

Solutions

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

2.1

- positive: the sign bit is 0 and the next bit is 1, so it begins 01
- negative: the sign bit is 1 and the next bit is 0, so it begins 10
- in both cases the bit after the sign is the **opposite** of the sign bit

2.2

- mantissa $01100000 = 0.5 + 0.25 = 0.75$; exponent $00000010 = 2$
- $0.75 \times 2^2 = \mathbf{3}$

2.3

- any one: it gives the greatest possible **precision** for the bits available, because no mantissa bit is wasted on a leading zero; or it makes the representation **unique**, so one value has one stored form

2.4

- **overflow**: the result is larger than the largest value the format can represent
- **underflow**: the result is a non-zero value smaller than the smallest the format can represent, so it is stored as zero

2.5

- the **exponent**

3.1

- **precision increases**: more mantissa bits means more significant figures, so values are stored more accurately and rounding errors are smaller
- **range decreases**: fewer exponent bits means the largest and smallest magnitudes that can be stored both move towards 1
- the total is fixed, so one can only be gained at the other's expense; overflow and underflow therefore occur for values that the old allocation could have held

3.2

(a) mantissa $00011000 = 0.125 + 0.0625 = 0.1875$; exponent = 5

- $0.1875 \times 2^5 = 0.1875 \times 32 = \mathbf{6}$

(b) the mantissa must begin 01, so shift **left** by **2** places

- new mantissa = **01100000**
- the exponent **decreases** by 2, from 5 to 3, so it is **00000011**
- shifting right, or increasing the exponent, changes the value and scores nothing

(c) $0.75 \times 2^3 = 0.75 \times 8 = 6$, the same value as in (a)

3.3

- the mantissa fixes the **precision** – how many significant figures are kept – and it always represents a value between -1 and $+1$
- the **exponent** fixes the **magnitude**, by saying what power of 2 the mantissa is multiplied by, so it alone decides how large or how small the number can be
- overflow happens when a result needs an exponent larger than the exponent field can hold, and underflow when it needs one more negative than the field can hold; running out of mantissa bits causes a loss of accuracy, not overflow

3.4

(a) 0.1 and 0.2 are **recurring** binary fractions, because their denominators are not powers of 2, so neither can be stored exactly

- each is rounded to fit the mantissa, and the two small errors survive into the sum, so the result differs from the stored value of 0.3 in its last bits

(b) test that the **difference** from 0.3 is smaller than a small tolerance rather than testing for zero, for example IF $X - 0.3 < 0.000001$ AND $0.3 - X < 0.000001$ THEN

- a modulus function is not on the 9618 insert, so a question needing one supplies it

3.5

- currency values such as 0.10 are recurring binary fractions and cannot be stored exactly in floating-point
- the rounding errors accumulate over many transactions, so totals drift and accounts fail to balance – an error of any size is unacceptable in money
- use **fixed-point** representation, or **BCD**, in which each denary digit is stored exactly