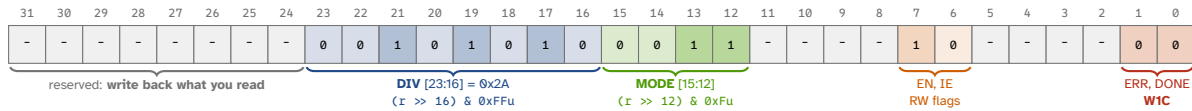


In one line: A register is a **memory address** you reach through a volatile `uint32_t *`; you change *one field* with a **mask + shift** and must keep every other bit intact — `|=` sets, `&=` ~ clears, `^=` toggles, `(r >> n) & 1u` reads. The dangerous part is not the operator, it is the **read-modify-write**.

CTRL — an illustrative 32-bit peripheral register (value `0x002A3080`); bit 31 left, bit 0 right. Field = *position + width*.



The idioms (Q4: masks swapped, toggle unknown)

set bit <i>n</i>	<code>r = (1u << n)</code>	OR forces 1
clear bit <i>n</i>	<code>r &= ~(1u << n)</code>	AND with a hole
toggle bit <i>n</i>	<code>r ^= (1u << n)</code>	XOR flips
test bit <i>n</i>	<code>(r >> n) & 1u</code>	0 or 1
get field	<code>(r >> pos) & mask</code>	mask=(1u<<w)-1
put field	<code>r = (r & ~(mask<<pos)) ((v & mask)<<pos)</code>	clear, then OR

The *bit number* goes in the shift, never in the mask: bit 3 is `1u << 3 = 0x08`. Write 1u, not 1 (see traps).

Tricks worth knowing cold

- `x & (x - 1)` clears the **lowest set bit**: -1 turns that 1 into 0 and every 0 below it into 1. Loop it = Kernighan popcount, O(set bits).
- `x & (~x + 1)` (= `x & -x`) isolates the lowest set bit.
- Power of two: `x && !(x & (x - 1))` — the `x != 0` guard is the trap.
- Round up to a power-of-two a: `(x + a - 1) & ~(a - 1)` — buffer / DMA alignment.
- Popcount: `__builtin_popcount` (GCC/Clang), `C23 stdc_count_ones` (<stdbit.h>). Also `__builtin_ctz/clz` (undefined for 0).
- `h & (cap - 1) = h % cap` when cap is 2^k .
- Byte *k* of a value: `(v >> (8*k)) & 0xFFu` — arithmetic, so it is the same on little- and big-endian. Endianness only bites when you reinterpret *memory* (ESP32 is little-endian: bytes 01 02 03 04 read as 0x04030201).

Types: <stdint.h>

- `uint8_t ... uint64_t` exact width; `int` is “at least 16 bits” — never use it for a register or a wire format. `uintptr_t` holds an address.
- Literals: `UINT32_C(1)`, `1u`, `1ull`; print with `PRId32/PRId32` from <inttypes.h>.
- Promotion**: anything narrower than `int` becomes `int` first. `uint8_t x = 0xFF`; `~x` is `0xFFFFF000`, not 0.

Shifts: unsigned vs signed

- Unsigned `>>` shifts in zeros (logical). Signed negative `>>` is *implementation-defined* (GCC: arithmetic, copies the sign).
- `1 << 31`: shifting a 1 into `int`’s sign bit is **UB**; `1u << 31 = 0x80000000`. Shift by $\geq$ the width (`1u << 32`) is UB for every type.

Register access

```
#define REG32(a) (*(volatile uint32_t *) (a))
#define CTRL    REG32(PERIPH_BASE + 0x10) // illustrative
#define DIV_POS  16u
#define DIV_MSK  0xFFu

uint32_t div = (CTRL >> DIV_POS) & DIV_MSK; // read field
CTRL = (CTRL & ~(DIV_MSK << DIV_POS)) // RMW: clear
      | ((new_div & DIV_MSK) << DIV_POS); // insert
STATUS = 1u << DONE_BIT; // W1C: write only that 1, never |=

// ESP-IDF (soc/soc.h): one store, no read -> atomic
REG_WRITE(GPIO_OUT_W1TS_REG, 1u << 5); // GPIO5 high
REG_WRITE(GPIO_OUT_W1TC_REG, 1u << 5); // GPIO5 low
```

- volatile**: every access is a real load/store (never kept in a register, dropped or merged). No atomicity, no ordering.
- RMW hazard**: `r |= b` is load → OR → store. An ISR or the *other core* changing another bit in between is overwritten → lost update.
- Hence **SET/CLR registers**: ESP32 `GPIO_OUT_W1TS/W1TC` (write-1-to-set/-clear) — one store, only the 1-bits act.
- W1C** status bits: `STATUS |= DONE` writes back every pending 1 → clears interrupts you never handled. Some bits clear *on read*.

C bitfields — why not for hardware

Bit order, allocation unit, straddling and plain-`int` signedness are **implementation-defined**; the compiler picks the *access width* (a byte store to a 32-bit-only register can fault) and each field write is a hidden RMW. ESP-IDF’s `soc/*_struct.h` uses them anyway — one pinned compiler + ABI. Portable: masks + shifts; wire formats: memcpy + shifts.

Interview traps

- You swapped the masks** (set 5, cleared 3) — the bit number goes in the shift. Toggle = **XOR**, the one you missed.
- if (r & 1u << n == 0)** — `==` binds tighter than `&`: it parses as `r & ((1u << n) == 0)`. Parenthesise every mask test.
- 1 << 40** into a `uint64_t` is still an `int` shift → UB. ESP-IDF’s `pin_bit_mask` is 64-bit: write `1ULL << pin`.

Remember

OR sets, AND-NOT clears, XOR flips, shift-then-AND reads — and every `|=` on a register is three instructions.

Likely questions

- Why `1u`? — `1 << 31` overflows signed `int` = UB.
- Two tasks, one register, different bits? — lost update → `W1TS/W1TC` or a lock.
- `n & (n-1)?` — drops the lowest 1; `== 0` ⇒ power of 2 ($n > 0$).
- Bitfields for registers? — layout + access width not portable.