A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

1.1.2 Binary magnitudes, one's complement, subtraction and where BCD and hex are used

Name: _____ Class: _____ Date: _____ **Total: 33 marks**

KEY WORDS binary 二进制 • denary 十进制 • hexadecimal 十六进制 • overflow 溢出 • two's complement 补码

Objective

Build the skills to answer the **data representation** questions that sheet 1.1 does not reach. That sheet converts between bases, adds binary numbers and uses two's complement; this one covers the four things beside them that 1.1 also asks for: **binary magnitudes** and how a binary prefix differs from a decimal one, **one's complement** 反码, binary **subtraction** using positive and negative integers, and the **practical applications** of BCD and hexadecimal.

You must be able to:

- show understanding of binary magnitudes and the difference between **binary prefixes** 二进制前缀 and **decimal prefixes**: kibi and kilo, mebi and mega, gibi and giga, tebi and tera
- use **one's complement** representation for binary numbers, and convert a value from one representation to another
- perform binary **subtraction** using positive and negative binary integers
- describe practical applications where **Binary Coded Decimal** (**BCD**) 二进码十进数 and **hexadecimal** 十六进制 are used

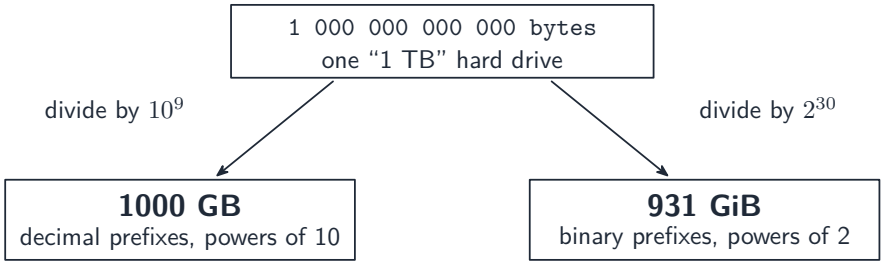
1 Worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

■ The two prefix families

A decimal prefix is a power of **10**; a binary prefix is a power of **2**. They are near enough to be confused and far enough apart to matter.

Decimal (SI)	Binary (memory)	In bytes
kilo (k) $= 10^3$	**kibi** (Ki) $= 2^{10}$	1024
mega (M) $= 10^6$	**mebi** (Mi) $= 2^{20}$	1 048 576
giga (G) $= 10^9$	**gibi** (Gi) $= 2^{30}$	1 073 741 824
tera (T) $= 10^{12}$	**tebi** (Ti) $= 2^{40}$	1 099 511 627 776



The bytes never changed. Only the size of the unit did.

Worked example. *A drive is sold as "1 TB". The operating system reports 931 GiB. Explain.*

The maker counts in decimal prefixes, so 1 TB $= 10^{12} = 1\,000\,000\,000\,000$ bytes. The operating system counts in binary prefixes, and one **gibibyte** is $2^{30} = 1\,073\,741\,824$ bytes, so

$$\frac{1\,000\,000\,000\,000}{1\,073\,741\,824} = \mathbf{931}\ \text{GiB}$$

Nothing is missing. The same bytes are being counted in a **larger unit**.

- Going the other way is multiplication: 4 GiB $= 4 \times 2^{30} =$ **4 294 967 296** bytes.
- Memory sizes are always powers of 2, because an address of n bits reaches exactly 2^n locations. That is why the binary family exists at all.

■ One's complement, and why two's complement replaced it

One's complement: to write a negative number, **invert every bit** of the positive. There is no "add 1".

	+30	−30
one's complement	`00011110`	`11100001`
two's complement	`00011110`	`11100010`

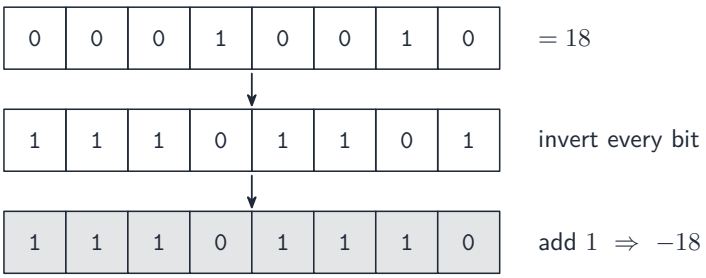
Reading a pattern **back** depends on which scheme you are told it is in. Take `11110100`. The sign bit is 1, so it is negative either way.

