A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

1.1 Data Representation

Name: _____ Class: _____ Date: _____ **Total: 28 marks**

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

Objective

Build the skills to answer the real exam questions on **data representation** 数据表示 — number bases, binary arithmetic, two's complement, BCD, and **character sets** 字符集.

You must be able to:

- convert between **binary** 二进制, **denary** 十进制 and **hexadecimal** 十六进制, and use **BCD** 二进码十进数
- state the **minimum number of bits** needed to store a value
- add 8-bit binary numbers and **name and describe overflow** 溢出
- use **two's complement** 补码: write a negative denary number in binary, read a two's-complement value, and subtract
- represent character data with **ASCII** and **Unicode**, and explain why Unicode is used

1 Worked examples

Study these first. Each one shows the method for an exam question type used later — follow the steps and you can do the Practice and Exam-style questions yourself.

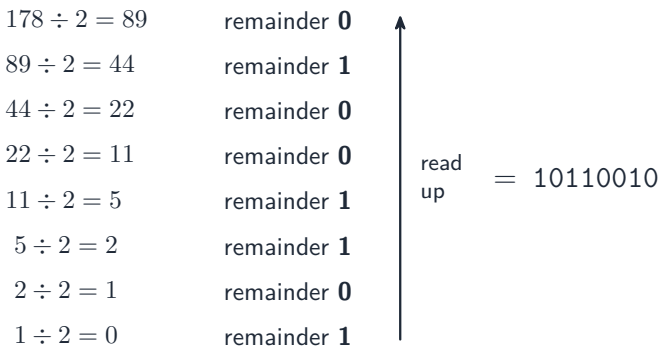
■ Binary ↔ denary

Multiply each bit by **2 raised to its position** (positions count from 0 at the right), then add. This same method works for *any* base — for hexadecimal you use powers of 16.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	0	0	1	0

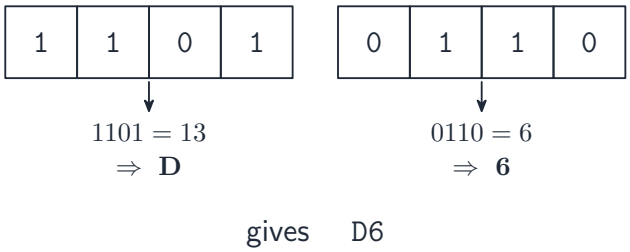
$$(1\times2^7)+(1\times2^5)+(1\times2^4)+(1\times2^1)$$
$$= 128+32+16+2 = \mathbf{178}$$

To go **denary → binary**, divide by 2 again and again; the **remainders** 余数, read from **bottom to top**, are the bits.



Hexadecimal

Each hex digit is exactly 4 bits (a **nibble** 半字节): 0–9, then A = 10 up to F = 15. **Hex → denary** uses the powers method with base 16: `2F` $= (2 \times 16^1) + (15 \times 16^0) = 32 + 15 =$ **47**. **Shortcut binary ↔ hex:** group the bits into nibbles of 4 from the right and convert each.



Number of bits to store a value

To store the values 0 to N, you need the smallest number of bits b with $2^b > N$. Example: to store 0–200, $2^7 = 128$ is too small but $2^8 = 256 > 200$, so **8 bits**. (An 8-bit byte holds 0–255.)

Binary addition and overflow

Add one column at a time from the right; when a column makes 2, write 0 and **carry** a 1.

```
  0101 1100   (92)
+ 0011 0110   (54)
------------
  1001 0010   (146)
```

Overflow is when the answer is too big for the register (over 255 for 8 bits): a 1 carries out of the leftmost (bit-7) column, so the stored 8-bit result is wrong.

```
  1100 1000   (200)
+ 0101 0000   (80)
------------
1 0001 1000   (carry out of bit 7 → overflow)
```

In words: **overflow** is when a calculation gives a result with more bits than the register can hold, so the carry out of the most-significant bit is lost and the stored answer is incorrect.

Two's complement: negative numbers

In two's complement the **most significant bit** is the **sign bit** (0 = positive, 1 = negative). For 8 bits the range is -128 to $+127$.

