signs, exponent alignment, and rounding

Check limiting cases and exact ranges

Integer problems have no physical units. In floating-point problems, report the numerical precision and assumptions used. Pointer sizes are in bytes on a stated platform.

Made by Mr. Mrinjapeyi (Sam)

Derive it cold

Cover the explanations and rebuild each result on blank paper.

1. **Base conversion:** expand a binary numeral into powers of 2. §3.1
2. **Binary to octal/hex:** derive why groups of 3/4 bits match one digit. §3.1
3. **Two's-complement negation:** show why invert-and-add-one gives $2^n - x$. §3.2
4. **Signed range:** determine the lowest and highest n-bit two's-complement values. §3.2
5. **Unsigned division:** verify a quotient by reconstructing the dividend. §3.4
6. **Addition overflow:** explain why same-sign inputs can produce an invalid opposite-sign result. §3.5
7. **Subtraction:** replace A−B with A+(−B) and show the bit width at every step. §3.5
8. **Binary fractions:** derive each bit's place value as a negative power of 2. §4.2
9. **Decimal fraction conversion:** use repeated multiplication by 2 and track the fractional remainder. §4.2
10. **Multiplication:** use exponent laws, then normalise. §4.3
11. **Addition:** align exponents, combine significands, and round. §4.3
12. **Single-precision decoding:** recover the sign, unbiased exponent, and hidden bit. §4.4
13. **Special values:** derive the classification from exponent and fraction fields. §4.5
14. **Cancellation and stable roots:** explain how input error is amplified and derive the product relation for the second quadratic root. §4.5
15. **One-rounding FMA:** compare fused evaluation with separately rounded multiply and add. §4.5
16. **Machine epsilon:** derive the gap near 1 from radix and significand precision. §4.5
17. **Sterbenz lemma:** check the ratio condition and distinguish exact subtraction from prior input rounding. §4.5
18. **Pointer scaling:** derive the address of a[i] from the base address and sizeof. §5.1
19. **Reference trace:** draw names and objects for a shared list, then a rebinding. §5.2

Made by Thanate Angkunjiraporn (Focus)

Explain it

Answer each why or when aloud before opening the model response.

1. Why does the rightmost binary digit have weight 1?
2. Why are octal and hexadecimal convenient ways to write binary?
3. Why do sign-and-magnitude and one's complement have two zero patterns?
4. Why does two's complement have one extra negative value?
5. When is carry-out discarded, and why is that not enough to detect signed overflow?
6. Why can BCD be easier for decimal display but less space-efficient?
7. Why does increasing the exponent change range more than precision?
8. Why can two nearby real values have different spacing when stored at different magnitudes?
9. Why does 0.1 decimal not have a finite binary fraction?
10. Why must exponents be aligned before adding two floating-point values?
11. Why can rounding during addition lose a small addend?
12. Why is the leading 1 omitted for normalised binary significands?
13. When would a subnormal number be used?
14. Why should an approximate comparison use a tolerance chosen for the problem?
15. Why is machine epsilon a relative-precision scale rather than one fixed absolute error for every number?
16. When can Sterbenz's lemma say the subtraction itself is exact, and what error can still remain?
17. Why do floating-point formats store the exponent in excess notation?
18. Why can a C function not change the caller's pointer when it is given that pointer?
19. Why does appending inside a Python function affect the caller's list, while assigning to the parameter does not?
20. Why does Python need a cycle collector as well as reference counts?

Answers: see "Explain it" under Answers at the end.

Made by Thomas Amorinjitapom (Focus)

Mixed practice

Try all 59 problems before opening the worked answers. Interleaving makes you choose the representation or method each time.