- as **one's** complement: invert to get `00001011` = 11, so the value is **−11**.

- as **two's** complement: invert to get `00001011`, add 1 to get `00001100` = 12, so the value is **−12**.

The drawback is **two zeros**: `00000000` is +0 and `11111111` is −0. One bit pattern is wasted, a test for zero has to check both, and the adder needs an extra correction step. Two's complement has a **single** zero and lets one circuit do both addition and subtraction, which is why it is what computers use.

■ Binary subtraction, two ways



Method 1, column subtraction. Work from the right. When the top bit is 0 and the bottom bit is 1, **borrow** 借位 from the left: the 0 becomes 2, and the column you borrowed from loses 1.

Method 2, the one the exam rewards: turn it into an addition. $A - B$ is $A + (-B)$, so take the **two's complement of** B and add. One circuit, no borrowing.

Worked example. *Work out `01011010` − `00110111` (that is* $90 - 55$*).*

1. Two's complement of `00110111`: invert to `11001000`, add 1 to get `11001001`.
2. Add:

$$\texttt{01011010} + \texttt{11001001} = \texttt{1 00100011}$$

1. **Discard the carry out of the eighth column.** The answer is `00100011` = **35**, and $90 - 55 = 35$.

Worked example. *Work out `00110010` − `01000110` (that is* $50 - 70$*).*

1. Two's complement of `01000110`: invert to `10111001`, add 1 to get `10111010`.
2. Add: `00110010` + `10111010` = `11101100`, with **no** carry out.
3. The sign bit is 1, so the answer is negative. Read it back: invert to `00010011`, add 1 to get `00010100` = 20. The answer is **−20**.

- A carry out of the top column is **discarded** in two's-complement subtraction; it is not an error and it is not overflow.
- **Overflow** 溢出 is different: it is when the true answer will not fit in the bits available, and in signed arithmetic you spot it by the sign bit being wrong – two positives giving a negative, or two negatives giving a positive.

■ Where BCD is used, and where hexadecimal is used

Both questions are "describe a practical application", so each answer must name a use **and** link it to a property of the representation.

BCD stores each denary digit in its own 4 bits: 93 is `1001 0011`, not binary 93 (`01011101`). Patterns `1010` to `1111` are unused.

- **Digital clocks, calculators and any device with a 7-segment display** 七段显示器. Each digit already has its own 4 bits, so it can be sent straight to its own display with no conversion of the whole number.
- **Currency and other decimal money amounts.** A value such as 0.10 is stored exactly, digit by digit; in ordinary binary a decimal fraction is a recurring value and has to be rounded, which accumulates error over many calculations.

Hexadecimal is not a different way of storing anything. One hex digit is exactly 4 bits, so a byte is exactly two hex digits – it is a shorter, more readable way for a **human** to write the binary that is already there.

- **Memory addresses** 内存地址 and memory dumps in low-level work, such as `0x7FFE`.
- **Colour values** in HTML and CSS, such as `#FF8800` – two hex digits per colour channel.
- **MAC addresses**, such as `AC:DE:48:00:11:22`.
- Also **error codes** and **Unicode code points**.

The reason is always the same: fewer characters than binary, fewer transcription mistakes, and conversion back to binary needs no arithmetic at all – just replace each digit with its nibble.

2 Practice

Now use the methods above.

2.1 State the number of bytes in each of these. [3]

(a) 1 kibibyte (KiB)

(b) 1 mebibyte (MiB)

(c) 2 gibibytes (GiB)

2.2 State the difference, in bytes, between one **kilobyte** (kB) and one **kibibyte** (KiB), and state which prefix family each belongs to. [2]

2.3 The denary value +45 is stored in 8 bits.

Write it in **one's complement** and state what −45 is in one's complement. [2]

2.4 The 8-bit pattern `11101100` is a negative number.

State its value (a) if it is **one's** complement, and (b) if it is **two's** complement. [2]

2.5 Work out `01101001` − `00101101` using two's complement. Show the two's complement you formed, the addition, and the final answer in binary. [3]

2.6 State **one** practical application of BCD and **one** practical application of hexadecimal. [2]

3 Exam-style questions

3.1 A memory module is described as 4 GiB and a hard drive is sold as 4 GB.

(a) Calculate the number of bytes in each. [2]

