A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

4.3 Bit manipulation

Name: ____________ Class: ________ Date: __________ **Total: 33 marks**

KEY WORDS bit manipulation 位操作 • logical shift 逻辑移位 • mask 掩码 • cyclic shift 循环移位 • bit 位

Objective

Build the skills to answer exam questions on **bit manipulation 位操作** — binary shifts and using bit masks to monitor and control a device.

You must be able to:

- perform **logical binary shifts** (left and right) and state their effect on the value
- use a **mask 掩码** to **set**, **clear**, **toggle** and **test** a single bit
- write the **assembly instructions** that do this: `AND`, `OR`, `XOR`, `LSL`, `LSR`
- read the three operand forms: `#` denary, `B` binary, `&` hexadecimal

1 Worked examples

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

Binary shifts

A **logical shift** moves every bit left or right, filling the new positions with 0.

0	0	0	0	1	0	1	1	$= 11$

LSL #1: each bit moves left, a 0 enters on the right

0	0	0	1	0	1	1	0	$= 22 \quad (11 \times 2)$

set bit 2

```
    01001000
OR  00000100
------------
=   01001100
```

clear bit 6

```
    01001000
AND 10111111
------------
=   00001000
```

toggle bit 3

```
    01001000
XOR 00001000
------------
=   01000000
```

Bit masking with AND/OR isolates or sets chosen bits

- **left shift** by $n \to \times\ 2^n$ (for an unsigned number).

- **right shift** by $n \rightarrow$ **integer** $\div 2^n$.
- An **arithmetic** right shift keeps the **sign bit**, so a negative signed number stays negative.
- A **cyclic shift** 循环移位 (rotate) feeds the bit that falls off one end back in at the other end, so no bits are lost.

■ Bit masks (monitor and control)

A device often uses **one bit per signal** (bit n = LED n). Combine the register with a **mask** using a logic operation:

Goal	Operation	Mask (for bit n)
set bit n to 1	`R OR mask`	bit $n = 1$, rest 0
clear bit n to 0	`R AND mask`	bit $n = 0$, rest 1
toggle bit n	`R XOR mask`	bit $n = 1$, rest 0
test bit n	`R AND mask`, then check $\neq 0$	bit $n = 1$, rest 0

Example — turn on bit 0 of `00000100` without changing the rest: `00000100 OR 00000001 = 00000101`.

■ Writing it as an assembly instruction

The exam asks for the **instruction**, not just the mask. Every one of these works on the **accumulator**, `ACC`, and there is only one general-purpose register:

Instruction	What it does to `ACC`
`AND #n` / `AND Bn` / `AND &n`	bitwise AND of `ACC` with the operand
`AND <address>`	bitwise AND of `ACC` with the contents of that address
`OR` and `XOR`	the same two forms, with OR and XOR
`LSL #n`	shift `ACC` **left** n places, zeros in on the right
`LSR #n`	shift `ACC` **right** n places, zeros in on the left

The operand prefix says which base the number is written in:

- `#123` — a **denary** number;
- `B01001010` — a **binary** number;
- `&4A` — a **hexadecimal** number.

So "set the least significant bit of `ACC` to 1" is `OR B00000001` (or `OR #1`, or `OR &01` — all the same instruction with the operand written three ways). "Clear bit 3" is `AND B11110111`. "Test bit 3" is `AND B00001000`, then check whether `ACC` is zero.

A `<label>:` in front of an instruction names it so a jump can reach it, and a `<label>: <data>` line gives a symbolic address to a memory location holding that data.

2 Practice

Now apply the methods above.

2.1 Perform a **logical shift left by 1** on `00000101`, and give the denary value before and after. [2]

2.2 State the arithmetic effect of a **logical shift right by 1** on an unsigned number. [1]

2.3 Give the mask and the operation needed to **set bit 2** of a register to 1, leaving the other bits unchanged. [2]

2.4 A logical shift **left by 3** multiplies an unsigned number by what? [1]

3 Exam-style questions

3.1 (a) Perform a logical shift **left by 2** on `00000110`, and give the denary result. [2]

(b) State the effect this shift has on the value. [1]

3.2 An 8-bit register controls 8 LEDs (bit n = LED n).

(a) Give the mask and operation to turn **on only LED 0**, leaving the others unchanged. [2]

(b) Give the mask and operation to **test whether LED 3 is on**. [2]

3.3 Explain why an **arithmetic** right shift, not a logical one, is used when halving a **negative** signed number. [2]

3.4 State the difference between a **logical** shift and a **cyclic** shift. [2]

3.5 A microcontroller uses an 8-bit accumulator `ACC` to control a set of sensors and indicator lamps. Bit 0 is the least significant bit.

(a) `ACC` currently contains `01101100`. **Write** the single bit-manipulation instruction that **sets the least significant bit to 1** without changing any other bit, and **give** the resulting contents of `ACC`. [2]

(b) **Write** the single instruction that **clears bit 6 to 0**, leaving the other bits unchanged, using a **binary** operand. [2]

(c) **Write** the instruction that would **test whether bit 2 is set**, and **describe** how the program then decides the answer. [3]

(d) **Complete** the table by giving the contents of `ACC` after each instruction. Each row starts from the value shown in the row above. [3]

instruction	**contents of `ACC`**
starting value	`10011110`
`LSR #3`	________________
`XOR B00001111`	________________
`LSL #2`	________________

(e) The value `10011110` is now treated as a **two's complement** integer. **Show** the result of an **arithmetic** right shift of 3 places on it, and **explain** why the result differs from your answer in (d). [3]

(f) **Complete** the binary addition below, showing your working and any **overflow** bit. **State** whether the answer is valid if the values are unsigned 8-bit integers. [3]

$$10110101 + 01011010$$

Solutions

2.1 `00000101` (5) → `00001010`; denary before **5**, after **10**.

2.2 It **halves** the number (integer ÷ 2).

2.3 Mask `00000100`, operation **OR**: `R OR 00000100`.

2.4 By **8** (that is 2^3).

3.1 (a) `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` gives `00010011`; `XOR B00001111` flips the low four bits, giving `00011100`; `LSL #2` gives `01110000`.
(e) An arithmetic shift copies the **sign bit** into the vacated places, so `10011110` becomes `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.