Number and Data Representation

Algorithms, Programming, and Data Representation · Module 92313 · Chapter 3

From Fundamentals to AI Apocalypse · personal study guide

Contents

Start from memory 2

How to use this guide 2

How the chapter fits together 3

3.1 Number systems and positional notation 4

3.2 Integer representations 6

3.3 Character data 9

3.4 Unsigned binary arithmetic 11

3.5 Two's-complement arithmetic and overflow 14

Formula map 17

Derive it cold 17

Explain it 18

Mixed practice 19

Exam-style problems 20

Slips in the notes 21

Self-check 21

Answers 22

Made by Thanate Amornjiraporn (Focus)

This chapter shows how bit patterns encode numbers and characters, then uses those representations to perform arithmetic. The meaning of a bit pattern depends on the encoding and the number of bits.

Start from memory

Answer before reading. Mark each response sure, unsure, or guessed; uncertainty shows what to revisit.

1. What does the position of a digit mean in a base-b number?
2. What is the signed range of an n-bit two's-complement value?
3. How do you negate a two's-complement bit pattern?
4. How many binary digits correspond to one hexadecimal digit?
5. What does a discarded carry-out tell you about overflow?

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

How to use this guide

Study one section, hide it, and reconstruct the rules and examples from memory. Then solve mixed questions before checking the worked answers.

1. **Day 0:** Try the cold start and study §3.1. Convert values both ways without a calculator.
2. **Day 1:** Rebuild the integer-encoding table in §3.2; solve practice 1–8.
3. **Day 3:** Recall the character encodings in §3.3 and work through unsigned arithmetic in §3.4.
4. **About day 7:** Re-derive two's-complement addition and subtraction in §3.5; solve practice 9–28.
5. **About day 14:** Complete the three exam-style problems without notes and read the slips list.
6. **Before the exam:** Tick only skills you can demonstrate cold; revisit sections named in missed answers.

When stuck: write the bit width and encoding first. Show each conversion or arithmetic step, then check the answer against the allowed range.

Made for Thanate Amornjraporn (Focus)

How the chapter fits together

First learn how positional notation creates values. Then learn the encoding rules that assign values to bit patterns; finally apply arithmetic rules and check the fixed-width result.

Represent · map information to bits

- **Number bases** §3.1

  digits × powers of the base

  Rests on: positional notation, with the rightmost place worth one.

- **Integer encodings** §3.2

  one bit pattern, several interpretations

  Rests on: a fixed width and an explicit signed or unsigned encoding.

Encode · store other data

- **Character codes** §3.3

  character → code point → encoded bits

  Rests on: agreed mappings between symbols and numeric codes.

- **Unsigned arithmetic** §3.4

  carry · borrow · partial products

  Rests on: positional values and column-wise binary operations.

Compute · preserve meaning

- **Signed arithmetic** §3.5

  fixed width → decode → check overflow

  Rests on: two's-complement interpretation and its representable range.

Question	Useful method
What value does this bit pattern mean?	State width and encoding, then decode.
How do I convert a base-b number?	Expand by powers or use repeated division.
Did signed arithmetic overflow?	Check the exact result against the fixed-width range.

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

$(\mathbf{1100010})_\mathbf{2}$: 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. (This goes beyond the notes, which give only those examples.)

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 Thanate

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).

Ranges for n bits

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

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

Made by (Focus)

What a good encoding should offer

The lecture lists the properties an integer encoding should have: one bit pattern per value, equal numbers of positive and negative values, the largest possible range, no gaps in the range, and fast, cheap arithmetic hardware. No encoding here has all of them, as this comparison for n bits shows.

Encoding	Patterns for zero	Equal positive and negative?	Arithmetic hardware
Sign and magnitude	Two	Yes	Costly: compares signs and needs a subtractor
One's complement	Two	Yes	Needs carry-out adjustments
Two's complement	One	No, one extra negative	Simplest: $X - Y = X + (-Y)$ uses the adder alone
Excess-B	One	Depends on B	Chosen for ordering, covered next

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. 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.

Excess exists to keep numbers in order. In two's complement a negative value starts with 1, so comparing patterns as unsigned numbers ranks every negative value above every positive one. In excess notation the all-zeros pattern is the smallest value and the all-ones pattern the largest, so the unsigned comparison gives the right order.

Decode rules

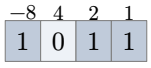
$$\text{Two's}-\text{complement value} = \text{unsigned(pattern)} - 2^n \text{ when the top bit is 1}$$

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

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

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.