1. Convert 37_{10} to binary and check by place value.
2. Convert 0.1101_2 to decimal.
3. Decode 1110 as a four-bit two's-complement number.
4. How many bits of significand precision does normalised IEEE single precision provide?
5. How many bits represent one hexadecimal digit?
6. Normalise 0.000101_2 as $M \times 2^E$ with $1 \le M < 2$.
7. Encode decimal 73 in four-bit BCD.
8. Convert 0.375 decimal to binary.
9. How many values are available in seven-bit ASCII?
10. In a floating-point format, which field chiefly determines range?
11. Add 1101_2 and 0111_2 as unsigned values.
12. Multiply $(3\times10^2)(2\times10^{-4})$ and give a normalised result.
13. What is the four-bit two's-complement range?
14. Align 7.2×10^5 to exponent 7.
15. Convert 101101_2 to octal.
16. How many stored fraction bits does double precision have?
17. In excess-4, what value does the three-bit pattern 110 represent?
18. Classify single-precision fields S=1, E=all ones, F=0.
19. For $47 \div 6$, give quotient and remainder and verify the division identity.
20. Convert 0.10011_2 to decimal.
21. Negate 0011 in four-bit two's complement.
22. Use two significant decimal digits to round $8.46\times10^3 + 2.1\times10^2$.
23. Convert 11101010_2 to hexadecimal.
24. What is the single-precision exponent bias?
25. Why does two's-complement addition discard a carry-out?
26. Convert decimal 0.625 to binary.
27. Write decimal 59 in BCD and ordinary binary.
28. For normalised single precision with exponent field E=130, what is the unbiased exponent?
29. Compute $7-(-6)$ using four-bit two's complement. State the exact result and whether it fits.
30. State one reason truncation can be less desirable than rounding.
31. What do the first 128 Unicode code points have in common with ASCII?
32. Classify single-precision fields E=0 and F non-zero.

Made by Thanate Amornjiraporn (Focus)

33. Multiply 101_2 by 11_2 as unsigned values.
34. For 1.25×2^3, give the decimal value.
35. In four-bit two's complement, add 0110 and 0011. Does it overflow?
36. How many total bits are used in double precision?
37. What is the decoded value of pattern 0111 in four-bit excess-8?
38. Give an absolute-tolerance test for a value a to be close to b using ε.
39. What is one reason BCD can be useful despite its extra storage?
40. Convert 1.101_2 to decimal.
41. Why is 1.25×2^3 an exact binary value?
42. What do $\pm$non-zero $\div$ 0 and $\pm0 \div \pm0$ produce?
43. What happens to the exponent when multiplying two floating-point values?
44. When the exponent field is all ones and the fraction is zero, what is the value when S=0?
45. For binary single precision with 24 significand bits, give machine epsilon and unit roundoff.
46. Convert 86_{10} to binary and verify it by place value.
47. Check Sterbenz's condition for x=3.0 and y=2.0, then state their exact difference.
48. Does 0.2 terminate in binary? Why or why not?
49. An int pointer p holds 0x2000 and sizeof(int) is 4. What address is p+5?
50. In Python, after a = [1, 2]; b = a; b.append(3), what does print(a) show?
51. Rewrite a[2] using pointer arithmetic.
52. In Python, after x = 10; y = x; x = 20, what is y? Why?
53. Take x=100 with δ_x=+0.001 and y=99 with δ_y=−0.001. Find the relative error of x−y.
54. A C function is void clear(int *q) { q = NULL; }. Does the caller's pointer change? What signature would change it?
55. In Python, def f(s): s = s + [1]. After t = [0]; f(t), what is t?
56. What is wrong with int *p = malloc(sizeof(int)); free(p); *p = 1;?
57. Two Python objects refer only to each other, and no name reaches them. Does reference counting free them? What does?
58. Add 0110 and 1101 in four-bit one's complement, using the end-around carry. Decode the result.
59. A single-precision result is 2^{-127}. What is it with gradual underflow, and what under flush-to-zero?

Answers: see "Mixed practice" under Answers at the end.

Made by Thanate Angunjaporn (Focus)

Exam-style problems

Allow about 20 minutes for each problem. Attempt each part before opening its worked answer.

Problem 1 · Compare encodings

Encode -5 in four-bit sign-and-magnitude, one's complement and two's complement.

1. Write the four-bit pattern in each encoding.
2. State the range and number of zero patterns for each representation.
3. Explain why two's complement is useful for arithmetic despite its asymmetric range.

Answers: see "Exam-style problems" under Answers at the end.

Problem 2 · Convert, calculate, and check

In four-bit two's complement, compute $-5 + 3$.

1. Write both input patterns and add them, showing any carry-out.
2. Decode the retained four-bit result and compare it with the exact sum.
3. State whether overflow occurred and explain why carry-out is not the test.

