Solutions

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

2.1

- logical shift left one place: insert 0 at the right
- `00000101` → **`00001010`**; denary 5 → **10**

2.2

- it **halves** the number (integer ÷2)

2.3

- bit 2 is the third bit from the right, so the mask is `00000100`
- operation **OR**: `R OR 00000100`

2.4

- a left shift of 3 places multiplies by $2^3 = $ **8**

3.1

(a) logical shift left two places

- `00000110` → **`00011000`**; denary **24**

(b) it **multiplies the value by 4**

3.2

(a) mask `00000001`, operation **OR** (`R OR 00000001`)

(b) mask `00001000`, operation **AND** (`R AND 00001000`)

- if the result is non-zero, LED 3 is on

3.3

- a logical shift puts a **0** into the top bit, which would make a negative number look positive
- an arithmetic shift **copies the sign bit**, so the number stays negative

3.4

- a logical shift fills the vacated end with **0** and the bit shifted off the other end is **lost**
- a cyclic shift feeds that bit **back in** at the other end, so no bits are lost

3.5

(a) `OR B00000001` (or `OR #1`, or `OR &01`)

- `ACC` becomes **`01101101`**

(b) `AND B10111111`

- the mask is 1 everywhere **except** bit 6, so every other bit survives the AND unchanged

(c) `AND B00000100`

- the AND leaves every other bit zero, so `ACC` is **non-zero** only if bit 2 was 1
- the program then branches on whether `ACC` is zero

(d) `LSR #3` on `10011110` → **`00010011`**

- `XOR B00001111` flips the low four bits → **`00011100`**
- `LSL #2` → **`01110000`**

(e) an arithmetic shift copies the **sign bit** into the vacated places

- `10011110` → **`11110011`**
- the logical shift in (d) put zeros in, turning a negative number positive; the arithmetic shift keeps it negative, which is what halving a negative number must do

(f)

```
  10110101
+ 01011010
----------
 100001111
```

- nine bits; the 8-bit answer is `00001111` with an **overflow** bit of 1
- as unsigned 8-bit integers the result is **not valid**: $181 + 90 = 271$, which will not fit in 8 bits