

Bitwise Operator Important Questions

1.Addition using bitwise operator

```
#include<stdio.h>
int sum(int a,int b)
{
    while(b!=0)
    {
        int data=a&b;
        a=a^b;
        b=data<<=1;
    }
    return a;
}
void main()
{
    int a=0,b=0;
    printf("enter the num:");
    scanf("%d%d",&a,&b);
    printf("sum:%d",sum(a,b));
}
```

Example

1. Input:

- a = 4
- b = 5

2. Binary representation:

- a = 100 (binary)
- b = 101 (binary)

3. Inside the sum function:

Iteration 1:

a = 100 (4)
b = 101 (5)

$\text{data} = a \& b = 100 \& 101 = 100 \text{ (4)}$
 $a = a \wedge b = 100 \wedge 101 = 001 \text{ (1)}$
 $b = \text{data} \ll 1 = 100 \ll 1 = 1000 \text{ (8)}$

Iteration 2:

$a = 001 \text{ (1)}$
 $b = 1000 \text{ (8)}$
 $\text{data} = a \& b = 001 \& 1000 = 000 \text{ (0)}$
 $a = a \wedge b = 001 \wedge 1000 = 1001 \text{ (9)}$
 $b = \text{data} \ll 1 = 000 \ll 1 = 0000 \text{ (0)}$

Iteration 3:

$a = 1001 \text{ (9)}$
 $b = 0000 \text{ (0)}$
 $\text{data} = a \& b = 1001 \& 0000 = 0000 \text{ (0)}$
 $a = a \wedge b = 1001 \wedge 0000 = 1001 \text{ (9)}$
 $b = \text{data} \ll 1 = 0000 \ll 1 = 00000 \text{ (0)}$

End of loop:

4. Return:

- The function returns **a**, which is 9.

5. Output:

- The program prints "sum: 9".

2. Multiplication using bitwise operator

```
#include<stdio.h>
void main()
{
    int a=0,b=0,res=0;;
```

```

printf("enter two nums:");
scanf("%d%d",&a,&b);
while(b!=0)
{
if(b&1)
{
res+=a;
}
a<<=1;
b>>=1;
}
printf("multiplication of two num:%d",res);
}

```

Step 1: Initialize variables.

- a = 4 (0100 in binary)
- b = 6 (0110 in binary)
- res = 0

Step 2: Enter the loop.

- As long as b is not 0, we continue the loop.

Step 3: Check if the least significant bit (LSB) of b is 1.

- In the first iteration, the LSB of b is 0. So, no action is taken.

Step 4: Left shift by 1.

- After left shifting a, it becomes 1000 (8 in decimal).

Step 5: Right shift b by 1.

- After right shifting b, it becomes 0011 (3 in decimal).

Step 6: Check if the least significant bit (LSB) of b is 1.

- In the second iteration, the LSB of b is 1.

Step 7: Add a to the result.

- As the LSB of b is 1, we add the current value of a (which is 8) to the result. So, $\text{res} = 8$.

Step 8: Left shift by 1.

- After left shifting a, it becomes 10000 (16 in decimal).

Step 9: Right shift b by 1.

- After right shifting b, it becomes 0001 (1 in decimal).

Step 10: Check if the least significant bit (LSB) of b is 1.

- In the third iteration, the LSB of b is 1.

Step 11: Add a to the result.

- As the LSB of b is 1, we add the current value of a (which is 16) to the result. So, $\text{res} = 8 + 16 = 24$.

Step 12: Left shift by 1.

- After left shifting a, it becomes 100000 (32 in decimal).

Step 13: Right shift b by 1.

- After right shifting b, it becomes 0000 (0 in decimal).

Step 14: Exit the loop.

- As b is now 0, we exit the loop.

Step 15: Output the result.

- The result of multiplying 4 and 6 using bitwise operations is 24.

The program computes the multiplication of 4 and 6 as 24.

