

Solutions

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

2.1

- (a) $2^{10} = \mathbf{1024}$ bytes
- (b) $2^{20} = \mathbf{1\,048\,576}$ bytes
- (c) $2 \times 2^{30} = \mathbf{2\,147\,483\,648}$ 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

- $+45 = \mathbf{00101101}$
- one's complement is every bit inverted, so $-45 = \mathbf{11010010}$

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 $60 - \text{and } 105 - 45 = 60$

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) $4\text{ GiB} = 4 \times 2^{30} = \mathbf{4\,294\,967\,296}$ bytes
- $4\text{ GB} = 4 \times 10^9 = \mathbf{4\,000\,000\,000}$ 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
- $00101110 + 10101111$
- $= \mathbf{11011101}$, 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