(b) Explain why the two figures are different even though both are written as "4 G". [2]

3.2 A computer could store negative integers in one's complement or in two's complement.

(a) Describe how a negative number is formed in **each** representation. [2]

(b) Give **two** reasons why two's complement is used in modern computers instead of one's complement. [2]

3.3 Two 8-bit values are `00101110` and `01010001`.

(a) Work out `00101110` − `01010001` using two's complement. [3]

(b) State the denary value of your answer, and explain how you read a negative two's-complement result. [2]

3.4 A programmer writing low-level code reads a memory dump printed in hexadecimal rather than in binary.

Explain **two** reasons why hexadecimal is used, and state **one** other practical application of hexadecimal. [3]

3.5 A supermarket till stores each price using BCD rather than ordinary binary.

Justify the use of BCD in this application. [3]

Solutions

Each answer shows the steps, then the **result**.

2.1

(a) $2^{10} =$ **1024** bytes

(b) $2^{20} =$ **1 048 576** bytes

(c) $2 \times 2^{30} =$ **2 147 483 648** bytes

2.2

- a kilobyte is $10^3 = 1000$ bytes and a kibibyte is $2^{10} = 1024$ bytes, so a kibibyte is **24 bytes** larger
- **kilo** is a decimal prefix, a power of 10; **kibi** is a binary prefix, a power of 2

2.3

- $+45 =$ **00101101**
- one's complement is every bit inverted, so $-45 =$ **11010010**

2.4

(a) one's complement: invert `11101100` to get `00010011` = 19, so the value is **−19**

(b) two's complement: invert to `00010011`, add 1 to get `00010100` = 20, so the value is **−20**

2.5

- two's complement of `00101101`: invert to `11010010`, add 1 to get `11010011`
- `01101001` + `11010011` = `1 00111100`
- discard the carry: the answer is **00111100**, which is 60 – and 105 − 45 = 60

2.6

- BCD, any one: digital clocks, calculators or any device driving a 7-segment display; storing currency amounts
- hexadecimal, any one: memory addresses and memory dumps; colour values in HTML or CSS; MAC addresses; error codes; Unicode code points

3.1

(a) 4 GiB $= 4 \times 2^{30} =$ **4 294 967 296** bytes

- 4 GB $= 4 \times 10^9 =$ **4 000 000 000** bytes

(b) "GiB" uses a **binary** prefix, a power of 2, while "GB" uses a **decimal** prefix, a power of 10

- a gibibyte is therefore larger than a gigabyte, so the same digit in front of it counts more bytes; memory is sized in powers of 2 because an n-bit address reaches 2^n locations, while drive makers quote the decimal figure

3.2

(a) **one's** complement: invert every bit of the positive number

- **two's** complement: invert every bit of the positive number **and then add 1**

(b) any two: one's complement has **two** representations of zero (`00000000` and `11111111`), so a bit pattern is wasted and a test for zero must check both; two's complement has a single zero

- addition and subtraction can use the **same** circuit in two's complement, with no end-around carry correction, so the hardware is simpler and the arithmetic is faster

3.3

(a) two's complement of `01010001`: invert to `10101110`, add 1 to get `10101111`

- `00101110` + `10101111`
- = **11011101**, with no carry out of the eighth column

(b) the sign bit is 1, so the value is negative: invert `11011101` to `00100010`, add 1 to get `00100011` = 35, so the answer is **−35**

- and $46 - 81 = -35$, which checks

3.4

- any two reasons: one hex digit is exactly 4 bits, so a byte is two characters instead of eight – far shorter to write and to read
- fewer characters means fewer mistakes when a human copies or reads a long value, and converting back to binary needs no arithmetic, just replacing each digit with its nibble
- one other application: colour values in HTML or CSS; MAC addresses; error codes; Unicode code points
- a reason that only says "hex is shorter" without saying **why** (4 bits per digit) does not earn the second mark

3.5

- in BCD each **denary digit** is stored in its own 4 bits, so the digits of a price are kept separately
- a decimal fraction such as 0.10 is stored **exactly**, whereas converting it to ordinary binary gives a recurring value that must be rounded, and the error grows across many prices
- each digit can also be sent straight to its own position on the display, and prices can be added digit by digit, with no conversion of the whole number