3. Subtraction using bitwise operator

```
#include<stdio.h>

int sub(int a,int b)

{
    while(b!=0)

    {
        int data=(~a)&(b);

        a=a^b;

        b=data<<=1;

    }

    return a;
}

void main()

{
    int a=0,b=0;

    printf("enter the numbers:");

    scanf("%d%d",&a,&b);

    printf("sub:%d",sub(a,b));

}
```

Example

- $a = 5$ (binary: 0101)
- $b = 3$ (binary: 0011)

Now, let's walk through the `sub()` function step by step:

1. Iteration 1:

- a : 0101
- b : 0011
- Inside the loop:
 - $data = (\sim a) \& b$: $(\sim 0101) \& 0011 = 1010 \& 0011 = 0010$
 - $a = a \wedge b$: $0101 \wedge 0011 = 0110$
 - $b = data \ll 1$: $0010 \ll 1 = 0100$
- End of iteration:
 - a : 0110
 - b : 0100

2. Iteration 2:

- a : 0110
- b : 0100
- Inside the loop:
 - $data = (\sim a) \& b$: $(\sim 0110) \& 0100 = 1001 \& 0100 = 0000$
 - $a = a \wedge b$: $0110 \wedge 0100 = 0010$
 - $b = data \ll 1$: $0000 \ll 1 = 0000$
- End of iteration:
 - a : 0010
 - b : 0000

3. End of loop:

- The loop ends when b becomes zero.

So, the final result of the subtraction (a) is 0010, which is 2 in decimal.

Therefore, when you input $a = 5$ and $b = 3$, the output of the subtraction is 2.

4. Division using bitwise operator

```
#include<stdio.h>
void main()
{
    int a=0,b=0,ans=0;
    printf("enter two nums:");
    scanf("%d%d",&a,&b);
    while(a>=b)
    {
        a-=b;
        ans++;
    }
    printf("division of two num:%d",ans);
}
```

Example

Step 1: Initialize variables.

- a = 32
- b = 4
- ans = 0

Step 2: Enter the loop.

- As long as a is greater than or equal to b, we continue the loop.

Step 3: Subtract b from a.

- Subtracting 4 from 32, we get 28.
- ans = 1 (incremented by 1 because we successfully subtracted once)

Step 4: Enter the loop again.

- a is still greater than or equal to b.

Step 5: Subtract b from a.

- Subtracting 4 from 28, we get 24.
- $\text{ans} = 2$ (incremented by 1 again)

Step 6: Enter the loop again.

- a is still greater than or equal to b.

Step 7: Subtract b from a.

- Subtracting 4 from 24, we get 20.
- $\text{ans} = 3$

Step 8: Enter the loop again.

- a is still greater than or equal to b.

Step 9: Subtract b from a.

- Subtracting 4 from 20, we get 16.
- $\text{ans} = 4$

Step 10: Enter the loop again.

- a is still greater than or equal to b.

Step 11: Subtract b from a.

- Subtracting 4 from 16, we get 12.
- $\text{ans} = 5$

Step 12: Enter the loop again.

- a is still greater than or equal to b.

Step 13: Subtract b from a.

- Subtracting 4 from 12, we get 8.
- $\text{ans} = 6$

Step 14: Enter the loop again.

- a is still greater than or equal to b.

Step 15: Subtract b from a.

- Subtracting 4 from 8, we get 4.
- ans = 7

Step 16: Enter the loop again.

- a is still greater than or equal to b.

Step 17: Subtract b from a.

- Subtracting 4 from 4, we get 0.
- ans = 8

Step 18: Exit the loop.

- a is now less than b ($0 < 4$), so we exit the loop.

Step 19: Output the result.

- The result of dividing 32 by 4 using subtraction is 8.

5 .Without using comparison operator

```
#include<stdio.h>
int fun(int x,int y)
{
    return !(x^y);
}
void main()
{
    int x=0,y=0;
    printf("enter two num:");
    scanf("%d%d",&x,&y);
```

```

    if(fun(x,y))
    {
        printf("equal");
    }
    else
    {
        printf("not equal");
    }
}

```

Example - 1

Step 1: Initialize variables.

- $x = 5$
- $y = 6$

Step 2: Call the function fun(x, y).

- Inside the function fun(), it computes $!(x \wedge y)$.
- \wedge is the XOR operator. So, $x \wedge y$ gives $011 \wedge 110$ which is 101.
- $!$ is the logical NOT operator. So, $!101$ is 0 (false).
- The function returns 0 (false).

