HW0206 - Bit Operation

The problem with the code arises from using a **signed integer** (`int32_t`) for the bitmask `bit`. When you right-shift a signed integer with a negative value (after shifting `1 << 31`), it performs an **arithmetic shift**, which preserves the sign bit by filling with 1s. This corrupts the bitmask, leading to incorrect bit checks.

Fix: Use an **unsigned integer** (`uint32_t`) for the bitmask. This ensures a **logical right shift**, filling with 0s and maintaining the correct bitmask position.

Here's the corrected code:

```
#include <stdio.h>
#include <stdint.h>

int main()
{
    int32_t number = 0;
    scanf("%d", &number);

    uint32_t bit = 1;       // Use unsigned type
    bit = bit << 31;        // Start at the MSB

    for (int i = 0; i < 32; i++)
    {
        printf("%c", (bit & number) ? '1' : '0');
        bit = bit >> 1;     // Logical right shift for unsigned
    }

    return 0;
}
```

Explanation:

- **Unsigned Bitmask:** Using `uint32_t` ensures right shifts (`>>`) are **logical** (fills with 0s), preventing sign extension.
- **Correct Bit Check:** Each shift correctly moves the bitmask from the most significant bit (MSB) to the least significant bit (LSB), accurately extracting each bit of `number`.

This change guarantees that each bit of the 32-bit integer is displayed correctly in binary form.

The above is an AI-generated analysis of the issue (courtesy of DeepSeek R1), but let's dive further than that.

The issue with the bits:

So, we can see that the crux of the issue is that the signedness is what messes up. In this way, we can say that, in a more computer architecture-y sort of perspective, that different shifts occur within the integer data value. In a signed data type, once you right shift any negative value, it performs an arithmetic shift that fills the rest of the bits with 1's. Meanwhile, an unsigned integer utilizes the right-shift as a logical shift, meaning it fills the remaining the bits with 0's.

For example, the integer `5` expressed as both a signed int and unsigned int after will end up different for these two different scenarios.

> Original : 5 = 0101

> Signed bitmask = 0111

> Unsigned bitmask = 0101

Then, reinterpreting them back as 32-bits, the signed output will be expressed as,

```
01111111111111111111111111111111
```

meanwhile the unsigned output is,

```
00000000000000000000000000000101
```

which is the correct and expected output!