02. Bitwise Operations Deep Dive - Comprehensive Textbook Reference

> **Module Focus**: Bit-level manipulations, Binary representations, Bitwise AND (`&`), OR (`|`), XOR (`^`), NOT (`~`), Left Shift (`<<`), Right Shift (`>>` , `>>>`), 2's Complement Math, and Bitwise Masking Shortcuts with worked step-by-step binary traces.

---

1. Bitwise Operators & Binary Truth Tables

Bitwise operators perform manipulations on individual bits of integer data types.

1.1 Truth Tables for Binary Bitwise Operators (`AND` , `OR` , `XOR`)

Bit A	Bit B	AND (`A & B`)	OR (`A | B`)	XOR (`A ^ B`)
0	0	**0**	**0**	**0**
0	1	**0**	**1**	**1**
1	0	**0**	**1**	**1**
1	1	**1**	**1**	**0**

Key Conceptual Rules:

- **Bitwise AND (`&`)**: Output bit is `1` **only if both input bits are 1**. Used for clearing or inspecting specific bits.
- **Bitwise OR (`|`)**: Output bit is `1` **if at least one input bit is 1**. Used for setting/enabling specific bits.
- **Bitwise XOR (`^`)**: Output bit is `1` **if input bits are different** (one `0` and one `1`). Used for toggling bits, detecting parity, and finding non-repeating elements.

1.2 Truth Table for Unary Bitwise NOT (`~`)

The bitwise NOT operator (`~`) is a unary operator that inverts every single bit of its operand (flips `0` to `1` and `1` to `0`).

Input Bit A	Bitwise NOT (`~A`)
0	**1**
1	**0**

Operational Properties:

- **Double Inversion**: `~(~A) = A`
- **Identity Laws with Masks**:
- $A \ \& \ 1 = A$, $A \ \& \ 0 = 0$
- $A \ | \ 0 = A$, $A \ | \ 1 = 1$
- $A \ \hat{} \ 0 = A$, $A \ \hat{} \ A = 0$, $A \ \hat{} \ 1 = \sim A$

---

2. Bit Shift Operators & Signed Shift Mechanics

Bit shift operators shift the bit pattern of an integer operand left or right by a specified number of bit positions.

2.1 Left Shift Operator (`x << n`)

Left shift pushes all bits to the left by n positions. The vacated rightmost positions are filled with zeros (`0`).

$$\text{Formula: } x \ll n = x \times 2^n$$

Worked Binary Trace: $13 \ll 2$ (in 8-bit binary)

- Binary representation of 13: `0000 1101`
- Shift left by 2 positions:
- Original: `0000 1101`
- Left Shift: `0011 0100`
- Value evaluation: $00110100_2 = 32 + 16 + 4 = \mathbf{52}$
- Verification: $13 \times 2^2 = 13 \times 4 = \mathbf{52}$

2.2 Right Shift Operators (`>>` vs `>>>`)

There are two distinct types of right shift operations depending on how vacated leftmost bits are filled:

1. **Arithmetic Right Shift (`x >> n`)**:
2. Preserves the sign bit (MSB). Vacated leftmost bits are filled with copies of the **Sign Bit** (1 for negative numbers, 0 for positive numbers).

3. Performs integer division by 2^n with floor rounding: $$x \gg n = \left\lfloor \frac{x}{2^n} \right\rfloor$$

4. **Logical Right Shift (`x >>> n`)**:

5. Zero-fill right shift. Always fills vacated leftmost bits with **zeros (`0`)**, regardless of whether the original number is positive or negative.

Worked Binary Traces:

Example 1: Positive Arithmetic Right Shift ($52 \gg 2$)

- Binary 52 (8-bit): `0011 0100`

- Shift right by 2 (fill left with sign bit `0`):
- Original: `0011 0100`
- Shifted: `0000 1101`
- Result: $00001101_2 = \mathbf{13}$ ($\lfloor 52 / 4 \rfloor = 13$).

Example 2: Negative Arithmetic Right Shift ($-16 \gg 2$)

- -16 in 8-bit 2's complement binary: `1111 0000`
- Shift right by 2 (fill left with sign bit `1`):
- Original: `1111 0000`
- Shifted: `1111 1100`
- Value of `1111 1100` in 2's complement: $\sim(11111100) + 1 = 00000011 + 1 = 4 \implies \mathbf{-4}$.
- Verification: $\lfloor -16 / 4 \rfloor = \mathbf{-4}$.