Step 3: Check the returned value.

- Since the returned value is 0 (false), the condition in the if statement evaluates to false.

Step 4: Execute the else block.

- The program prints "not equal".

So, with inputs 5 and 6, the program outputs "not equal".

Example -2

1. Initialization:

- We initialize two variables `x` and `y` with values obtained from user input. In this case, both `x` and `y` are initialized to `4`.

2. Function Call:

- The function `fun(x, y)` is called.
- This function checks whether `x` and `y` are equal or not by using the XOR operator (`^`).
- The XOR operator returns `1` if the bits are different and `0` if the bits are the same.

3. XOR Operation:

- For both `x = 4` and `y = 4`, their binary representations are `100`.
- When we perform the XOR operation between them, we get `000`.
- The logical NOT (!) operator negates this result. So, `!000` becomes `1`.

4. Condition Check:

- The result of the `fun(x, y)` function call is `1`.
- The `if` statement checks if the result is true (non-zero) or false (zero).
- Since the result is `1` (true), the condition in the `if` statement evaluates to true.

5. Output:

- As the condition in the `if` statement is true, the program executes the `if` block.
- It prints "equal" because `4` and `4` are indeed equal.

6. Bitwise operator swapping

```
#include<stdio.h>
void main()
{
    int a=0,b=0;
    printf("enter two num:");
    scanf("%d%d",&a,&b);
```

```

printf("\n Before swapping :%d\t%d",a,b);
a=a^b;
b=a^b;
a=a^b;
printf("\n After swapping :%d\t%d",a,b);
}

```

Example

enter the values for a and b:24 56

value of a=24 and b=56 before swap

value of a=56 and b=24 after swap

Explanation:

a=24 binary equivalent of 24 =011000

b=56 binary equivalent of 56= 111000

$a = a \oplus b = 100000$

$b = a \oplus b = 100000 \oplus 111000 = 011000$

$a = a \oplus b = 100000 \oplus 011000 = 111000$

Now a=111000 decimal equivalent =56

b= 011000 decimal equivalent = 24

7. Odd or even using bitwise operator

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

```
void main()
```

```
{
```

```
    int n=0;
```

```
    printf("Enter the number:");
```

```
    scanf("%d",&n);
```

```
    if((n&1)==0)
```

```
    {
```

```
        printf("even number:%d",n);
```

```
    }
```

```
    else
```

```
    {
```

```

        printf("odd number:%d",n);
    }

}

```

Now, let's go through the steps with the input of 5 (which is 101 in binary):

1. **Input:** User inputs 5.
2. **Binary representation of 5:** 5 in binary is 101.
3. **Check even or odd:**
 - $(n \& 1)$: This performs a bitwise AND operation between the binary representation of n and 1. Since 1 in binary is 000...0001, this operation effectively checks the least significant bit (LSB) of n . If the LSB is 0, the number is even; if it's 1, the number is odd.
 - For $n = 5$, which in binary is 101, the LSB is 1. Therefore, $(n \& 1)$ results in 1.
 - Since the result is 1, the condition $(n \& 1) == 0$ is false, indicating that n is odd.
4. **Output:** It prints "Odd number: 5" because the input number is odd.

Example 2:

1. **Input:** User inputs 8.
2. **Binary representation of 8:** 8 in binary is 1000.
3. **Check even or odd:**
 - $(n \& 1)$: This performs a bitwise AND operation between the binary representation of n and 1. Since 1 in binary is 000...0001, this operation effectively checks the least significant bit (LSB) of n . If the LSB is 0, the number is even; if it's 1, the number is odd.
 - For $n = 8$, which in binary is 1000, the LSB is 0. Therefore, $(n \& 1)$ results in 0.
 - Since the result is 0, the condition $(n \& 1) == 0$ is true, indicating that n is even.
4. **Output:** It prints "Even number: 8" because the input number is even.