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.

A BCD digit usually takes four or eight bits. With four, only 0000–1001 are used; the six patterns 1010–1111 are not valid digits. The lecture's number 9876510 needs 7×4=28 bits in BCD but 24 bits in ordinary binary.

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?
4. Which property does excess notation offer that two's complement lacks?
5. How many four-bit patterns are unused in BCD?

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

Made by Thanate Ang

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.

Made by Thanate Amornjiraporn (Focus)

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. (This distinction goes beyond the notes, which do not discuss byte encodings.)

The 128 ASCII patterns divide into 52 letters (26 lowercase and 26 uppercase), 10 digits, 32 punctuation marks and 34 others: whitespace such as space, tab and return, plus control characters. The lecture's table reads by position: bits 6–4 pick the column and bits 3–0 the row, so F, 100 0110, sits in column 100 and row 0110. Unicode aims to give every character in any language a number. The lecture says over 120,000 are defined, and the first 65,536 (the 16-bit range) cover the world's major alphabets.

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?
3. How many ASCII patterns are letters, and how many are digits?
4. How many characters does the 16-bit Unicode range hold?

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. In a subtraction, the **minuend** is the number subtracted from and the **subtrahend** is the number being subtracted: in 612−485, 612 is the minuend and 485 the subtrahend. 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 zeros

Subtract 10011_2 from 1000101_2, numbering columns from 0 at the right.

1. Column 0: 1−1=0.
2. Column 1: 0−1 needs a borrow. Column 2 drops from 1 to 0, and column 1 becomes 10, so 10−1=1.
3. Columns 2 and 3: 0−0=0 each.
4. Column 4: 0−1 needs a borrow, but column 5 is 0. Borrow from column 6, which drops to 0. Column 5 becomes 10, lends 1 and keeps 1, and column 4 becomes 10, so 10−1=1.
5. Column 5: 1−0=1. Column 6: 0.
6. The result is 0110010_2. Check: 69−19=50, and 110010_2=50. §3.4

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>0, dividend = quotient×divisor + remainder, where 0≤remainder<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 · a carry of 10

Multiply 1111_2 by 111_2. The partial products are 1111, 11110 and 111100. Add them column by column from the right.

1. Column 0 totals 1: write 1. Column 1 totals 2: write 0, carry 1.
2. Column 2 holds three 1s plus the carry, total 4, or 100_2. Write 0 and carry 10_2, which puts 2 in the next column.
3. Column 3 totals 3+2=5: write 1, carry 2. Column 4 totals 2+2=4: write 0, carry 2.
4. Column 5 totals 1+2=3: write 1, carry 1. Column 6 is the carry: write 1.
5. The result is 1101001_2. Check: 15×7=105=64+32+8+1. §3.4

Worked long division · quotient and remainder

Divide 110101_2 by 101_2.

1. The first three bits, 110, are at least 101: quotient bit 1, leaving 110−101=1.
2. Bring down 1 to get 11, less than 101: quotient bit 0.
3. Bring down 0 to get 110: quotient bit 1, leaving 1.
4. Bring down 1 to get 11: quotient bit 0. No bits remain, so the remainder is 11_2.
5. Quotient 1010_2, remainder 11_2. Check: 53=10×5+3 and 0≤3<5. §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. In multiplication, a column can total more than 3, so a carry can be 10_2.

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?
3. In 9−4, which number is the subtrahend?

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

Made by Thanate Amornjiraporn (Focus)

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.

Made by Thanate Amornjiraporn (Focus)

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

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

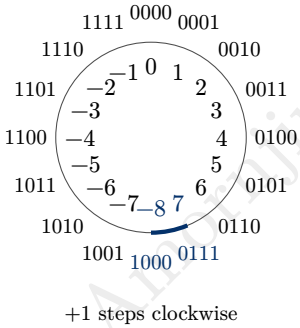
$$A - B = A + (\text{two's} - \text{complement negation of } B)$$

$$\text{Addition overflow}: \text{same} - \text{sign operands, opposite} - \text{sign result}$$

$$\text{Subtraction overflow}: \text{operand signs differ, result has the subtrahend's sign}$$

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

Worked example · carry-out without overflow

Use eight-bit two's complement to calculate −2+(−5).