Example 3: Negative Logical Right Shift ($-16 \gg\gg 2$)

- -16 in 8-bit binary: `1111 0000`
- Shift right by 2 (fill left with zeros `0`):
- Shifted: `0011 1100`
- Unsigned Decimal Value: $32 + 16 + 8 + 4 = \mathbf{60}$.

---

3. Two's Complement Math & Binary Representation

Computers represent signed integers using **Two's Complement** notation because it allows addition and subtraction to be performed with the exact same hardware circuit.

3.1 Two's Complement Formula & Conversion Steps

To calculate the Two's Complement (negative representation) of an integer x: $$\text{Two's Complement}(x) = (\sim x) + 1 = -x$$

Step-by-Step Worked Conversion: Finding 8-Bit Binary Representation of -6

1. Write positive integer 6 in 8-bit binary: $$6_{10} = \mathbf{0000\ 0110}_2$$
2. Invert all bits (1's complement): $$\sim(0000\ 0110) = \mathbf{1111\ 1001}_2$$
3. Add 1 to the result: $$1111\ 1001 + 1 = \mathbf{1111\ 1010}_2$$
4. Therefore, -6 in 8-bit binary is `1111 1010`.

3.2 The Bitwise NOT Identity: $\sim x = -(x + 1)$

Because $-x = \sim x + 1$, rearranging the equation gives: $$\sim x = -x - 1 = -(x + 1)$$

Worked Examples:

1. `~ 9`: $$\sim 9 = -(9 + 1) = \mathbf{-10}$$
2. *Binary trace (8-bit)*: $9 = 00000001_2 \implies \sim 9 = 11110110_2$.

3. In 2's complement, $11110110_2 = -(00001001_2 + 1) = -10_{10}$.

4. `~ (-15)` : $$\sim (-15) = -((-15) + 1) = -(-14) = \mathbf{14}$$

5. *Binary trace (8-bit)*: $-15 = 11110001_2 \implies \sim(-15) = 00001110_2 = 14_{10}$.

---

4. Essential Bitwise Masking Shortcuts & Worked Examples

Bit masking involves using logical operations with a crafted binary mask to inspect, set, clear, or toggle specific bit positions.

4.1 Bitwise Shortcuts Summary Matrix

Objective	Pseudocode / Expression	Mathematical / Logical Effect
Check Even / Odd	`x & 1`	`0` if Even, `1` if Odd (inspects LSB)
Set k-th Bit	`x | (1 << k)`	Forces bit at index k to become `1`
Clear k-th Bit	`x & ~(1 << k)`	Forces bit at index k to become `0`
Toggle k-th Bit	`x ^ (1 << k)`	Flips bit at index k (`0` $\to$ `1` , `1` $\to$ `0`)
Check k-th Bit	`(x & (1 << k)) != 0`	Returns `TRUE` if bit k is set
Clear Lowest Set Bit	`x & (x - 1)`	Flips rightmost `1` bit to `0`
Isolate Lowest Set Bit	`x & (-x)`	Creates mask with only the rightmost `1` bit
Check Power of 2	`(x > 0) && ((x & (x - 1)) == 0)`	Returns `TRUE` if x is a power of 2 (2^p)
Swap Two Integers	`a = a ^ b; b = a ^ b; a = a ^ b`	Swaps values of `a` and `b` without extra variable
Turn On Rightmost 0 Bit	`x | (x + 1)`	Changes rightmost `0` bit to `1`

4.2 Step-by-Step Worked Traces of Key Bit Tricks

Worked Example 1: Clear Lowest Set Bit (`x & (x - 1)`)