Answers: see "Exam-style problems" under Answers at the end.

Problem 3 · Encode and decode single precision

Store 13.25 in IEEE single precision.

1. Convert 13.25 to binary.
2. Normalise it and identify sign, exponent, and stored fraction.
3. Give the exponent field and verify by decoding.

Answers: see "Exam-style problems" under Answers at the end.

Problem 4 · Align, add, and assess precision

Add 3.4×10^4 and 5.6×10^2 using two significant decimal digits.

1. Align both values to exponent 4.
2. Add the significands and round to two significant digits.
3. Compare the rounded result with the exact decimal sum and explain the source of the difference.
4. Find the relative error and compare it with the unit roundoff of this two-digit decimal format.

Answers: see "Exam-style problems" under Answers at the end.

Made by Thanate Angoritiraporn (Focus)

Problem 5 · From addresses to names

In C, int a[4] starts at 0x2000 and sizeof(int) is 4, with int *p = a;. In Python, def f(s): s.append(1); s = [] is called as t = [0]; f(t).

1. Give the address p+3 in hexadecimal, and two other ways to write the element there.
2. A C function sets its int * parameter to NULL. Does p change? Give the signature that would change it.
3. State what print(t) shows and which line of f affects it.
4. Compare how the C and Python functions reach the caller's data.

Answers: see "Exam-style problems" under Answers at the end.

Made by Thanate Amornjiraporn (Focus)

Slips in the notes

Entries were checked against the slides, some only as extracted text, so confirm any entry against the slide itself before relying on it. "Slide" means a slide of "APDR lecture 2 extended" unless stated. Where the guide differs from the notes, this list says so; items marked as notation or simplification are not errors of value.

- **Chapter 3 slide 24:** the four-bit Excess-8 row for 0111 is printed as 1; stored value 7 minus bias 8 gives −1. §3.2
- **Slide 3 against slide 5 (uneven claim):** a balanced range is listed as desirable and only the older encodings are said to fail. Two's complement also misses it, as slide 5 itself shows. §3.2
- **Slide 6:** the C example says assigning 127+1 to signed char wraps to −128. That is not guaranteed by C: integer promotion computes 128 as int, then converting an out-of-range value to signed char has an implementation-defined result or signal. Do not generalise this example as signed integer overflow being defined wraparound. See clauses 6.3.1.1 and 6.3.1.3 of the WG14 C standard draft N1570. §3.5
- **Slide 17 (simplification):** the performance cost of subnormals depends on the CPU and its mode. The guide reports it as the lecture's statement. §4.5
- **Slides 18–19:** ε is called the upper bound on relative rounding error. That holds, but under round-to-nearest the tight bound is $\varepsilon/2$, which the guide calls u. Slide 19's "any real number x that can be represented" means any x within range. §4.5
- **Slide 21:** Sterbenz's lemma is stated without its conditions. The guide adds positive values of the same format, and gradual underflow. §4.5
- **Slide 22:** x_1 with sgn(b) breaks at b=0 if sgn(0)=0, because it gives $x_1=0$ and then $c/(a\cdot 0)$. The slide also leaves $a\neq 0$ and $D\geq 0$ unstated. The guide's branch treats b=0 as $b\geq 0$ and requires both. §4.5
- **Slide 26:** "int arr;" declares one int, yet the text calls it an array and labels p+1 and p+3 both "address of arr". The brackets were probably lost; the guide reads int a[10], a[1] and a[3]. §5.1
- **Slide 27:** "int int_array;" and "struct Point pt_array;" have probably lost their sizes in the same way. "sizeof(Point)" needs "struct Point" in C, and the sizes 4 and 16 assume a platform. §5.1
- **Slide 28:** "array is equivalent to array" is a tautology. It reads as if it should be array equivalent to &array[0]. §5.1
- **Slide 32:** dereferencing NULL is said to cause a crash always. It is undefined behaviour; a crash is only typical. The fix printed as "p = some ariable" should be p = &some_variable. §5.1
- **Slides 35 and 38:** "a =", "data =" and the printed outputs have lost their list literals. The guide supplies [1, 2, 3], which is an assumption, chosen to fit the append(4). §5.2
- **Slide 39 (simplification):** the count is described as names that refer to an object. It counts all references, including list slots. §5.2

Wordsby Thomas Amorniiraporn (11 cms)

Self-check

Tick a skill only in a later session, at least one day after studying, when you can demonstrate it without notes.

