📌 **Why bitwise operations matter in embedded systems:**

& (AND) – Masking bits

Use when you want to isolate specific bits.
🧪 *Check if a pin is high:*

```
if (PINA & (1 << PA0)) {
    // PA0 is HIGH
}
```

✅ Only the bit at `PA0` is checked; others are ignored.

| (OR) – Setting bits

Use to set a bit without affecting others.
💡 *Turn on an LED connected to PA0:*

```
PORTA |= (1 << PA0);
```

🔐 Other bits remain unchanged. Only PA0 is turned ON.

---

^ (XOR) – Toggling bits

Flip the state of a bit: 1 becomes 0, 0 becomes 1.
🕹️ *Toggle an LED state:*

```
PORTA ^= (1 << PA0);
```

🌀 Great for blinking LEDs or switching state with minimal logic.

~ (NOT) – Inverting bits

Flip all bits. Useful when working with *active-low* signals.
🚨 *Example:* Invert a mask or handle logic inversion:

```
uint8_t inverted = ~0x0F; // Becomes 0xF0
```

`<<` and `>>` – Bit shifting

Used to move bits left or right.
⚙️ *Shifting a `1` to the correct bit position:*

```
(1 << 3) // Results in 0b00001000
```

📍 Useful when creating bitmasks or setting specific bits in registers.

🧠 **Why this matters:**
Bitwise operations are how we:
✅ Read sensor data from specific bits
✅ Write config values into control registers
✅ Set/clear flags
✅ Handle interrupts
✅ Control I/O pins with precision

They allow **fast, memory-efficient**, and **deterministic** code—critical for real-time systems and bare-metal programming.

🔧 **More real-world C snippets:**

```
// Clear PA0, set PA1
PORTA = (PORTA & ~(1 << PA0)) | (1 << PA1);

// Check if multiple pins (PA0 and PA1) are high
if ((PINA & ((1 << PA0) | (1 << PA1))) == ((1 << PA0) | (1 << PA1)))
{
    // Both PA0 and PA1 are high
}// Toggle only PA2 and PA3
PORTA ^= (1 << PA2) | (1 << PA3);
```