- **Problem**: Evaluate `x & (x - 1)` for $x = 12$.
- **Step 1**: $x = 12_{10} = \mathbf{0000\ 1100}_2$
- **Step 2**: $x - 1 = 11_{10} = \mathbf{0000\ 1011}_2$
- **Step 3**: Perform Bitwise AND: ```text 0000 1100 (12) & 0000 1011 (11)

```
0000 1000  (8)
```

``` - **Result**: $8_{10}$. Notice that the lowest set bit (bit at position 2 with value 4) was cleared to 0.

---

**Worked Example 2: Isolate Lowest Set Bit ( `x & (-x)` )**

- **Problem**: Find the lowest set bit mask for $x = 20$.
- **Step 1**: $x = 20_{10} = \mathbf{0001\ 0100}_2$
- **Step 2**: Compute $-x$ using 2's complement ($\sim x + 1$):
- $\sim x = \sim(0001\ 0100) = 1110\ 1011$
- $-x = 1110\ 1011 + 1 = \mathbf{1110\ 1100}_2$
- **Step 3**: Perform Bitwise AND ( `x & -x` ): ```text 0001 0100 (20) & 1110 1100 (-20)

---

```
0000 0100 (4)
```

``` - **Result**: $4_{10}$. Isolates bit position 2 ($2^2 = 4$).

Worked Example 3: XOR Swap Algorithm (`a = a ^ b; b = a ^ b; a = a ^ b`)

- **Problem**: Swap $a = 9$ and $b = 5$ without a temporary variable.
- **Initial Values**: $a = 9_{10} = \mathbf{1001}_2$, $b = 5_{10} = \mathbf{0101}_2$.
- **Step 1**: $a = a \ \hat{} \ b$ ```text 1001 (9) ^ 0101 (5)

```
1100 (12) -> New value of 'a'
```

`- **Step 2**: $b = a \ \hat{} \ b$` text 1100 (12) ^ 0101 (5)

```
1001 (9)  -> New value of 'b' (b now equals original a!)
```

`- **Step 3**: $a = a \ \hat{} \ b$` text 1100 (12) ^ 1001 (9)

```
0101 (5)  -> New value of 'a' (a now equals original b!)
```

``` - **Final Result**: $a = 5$, $b = 9$. Swapping successfully accomplished!

---

**Worked Example 4: Counting Set Bits (Brian Kernighan's Algorithm)**

Brian Kernighan's algorithm repeatedly clears the lowest set bit using `x & (x - 1)` until `x` becomes 0.
```

```
Function CountSetBits(Integer x)
    Integer count = 0
    WHILE (x > 0) DO
        SET x = x & (x - 1)
        SET count = count + 1
    END WHILE
    RETURN count
End Function
```

Trace for $x = 23_{10} = 0001\ 0111_2$:

- Iteration 1: $x = 23 \ \& \ 22 = 00010111_2 \ \& \ 00010110_2 = 00010110_2 = 22$, `count` = 1
- Iteration 2: $x = 22 \ \& \ 21 = 00010110_2 \ \& \ 00010101_2 = 00010100_2 = 20$, `count` = 2
- Iteration 3: $x = 20 \ \& \ 19 = 00010100_2 \ \& \ 00010011_2 = 00010000_2 = 16$, `count` = 3
- Iteration 4: $x = 16 \ \& \ 15 = 00010000_2 \ \& \ 00001111_2 = 00000000_2 = 0$, `count` = 4
- **Final Output**: `count = 4` (since $23_{10} = 16 + 4 + 2 + 1$ has 4 set bits).

---

5. Placement Exam Tricky Practice Problems

Problem 1: Code Evaluation

```
Integer a = 14, b = 11, c = 0
c = (a & b) + (a ^ b)
PRINT c
```

- **Trace**:
- $a = 14 = 1110_2$, $b = 11 = 1011_2$
- $a \ \& \ b = 1110 \ \& \ 1011 = 1010_2 = 10$
- $a \ \hat{} \ b = 1110 \ \hat{} \ 1011 = 0101_2 = 5$
- $c = 10 + 5 = \mathbf{15}$
- **Mathematical Identity**: Note that for any two integers a and b: $$(a \ \& \ b) + (a \ \hat{} \ b) = a \ | \ b$$ $$(a \ \& \ b) + (a \ | \ b) = a + b$$

Problem 2: Bit Shift Pseudocode Tracing

```
Integer p = 7, q = 3
Integer result = (p << q) - (p >> 1)
PRINT result
```