- ☐ Expand numbers by powers of their base and convert binary groups to octal and hexadecimal. (1, 5, 15, 23, 46)
- ☐ State signed and unsigned ranges from a bit width, and decode patterns in each signed encoding, including excess-B. (3, 13, 17, 37)
- ☐ Negate a two's-complement value at fixed width, and subtract by negating and adding. (21, 29)
- ☐ Write and read decimal digits in BCD, and relate ASCII and Unicode code points. (7, 9, 27, 31, 39)
- ☐ Add binary values with correct carries, and verify a quotient and remainder. (11, 19, 33)
- ☐ Add signed values and detect overflow. (25, 35, 58)
- ☐ Explain how significand and exponent set precision and range, and normalise a value without changing its magnitude. (6, 10)
- ☐ Convert fractions between binary and decimal, and explain why some decimal fractions repeat in binary. (2, 8, 20, 26, 40, 48)
- ☐ Multiply floating-point numbers, align exponents before adding, and round to a stated precision. (12, 14, 22, 30)
- ☐ State an approximate comparison using a tolerance. (38)
- ☐ State the single-precision field sizes and bias, and encode and decode a normalised value. (4, 24, 28, exam 3)
- ☐ Classify subnormal and special-value patterns and predict basic infinity and NaN arithmetic. (18, 32, 42, 44, 59)
- ☐ Calculate machine epsilon and unit roundoff, and apply Sterbenz's condition. (45, 47)
- ☐ Trace pointers, scaled addresses and common C memory errors. (49, 51, 54, 56, exam 5)
- ☐ Trace Python names, rebinding, mutability and reclamation. (50, 52, 55, 57, exam 5)
- ☐ Rebuild the derivations from the formula list.
- ☐ Complete exam problems 1 to 5 with checks.

Made by Thanate Angkunjiaporn (Focus)

Answers

Start from memory

1. The same bits can represent different values under different widths and encoding rules. §3.2
2. -2^{n-1} through $2^{n-1}-1$. §3.2
3. The significand field controls precision; the exponent field chiefly controls range. §4.1
4. Earlier rounding errors become large compared with a small difference. §4.5
5. It rounds the product-plus-sum once, avoiding an intermediate rounding. §4.5
6. 0x100C, because the step is scaled by the element size. §5.1
7. One. Both names refer to the same object. §5.2

3.1 Number systems and positional notation

1. The zeroth power, which is 1. §3.1
2. Each octal digit represents 0–7, exactly the range of three bits, and $8=2^3$. §3.1
3. 10. §3.1

3.2 Integer representations

1. -8 through $+7$. §3.2
2. There is one zero and one additional negative pattern, 1000, representing -8. §3.2
3. 0100 0010, one four-bit code for 4 and one for 2. §3.2

3.3 Character data

1. $2^7=128$. §3.3
2. The Unicode code points 0–127 correspond to the ASCII characters; their byte representation depends on the encoding used. §3.3

3.4 Unsigned binary arithmetic

1. Write 0 and carry 1 to the next column. §3.4
2. For a positive divisor, $0 \le \text{remainder} < \text{divisor}$. §3.4

Made by Thanate Amornjiraporn (Focus)

3.5 Two's-complement arithmetic and overflow

1. Negate B in the chosen width, then add it to A and discard carry-out. §3.5
2. No. Overflow in addition requires two operands with the same sign and a result of the opposite sign. §3.5
3. The product magnitude can exceed the range of either input width; a full-width product may need up to twice as many bits. §3.5

4.1 The floating-point model

1. The exponent field. §4.1
2. The significand field. §4.1
3. Each exponent scales the significand by a different power of the base, so the gap between representable values grows with magnitude. §4.1

4.2 Binary fractions

1. $2^{-4}=1/16$. §4.2
2. $1/2+1/8=0.625$. §4.2
3. The binary expansion repeats, but the significand has a finite number of bits and must be rounded or truncated. §4.2