1. +2 is 00000010, so −2 is 11111101+1=11111110. +5 is 00000101, so −5 is 11111010+1=11111011.
2. Add: 11111110+11111011=1 11111001. Discard the carry-out, leaving 11111001.
3. Decode: 249−256=−7, which is the exact sum and lies in −128…+127.
4. A carry came out, yet there is no overflow: both inputs are negative and so is the result. §3.5

Trap: do not test a subtraction by applying the addition rule to the negated subtrahend. Take $0-(-8)$ in four bits: -8 negates to itself, so the sum is 0000+1000, unlike signs, which seems safe. The exact result $+8$ overflows. The subtraction rule catches it: the operand signs differ and the result 1000 has the subtrahend's sign.

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. The reason: $-Y$ is stored as 2^n-Y, so unsigned multiplication gives $X(2^n-Y)=2^nX-XY$, whereas the correct 2n-bit pattern is $2^{2n}-XY$. Sign-extending both operands to 2n bits and keeping the low 2n bits of the product works, but it is very inefficient. 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 · -3×2 in four bits

1. Take magnitudes: 3 is 0011 and 2 is 0010.
2. Multiply: $3\times2=6$, or 0110.
3. The signs differ, so negate: invert 0110 to 1001 and add 1, giving 1010.
4. Check: $10-16=-6$, inside $-8\ldots+7$.
5. Compare the raw patterns: 1101×0010 is $13\times2=26$, or 0001 1010, while -6 at eight bits is 1111 1010. Only the low four bits agree, which is why the patterns cannot simply be multiplied. §3.5

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?
4. State the subtraction overflow rule.

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

Formula map

Idea	Relation	Use
Positional value	$\sum_{i=0}^{k} d_i b^i$	Convert a base-b numeral to a value.
Unsigned n-bit range	$0 \dots 2^n-1$	Check if a non-negative integer fits.
Two's-complement range	$-2^{n-1} \dots 2^{n-1}-1$	Check signed fixed-width values.
Two's-complement negation	Invert n bits, then add 1	Negate within the selected width.
Excess-B	actual = stored − B	Decode a biased pattern.
Division	dividend = quotient×divisor + remainder	Verify quotient and remainder.
Signed overflow	exact result outside representable range	Determine whether the encoded answer is valid.
Addition overflow	same-sign operands, opposite-sign result	Test a sum from its sign bits.
Subtraction overflow	operand signs differ, result has the subtrahend's sign	Test A−B from its sign bits.

Check every answer

State the base, encoding, and bit width

Verify place values or arithmetic in decimal

Compare signed results with the allowed range

Units are not used in these number-representation problems; check the encoding and range instead.

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. **Signed multiplication:** expand $X(2^n-Y)$ and compare it with the correct 2n-bit pattern $2^{2n}-XY$. §3.5

Made by Thanate Ang... (Focus)

Explain it

Answer each why or when question aloud before opening its 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?

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

Made by Thanate Amornjiraporn (Focus)

Mixed practice

Try all 28 problems before opening the worked answers. The topics are interleaved so you must select the encoding or operation each time.

1. Convert 37_{10} to binary and check by place value.
2. Decode 1110 as a four-bit two's-complement number.
3. How many bits represent one hexadecimal digit?
4. Encode decimal 73 in four-bit BCD.
5. How many values are available in seven-bit ASCII?
6. Add 1011_2 and 0110_2 as unsigned values.
7. What is the four-bit two's-complement range?
8. Convert 110111_2 to octal.
9. In excess-4, what value does the three-bit pattern 110 represent?
10. For $29 \div 5$, give quotient and remainder and verify the division identity.
11. Negate 0011 in four-bit two's complement.
12. Convert 11101010_2 to hexadecimal.
13. Why does two's-complement addition discard a carry-out?
14. Write decimal 83 in BCD and ordinary binary.
15. Compute $7-(-6)$ using four-bit two's complement. State the exact result and whether it fits.
16. What do the first 128 Unicode code points have in common with ASCII?
17. Multiply 110_2 by 101_2 as unsigned values.
18. In four-bit two's complement, add 0110 and 0011. Does it overflow?
19. What is the decoded value of pattern 0111 in four-bit excess-8?
20. What is one reason BCD can be useful despite its extra storage?
21. Compute $1001010_2 - 10111_2$ as unsigned values, and name the minuend and the subtrahend.
22. In eight-bit two's complement, add 11111100 and 11111001. Is there a carry-out? Does it overflow?
23. How do the 128 ASCII patterns divide into letters, digits, punctuation and others?
24. Multiply 1011_2 by 111_2 as unsigned values.
25. Compute $5 \times (-3)$ by the magnitude method. Does the result fit in four bits? Give the eight-bit pattern.
26. In four-bit two's complement, compute $0011 - 1000$. Does it overflow? Name the rule you used.
27. Why is 1100 not a valid four-bit BCD digit, and how many patterns share this problem?
28. Compare the patterns 0011 and 1010 as unsigned numbers. Does that order match their values in excess-8? In two's complement?