- **Trace**:
- $p \ll q = 7 \ll 3 = 7 \times 2^3 = 7 \times 8 = 56$
- $p \gg 1 = 7 \gg 1 = \lfloor 7 / 2 \rfloor = 3$
- $\text{result} = 56 - 3 = \mathbf{53}$

---

6. CoCubes High-Frequency Bitwise PYQ Traces (Web-Researched GFG & Sanfoundry Sets)

PYQ 6.1: CoCubes XOR Inequality Branching Trap

```
Integer a = 6, b = 8, c = 15
If ((b ^ a) < a)
    b = a
End If
Print a + b + c
```

Step-by-Step Binary State Table:

Expression	Binary Conversion	Operation	Resulting Value	Decision / Action
`b ^ a`	$8 = 1000_2, 6 = 0110_2$	$1000_2 \oplus 0110_2 = 1110_2$	**14**	Evaluated condition: `14 < 6`
`(14 < 6)`	-	Relational Comparison	**FALSE**	`If` block is skipped (`b` remains 8)
`a + b + c`	$6 + 8 + 15$	Integer Addition	**29**	Final Output Printed

- **Final Output**: `29`
- **Exam Trap Warning**: Candidates often miscalculate `8 ^ 6` as `2` thinking `1000 ^ 0110` zeroes out bits, leading them to execute `b = a` and get `6 + 6 + 15 = 27` (a classic wrong option on CoCubes!).

PYQ 6.2: Sanfoundry Bitwise NOT & Shift Masking

```
Integer x = 5, y = 0
y = ((~x) << 2) ^ (x >> 1)
Print y
```

Binary Trace & State Table (using 8-bit signed integer representation):

Step	Operand / Expression	Binary State	Decimal Value	Notes
1	`x`	`0000 0101`	**5**	Original Value
2	`~x`	`1111 1010`	**-6**	Bitwise NOT identity $\sim x = -(x + 1) = -6$
3	`(~x) << 2`	`1110 1000`	**-24**	Shift left by 2 ($11111010_2 \ll 2 = 11101000_2$)

Step	Operand / Expression	Binary State	Decimal Value	Notes
4	`x >> 1`	`0000 0010`	**2**	Arithmetic Right Shift ($\lfloor 5 / 2 \rfloor = 2$)
5	`y = (-24) ^ 2`	`1110 1000 ^ 0000 0010` = `1110 1010`	**-22**	11101010_2 in 2's complement $= -22$

- **Final Output**: `-22`

PYQ 6.3: GFG Bitwise Accumulator Loop with Mask Inversion

```
Integer a = 12, b = 5, sum = 0
While (b > 0)
    If ((a & 1) != 0)
        sum = sum + (a ^ b)
    Else
        sum = sum + (a & (~b))
    End If
    a = a >> 1
    b = b >> 1
End While
Print sum
```

Step-by-Step State Table Trace:

Iteration	Initial `a`	Initial `b`	Condition `(a & 1) != 0`	Branch	Calculation	Result	Updated `sum`	`a >> 1`	`b >> 1`
1	12 (`1100`)	5 (`0101`)	$12 \& 1 = 0 \implies$ **FALSE**	`Else`	$a \& (\sim b) = 1100_2 \& (\sim 0101_2 = 1010_2) = 1000_2$	**8**	$0 + 8 = \mathbf{8}$	6 (`0110`)	2 (`0010`)
2	6 (`0110`)	2 (`0010`)	$6 \& 1 = 0 \implies$ **FALSE**	`Else`	$a \& (\sim b) = 0110_2 \& (\sim 0010_2 = 1101_2) = 0100_2$	**4**	$8 + 4 = \mathbf{12}$	3 (`0011`)	1 (`0001`)
3			$3 \& 1 = 1$	`If`	$a \hat{} b = 0011_2$	**2**	$12 + 2 = \mathbf{14}$		

Iteration	Initial `a`	Initial `b`	Condition `(a & 1) != 0`	Branch	Calculation	Result	Updated `sum`	`a >> 1`	`b >> 1`
	3 (`0011`)	1 (`0001`)	\implies$ **TRUE**		\ \hat{} \ 0001_2 = 0010_2$			1 (`0001`)	0 (`0000`)
End	1	0	$0 > 0 \implies$ **FALSE**	-	Loop Terminates	-	**14**	-	-

- **Final Output**: `14`