4.3 Floating-point arithmetic and error

1. Add them. §4.3
2. Shift the significand belonging to the smaller exponent to the right. §4.3
3. Overflow exceeds the maximum magnitude; underflow falls below the smallest normalised magnitude. §4.3
4. Finite precision can make a computed result differ slightly from its exact mathematical value, so a suitable tolerance may be needed. §4.3

4.4 IEEE 754 binary formats

1. Twenty-three. §4.4
2. 127. §4.4
3. The leading binary 1 is implicit, giving 24 significant bits including the 23 stored fraction bits. §4.4

Made by Thanate Amornjiraporn (Focus)

4.5 Zero, subnormal numbers, infinity, and NaN

1. Both use an all-ones exponent. A zero fraction means infinity; a non-zero fraction means NaN. §4.5
2. They represent values between zero and the smallest normalised magnitude, making the transition toward zero gradual. §4.5
3. Positive or negative zero, according to the sign bit. §4.5
4. The gap from 1 to the next larger value; u is half of it under round-to-nearest. §4.5
5. The one that would suffer cancellation. Compute it from the product: $x_2 = c/(a\ x_1)$. §4.5

5.1 C memory and pointers

1. 0x2010, since $2 \times 8 = 16 = 0x10$. §5.1
2. Arguments are copied, so changing an int * parameter changes only the copy. Through int ** the function reaches the caller's pointer. §5.1
3. One that still holds the address of memory that has been freed. §5.1

5.2 The Python memory model

1. Nothing changed in it. The name x now refers to another object. §5.2
2. A list is mutable, so its contents change in place. An integer is immutable, and rebinding the parameter affects only the local name. §5.2
3. Reference cycles, whose counts never reach zero. §5.2

Made by Thanate Amornjiraporn (Focus)

Explain it

1. The rightmost place is position zero, and any base raised to zero is 1. Its bit contributes either zero or one to the value.
2. Each octal digit maps to exactly three bits and each hexadecimal digit to four. Grouping bits preserves their value while shortening the written form.
3. Both encodings use a sign bit and a separate magnitude/complement pattern for zero. That creates distinct positive-zero and negative-zero bit patterns.
4. With n bits, one pattern is used for zero and the remaining patterns cannot be divided evenly into equal positive and negative ranges. Two's complement assigns the extra pattern to the negative side.
5. Fixed-width addition retains only n result bits, so carry beyond the word is discarded. Signed overflow depends on the signs of the inputs and result, not on that discarded bit.
6. Each four-bit BCD group can be read as one decimal digit, but four bits are used per decimal digit even when ordinary binary could represent the whole number with fewer bits.
7. The exponent scales a significand by powers of the base, expanding which magnitudes can be represented. Precision depends chiefly on how many significand bits distinguish values at a given scale.
8. At a larger exponent, the same significand step is multiplied by a larger power of 2. Therefore adjacent representable values are farther apart.
9. A finite binary fraction has a denominator that is a power of 2 after reduction. One tenth has a factor of 5 in its denominator, so its binary expansion repeats.
10. Only values with the same exponent have significands describing the same scale. Alignment expresses both at a shared scale before combining them.
11. Alignment shifts the smaller significand to the right. If the exponent difference is large, its significant bits may fall beyond the available precision.
12. Every normalised binary number begins with 1 before the point, so the format can infer that bit and use its storage for another fraction bit.
13. A subnormal represents a non-zero value smaller than the minimum normalised magnitude, close to zero.
14. Absolute or relative error that is acceptable depends on the scale and purpose of the calculation. One fixed tolerance may be too strict for large values or too loose for small ones.
15. Spacing scales with the exponent, so a fixed absolute difference is a different fraction of values at different magnitudes. Machine epsilon measures spacing near 1; the relative rounding model expresses error as a fraction of the input magnitude.
16. For positive floating-point x and y with $y/2 \le x \le 2y$, their difference is exactly representable under the format assumptions. Any rounding that already changed x or y can still affect the difference.
17. Stored order then matches true order. Exponents can be compared as plain unsigned integers, which is simple and fast in hardware.
18. C copies every argument, so the function holds its own copy of the address. Changing that copy leaves the caller's pointer alone, which is why it must receive the pointer's address instead.
19. Both names refer to the same object, so append changes what the caller sees. Assignment only rebinds the local name to another object.
20. Two objects that refer to each other keep each other's count above zero even when nothing else can reach them. Counting alone would never free them, so a periodic collector finds such cycles.