8. SWAPPING NIBBLES

```
#include<stdio.h>
void main()
{
    int n=0,res=0;
    printf("enter the number:");
    scanf("%d",&n);
    res=(n&0xF0)>>4|(n&0x0F)<<4;
    printf("\n After swapping nibble :%d",res);
}
```

Example

Step 1: Input from User

enter the number: 15

Step 2: Binary Representation of 15

In binary: 15 = 0000 1111

Step 3: Performing Bitwise AND with 0xF0 (1111 0000)

$n \& 0xF0 = 0000\ 1111 \& 1111\ 0000 = 0000\ 0000$

This operation isolates the higher nibble of the input number n.

Step 4: Right Shifting by 4 Bits

$(0000\ 0000) \>> 4 = 0000\ 0000$

This step shifts the isolated higher nibble to the right by four positions.

Step 5: Performing Bitwise AND with 0x0F (0000 1111)

$$n \& 0x0F = 0000\ 1111 \& 0000\ 1111 = 0000\ 1111$$

This operation isolates the lower nibble of the input number n.

Step 6: Left Shifting by 4 Bits

$$(0000\ 1111) \ll 4 = 1111\ 0000$$

This step shifts the isolated lower nibble to the left by four positions.

Step 7: Performing Bitwise OR of Results

$$(0000\ 0000) | (1111\ 0000) = 1111\ 0000$$

This operation combines the shifted nibbles to form the result.

Step 8: Final Output

After swapping nibble: 240

9. swapping the single bit

```
#include<stdio.h>
void main()
{
    int n=0,p1=0,p2=0;
    printf("enter the number:");
    scanf("%d",&n);
    printf("enter the position 1:");
    scanf("%d",&p1);
    printf("enter the position 2:");
    scanf("%d",&p2);
    unsigned int a=(n>>p1)&1;
```



```

        unsigned int b=(n>>p2)&1;
        unsigned int c=a^b;
        c=(c<<p1)|(c<<p2);
        unsigned int result=c^n;
        printf("result:%d",result);
    }

```

Example

Step 1: Input from User

```

enter the number: 10
enter the position 1: 0
enter the position 2: 1

```

The user inputs the number 10 and position values $p1 = 0$, $p2 = 1$.

Step 2: Binary Representation of 10

In binary: $10 = 0000\ 1010$

Step 3: Extracting Bits at Positions $p1$ and $p2$

```

a = (n >> p1) & 1
  = (0000 1010 >> 0) & 1
  = 0000 1010 & 0000 0001
  = 0000 0000
  = 0

```

```

b = (n >> p2) & 1
  = (0000 1010 >> 1) & 1
  = 0000 0101 & 0000 0001
  = 0000 0001
  = 1

```

Here, we extracted the bits at positions $p1$ (0th position) and $p2$ (1st position).

Step 4: XOR Operation on Extracted Bits

$$\begin{aligned}c &= a \wedge b \\&= 0000\ 0000 \wedge 0000\ 0001 \\&= 0000\ 0001\end{aligned}$$

XOR operation gives 1 because the bits at p1 and p2 positions are different.

Step 5: Generating Mask to Flip Bits at p1 and p2 Positions

$$\begin{aligned}c &= (c \ll p1) \mid (c \ll p2) \\&= (1 \ll 0) \mid (1 \ll 1) \\&= 0000\ 0001 \mid 0000\ 0010 \\&= 0000\ 0011 \\&= 3\end{aligned}$$

We shift the result c to the left by p1 and p2 positions respectively and then combine them using bitwise OR.

Step 6: Flipping Bits at p1 and p2 Positions in the Original Number

$$\begin{aligned}\text{result} &= c \wedge n \\&= 0000\ 0011 \wedge 0000\ 1010 \\&= 0000\ 1001\end{aligned}$$

Finally, we XOR the generated mask (c) with the original number (n) to flip the bits at positions p1 and p2.

Step 7: Final Output

$$\text{result: } 9 \text{ (binary: } 0000\ 1001\text{)}$$

The program prints the result, which is 9 in binary representation

10 . Particular Bit set or not

```
#include<stdio.h>
void main()
{
    int n=0,pos=0;
    printf("enter the number:");
    scanf("%d",&n);
    printf("enter the position:");
    scanf("%d",&pos);
    if(n&(1<<pos))
    {
        printf("bit set");
    }
    else
    {
        printf("bit not set");
    }
}
```