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

Made by Thanate Amornjiraporn (Focus)

Exam-style problems

Allow about 20 minutes per problem. State the bit width and encoding before each calculation.

1 · Compare encodings

Use four bits to represent the integer -5. Compare 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.

2 · Carry-out and overflow

Use four-bit two's-complement inputs for -4 and $+6$.

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.

3 · From hexadecimal to a checked result

Two bytes hold $C8_{16}$ and $9C_{16}$.

1. Write each byte in binary and give its unsigned decimal value.
2. Decode each byte as eight-bit two's complement.
3. Add the bytes as unsigned numbers, keeping every bit.
4. Keep eight bits of that sum, decode it as two's complement, and decide whether the signed addition overflowed.
5. Compute $C8_{16} - 9C_{16}$ by negating and adding, then test it for overflow.

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

Made by Nanate Amornjitaporn (Focus)

Slips in the notes

Each of these slips was checked on the rendered slide. They are transcription or wording errors, not different rules, and this guide uses the corrected form.

- **Slide 5 · "octet":** the slide calls an octal digit an octet. An octal digit is three bits; an octet is eight bits. This guide just says octal digit.
- **Slide 22 · "first 127 characters":** the slide says the first 127 Unicode characters match ASCII. ASCII has 128 codes, 0–127, as slide 20 says, so this guide says 128.
- **Slide 24 · missing minus sign in the Excess-8 row:** the entry for bit pattern 0111 is printed as 1. With four-bit excess-8, the decoded value is stored value 7 minus bias 8, which equals −1. Slides 17 and 19 use that mapping.
- **Slide 39 · lost minus and wrong exponent:** "X * (Y)" should read X * (−Y). "Sign extension to 2^{2n}" should read sign extension to 2n bits; 2^{2n} is the modulus, not a width.
- **Slide 40 · sign and steps:** "3 in two's complement … 1101" should say −3. The steps also conflict: they encode negatives, multiply ignoring signs, then apply a sign. Read literally, 1101×0010 gives 26, then −26, not −6. This guide multiplies the magnitudes instead (§3.5).

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. (1, 14)
- ☐ Convert binary groups to octal and hexadecimal. (3, 8, 12)
- ☐ State signed and unsigned ranges from a bit width. (7, 18)
- ☐ Decode patterns using distinct signed encodings. (2, 19, 28)
- ☐ Negate a two's-complement value at fixed width. (11, 15)
- ☐ Decode an excess-B value. (9, 19, 28)
- ☐ Write and read decimal digits in BCD. (4, 14, 20, 27)
- ☐ Explain ASCII and Unicode code-point relationships. (5, 16, 23)
- ☐ Add binary values and carry correctly. (6)
- ☐ Subtract binary values with borrows. (21)
- ☐ Multiply binary and signed values. (17, 24, 25)
- ☐ Verify quotient and remainder. (10)
- ☐ Add signed values and detect overflow. (18, 22)
- ☐ Subtract by negating and adding. (15, 26)
- ☐ Identify and correct the slide 24 excess-8 typo. (19)
- ☐ Rebuild the derivations from the formula list.
- ☐ Complete all three exam-style problems with checks.

Made by Thanate Amornjirapon (Focus)

Answers

Start from memory

1. It is the digit's coefficient on a power of the base, counting from power zero at the right. §3.1
2. -2^{n-1} through $2^{n-1}-1$. §3.2
3. Invert all bits and add 1, keeping the chosen width. For the minimum value -2^{n-1}, the result is the same pattern because its positive value cannot fit at that width. §3.2
4. Four bits, because $16 = 2^4$. §3.1
5. Nothing by itself. In fixed-width two's-complement addition, carry-out is discarded and is not the signed-overflow test. §3.5

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
4. Comparing patterns as unsigned numbers gives the correct order of values. §3.2
5. Six: 1010 through 1111. §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. 52 letters and 10 digits. §3.3
4. $2^{16}{=}65{,}536$. §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
3. 4, the number being subtracted. §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. Overflow occurs when the operand signs differ and the result has the subtrahend's sign. §3.5

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.