- **Write** -18**:** take $+18$ = `00010010`, **invert** every bit → `11101101`, **add 1** → `11101110`.
- **Read `11101110`:** the sign bit is 1, so it is negative. Invert → `00010001`, add 1 → `00010010` = 18, so the value is -18.
- **Subtract** by adding a negative. $45-18$: add `00101101` (45) and `11101110` (-18), then **discard** the carry out of bit 7:

```
  0010 1101   (45)
+ 1110 1110   (-18)
------------
1 0001 1011   (discard carry → 27)
```

Binary Coded Decimal (BCD)

BCD stores each denary digit on its own as a 4-bit code — *not* the plain binary of the whole number. For 59: 5 → `0101` and 9 → `1001`, so 59 is `0101 1001`. Each nibble uses only 0–9; patterns `1010`–`1111` are invalid.

Character sets: ASCII and Unicode

A computer stores text as numbers: each character has a **code point** 码点 fixed by a character set.

- **ASCII** uses **7 bits** → 128 characters (basic English letters, digits, punctuation, control codes). Example: `'A'` = 65 = `1000001`.

- **Unicode** is a universal set covering (almost) every script plus symbols and emoji, so it needs **more bits per character**.

A character's stored code is just a binary number, so you convert it like any other: the value `0010 0111` $= 32 + 4 + 2 + 1 = 39$.

Why Unicode? It represents **far more characters** than ASCII (every language, not just English), at the cost of **larger files** for plain English text.

2 Practice

Now apply the methods above.

2.1 Convert binary `01101010` to denary. [1]

2.2 Convert denary 200 to 8-bit binary. [2]

2.3 Convert binary `11110001` to (a) denary and (b) hexadecimal. [2]

2.4 State the **minimum number of bits** needed to store the denary value 500. Show how you decided. [2]

2.5 Represent the denary number 59 in **BCD**. [2]

2.6 Add the 8-bit numbers `01011100` and `00110110`. State whether overflow occurs. [2]

3 Exam-style questions

3.1 Add the 8-bit binary numbers `10110100` and `01101101`. Show your working, then **name and describe** the error that occurs. [4]

3.2 (a) Convert the hexadecimal number `3D` to denary. Show your working. [2]

(b) Convert the denary number 158 to hexadecimal. Show your working. [2]

3.3 (a) Write the denary number −45 as an 8-bit two's complement binary number. Show your working. [2]

(b) The 8-bit two's complement number `11101110` is stored. Calculate the denary value

it represents. [2]

3.4 A computer stores text using a character set.

(a) The text is stored using the **Unicode** character set instead of **ASCII**. Give **two** advantages of using Unicode. [2]

(b) A character has the binary code `0010 0111`. Convert this code to denary. [1]

3.5 A program stores currency amounts using **BCD** rather than ordinary binary.

(a) State what **BCD** stands for. [1]

(b) Give one **advantage** of using BCD for this purpose. [1]

Solutions

2.1 `01101010` $= 64 + 32 + 8 + 2 = 106$.

2.2 Repeated division by 2 gives remainders, read bottom-up $\rightarrow$ `11001000`.

2.3 (a) `11110001` $= 128 + 64 + 32 + 16 + 1 = 241$.
(b) split into `1111 | 0001` = `F1`.

2.4 $2^8 = 256$ is too small for 500, but $2^9 = 512 > 500$, so **9 bits**.

2.5 $5 \rightarrow$ `0101`, $9 \rightarrow$ `1001`, so `0101 1001`.

2.6 `01011100` (92) + `00110110` (54) = 146 = `10010010`; $146 \leq 255$, so **no overflow**.

3.1 Adding the columns: \

```
  1011 0100   (180)
+ 0110 1101   (109)
------------
  0010 0001   (carry out of bit 7)
```

$180 + 109 = 289 > 255$. The error is **overflow**: the result needs 9 bits but only 8 are stored, so the carry out of the most-significant bit is lost and the stored answer is wrong.

3.2 (a) `3D` $= (3 \times 16) + 13 = 48 + 13 = 61$.
(b) $158 = (9 \times 16) + 14$, and 14 = E, so `9E`.

3.3 (a) $+45$ = `00101101`; invert $\rightarrow$ `11010010`; add 1 $\rightarrow$ `11010011`.
(b) sign bit is 1 $\rightarrow$ negative; invert `11101110` $\rightarrow$ `00010001`; add 1 $\rightarrow$ `00010010` = 18; so the value is -18.

3.4 (a) Any **two**: Unicode can represent characters from (almost) every language / many more characters than ASCII; it includes symbols and emoji; it avoids the regional clashes of extended ASCII.
(b) `0010 0111` $= 32 + 4 + 2 + 1 = 39$.

3.5 (a) Binary Coded Decimal.
(b) Each denary digit is stored exactly, so values like money are not changed by rounding errors when converting fractions to binary (or: BCD digits map directly to a display).