Made by Thabang Amonjiraporn (Focus)

Mixed practice

1. 37=32+4+1, so 100101_2. §3.1
2. 1/2+1/4+1/16=0.8125. §4.2
3. Unsigned value 14−16=−2. §3.2
4. 24 bits: the 23 stored fraction bits plus the implicit leading 1. §4.4
5. Four bits, since $16=2^4$. §3.1
6. $1.01_2\times2^{-4}$. The point moves four places right to make the significand 1.01, so the exponent is −4. §4.1
7. 7 is 0111 and 3 is 0011, so 73 is 0111 0011 in BCD. §3.2
8. $0.375=3/8=0.011_2$. §4.2
9. $2^7=128$ code values. §3.3
10. The exponent field. §4.1
11. 1101+0111=10100. As an unsigned five-bit result it is 20, which matches 13+7. §3.4
12. Multiply significands: 6. Add exponents: −2. Result 6×10^{-2}, already normalised. §4.3
13. −8 through +7. §3.2
14. $7.2\times10^5=0.072\times10^7$; shift the significand two places right as the exponent rises by two. §4.3
15. Group 101 101; each group is 5, so 55_8. §3.1
16. 52 stored fraction bits. §4.4
17. Pattern 110 is 6; 6−4=2. §3.2
18. Negative infinity: all-ones exponent and zero fraction mean infinity; sign bit 1 makes it negative. §4.5
19. 47=7×6+5, so quotient 7 and remainder 5; 0≤5<6. §3.4
20. 1/2+1/16+1/32=0.59375. §4.2
21. Invert 0011 to 1100 and add 1: 1101, which is −3. §3.2
22. Align $2.1\times10^2=0.21\times10^3$. Sum 8.67×10^3; to two significant digits this is 8.7×10^3. §4.3
23. 1110 1010 maps to E A, so EA_{16}. §3.1
24. 127. §4.4
25. The machine keeps only the fixed n result bits. Carry-out is outside that word and does not alone indicate signed overflow. §3.5
26. $0.625=1/2+1/8=0.101_2$. §4.2
27. BCD: 0101 1001. Ordinary binary: 111011. §3.2
28. 130−127=3. §4.4
29. −6 is 1010; invert and add 1 to get +6=0110. Then 0111+0110=1101, which decodes as −3. The exact result is +13, outside −8…+7, so overflow occurs. §3.5
30. Truncation always discards in the same direction for positive values and can introduce a biased error. §4.3
31. Unicode code points 0–127 correspond to ASCII characters. §3.3

Made by Nhanate Amornjraporn (Focus)

32. A subnormal number. The exponent is zero and the fraction is non-zero. §4.5

33. 101 is 5 and 11 is 3; $5\times3=15=1111_2$. §3.4

34. $1.25\times8=10$. §4.1

35. 0110+0011=1001, which decodes as −7. The exact result is +9, outside the four-bit range −8…+7; the two positive inputs produced a negative-sign pattern, so it overflows. §3.5

36. 64 bits: one sign, eleven exponent, and 52 fraction bits. §4.4

37. $7-8=-1$. §3.2

38. For example, treat a as close to b when $|a-b|<\varepsilon$, with ε chosen for the expected scale and precision. §4.3

39. BCD keeps each decimal digit directly encoded, which can simplify decimal display or input/output. §3.2

40. $1+1/2+1/8=1.625$. §4.2

41. $1.25=1.01_2$ is a finite binary fraction, and multiplying by 2^3 only shifts the binary point. §4.2

42. $\pm\infty$ and NaN, respectively; the first divides a non-zero value by zero, while $0\div0$ has no real result. §4.5

43. Add the exponents and multiply the significands, then normalise if needed. §4.3

44. Positive infinity. §4.5

45. For $\beta=2$ and $p=24$, $\varepsilon_{mach}=2^{-23}\approx1.19\times10^{-7}$. Under round-to-nearest, $u=\varepsilon_{mach}/2=2^{-24}\approx5.96\times10^{-8}$. §4.5

46. $86=64+16+4+2=1010110_2$. §3.1

47. $y/2=1\le3\le4=2y$, so the condition holds. The difference $3.0-2.0=1.0$ is exact. §4.5