Made by Thanate Angunjaporn (Focus)

Mixed practice

1. $37=32+4+1$, so 100101_2. §3.1
2. Unsigned value $14-16=-2$. §3.2
3. Four bits, since $16=2^4$. §3.1
4. 7 is 0111 and 3 is 0011, so 73 is 0111 0011 in BCD. §3.2
5. $2^7=128$ code values. §3.3
6. $1011+0110=10001$. As an unsigned five-bit result it is 17. §3.4
7. -8 through $+7$. §3.2
8. Group 110 111; the groups are 6 and 7, so 67_8. Check: $6\times8+7=55=32+16+4+2+1$. §3.1
9. Pattern 110 is 6; $6-4=2$. §3.2
10. $29=5\times5+4$, so quotient 5 and remainder 4; $0\leq4<5$. §3.4
11. Invert 0011 to 1100 and add 1: 1101, which is -3. §3.2
12. 1110 1010 maps to E A, so EA_{16}. §3.1
13. The machine keeps only the fixed n result bits. Carry-out is outside that word and does not alone indicate signed overflow. §3.5
14. BCD: 1000 0011. Ordinary binary: $83=64+16+2+1$, so 1010011. §3.2
15. -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 $-8\ldots+7$, so overflow occurs. §3.5
16. Unicode code points 0–127 correspond to ASCII characters. §3.3
17. 110 is 6 and 101 is 5; $6\times5=30=11110_2$. §3.4
18. $0110+0011=1001$, which decodes as -7. The exact result is $+9$, outside the four-bit range $-8\ldots+7$; the two positive inputs produced a negative-sign pattern, so it overflows. §3.5
19. $7-8=-1$. §3.2
20. BCD keeps each decimal digit directly encoded, which can simplify decimal display or input/output. §3.2
21. Minuend 1001010, subtrahend 10111. $74-23=51=110011_2$. §3.4
22. $252+249=501=1$ 11110101, so there is a carry-out. The kept 11110101 decodes as $245-256=-11$, the exact sum $-4+(-7)$, inside $-128\ldots+127$. No overflow. §3.5
23. 52 letters, 10 digits, 32 punctuation marks and 34 others (whitespace and control characters); $52+10+32+34=128$. §3.3
24. 1011 is 11 and 111 is 7; $11\times7=77=1001101_2$. §3.4
25. Magnitudes give $5\times3=15$, and the signs differ, so the result is -15. That is outside $-8\ldots+7$, so it does not fit in four bits. At eight bits, $+15$ is 00001111; invert and add 1 to get 11110001, or $241-256=-15$. §3.5
26. -8 negates to itself, so $0011+1000=1011$, which decodes as -5. The exact result is $3-(-8)=+11$, above $+7$, so it overflows. Subtraction rule: the operand signs differ and the result has the subtrahend's sign. §3.5
27. 1100 is 12, and a BCD digit stops at 9. Six patterns, 1010–1111, are unused. §3.2

Made by Thanate Ameriiiraporn (Focus)

28. As unsigned numbers 3<10. In excess-8 they are −5 and 2, so the order matches. In two's complement they are +3 and −6, so it does not. §3.2

Made by Thanate Amornjiraporn (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. +4 is 0100, so −4 is 1011+1=1100; +6 is 0110. Addition gives 1 0010, a carry-out of 1. §3.2 §3.5
2. The retained 0010 is +2. The exact sum is also +2. §3.5
3. No overflow: +2 lies in −8…+7, and unlike signs cannot overflow in addition. A carry came out with a correct result, so carry-out is not the signed-overflow test. §3.5

Exam-style problems

1. C8 is 1100 1000=200, and 9C is 1001 1100=156. §3.1
2. 200−256=−56, and 156−256=−100. §3.2
3. 200+156=356=1 01100100_2. §3.4
4. The kept 01100100 is +100. The exact sum −156 is outside −128…+127, and two negatives gave a positive result, so it overflowed. §3.5
5. Negating 10011100 gives 01100011+1=01100100. Then 11001000+01100100=1 00101100, and the kept 00101100 is +44, matching −56−(−100). The operand signs are the same, so the subtraction cannot overflow; the carry-out changes nothing. §3.5

Made by Thanate Amornjiraporn (Focus)

Personal study guide based on the supplied "03_data_representation.pdf" (41 slides; slide 40 is dated 22 April 2024). Written in my own words; no lecture pages or figures are reproduced. The topic headings in this guide organise the slide sequence. Not an official FAU publication; check course details against the current notes. Made by **Thanate Amornjiraporn (Focus)**.