48. No. $0.2=1/5$, and the factor 5 means the denominator is not a power of 2. Repeated doubling gives $0.0011\ 0011\ldots_2$, which repeats. §4.2

49. $0x2000+5\times4=0x2014$, since $20=0x14$. §5.1

1, 2, 3 , because a and b name one list. §5.2

50. *(a + 2). §5.1

51. 10. y was bound to the object 10, and x = 20 rebinds only x. §5.2

52. The numerator is $100\times0.001-99\times(-0.001)=0.199$ and $x-y=1$, so the relative error is 0.199, or 19.9%. Input errors of 0.1% grew about 200-fold. §4.5

53. No, q is a copy. Use void clear(int **q) { *q = NULL; } and call clear(&p). §5.1

0 . s + [1] builds a new list and rebinds the local name, so the caller's list is untouched. §5.2

54. After free, p dangles, so writing through it is undefined behaviour. §5.1

55. No, their counts stay at 1. The generational garbage collector detects the cycle and frees them. §5.2

56. 0110+1101=1 0011. Add the carry back: 0011+1=0100, which is +4. This matches 6+(−2). §3.2

57. Gradual underflow keeps it as a subnormal: $0.1_2\times2^{-126}$. Flush-to-zero turns it into 0. §4.5

Made by Thanate Ariornjiraporn (Focus)

Exam-style problems

1. +5 is 0101. Sign-and-magnitude is 1101. One's complement is 1010, the inversion of 0101. Two's complement is 1011, the inversion plus 1. §3.2
2. For four bits, sign-and-magnitude and one's complement both range from −7 to +7 and have two zero patterns. Two's complement ranges from −8 to +7 and has one zero. §3.2
3. Negation and subtraction use the same fixed-width adder: invert the subtrahend and add one. Its one extra negative value is usually a small trade-off for simpler arithmetic hardware. §3.2 §3.5

Exam-style problems

1. −5 is 1011 and +3 is 0011. Addition gives 1110 with no carry-out. §3.5
2. 1110 is 14 unsigned; subtract 16 to decode −2. The exact sum is also −2. §3.2 §3.5
3. No overflow: the exact result −2 lies in −8…+7. Carry-out is not the signed-overflow criterion; compare the signs or the exact result to the range. §3.5

Exam-style problems

1. $13.25{=}1101.01_2$, since $13{=}1101_2$ and $0.25{=}0.01_2$. §4.2
2. Normalise: $1.10101{\times}2^3$. The sign is 0. The stored fraction is 10101000000000000000000 after padding to 23 bits. §4.4
3. Biased exponent is $3{+}127{=}130{=}10000010_2$. The pattern is 0 | 10000010 | 10101000000000000000000. Decoding restores $1.10101_2{\times}2^3{=}13.25$. §4.4

Exam-style problems

1. $5.6{\times}10^2{=}0.056{\times}10^4$. §4.3
2. $(3.4{+}0.056){\times}10^4{=}3.456{\times}10^4$, which rounds to $3.5{\times}10^4$. §4.3
3. The exact sum is 34560 and the rounded result is 35000. The smaller addend contributes digits beyond the two-digit precision, so rounding changes the stored result. §4.3
4. The relative error is $440/34560{\approx}1.27\%$. With $\beta{=}10$ and $p{=}2$, $\varepsilon{=}10^{-1}$ and $u{=}0.05$, so the error is within the 5% bound. §4.5

Exam-style problems

1. 0x2000+3×4=0x200C. The element is a[3], or *(p + 3). §3.1 §5.1
2. No, the parameter is a copy. Use int ** and pass &p. §5.1

0, 1 . The append changes the shared list; s = [] only rebinds the local name. §5.2

3. C copies an address, and the function writes through it with *. Python copies a reference, and the function changes a mutable object. Neither can rebind the caller's own variable through a plain parameter. §5.1 §5.2

Personal study guide based on the supplied APDR Chapter 3 and Chapter 4 lecture notes and "APDR lecture 2 extended". Written in my own words; no lecture pages or figures are reproduced. Not an official FAU publication; check course details against the current notes. Made by **Thanate Amornjiraporn (Focus)**.

Made by Thanate Amornjiraporn (Focus)