LECTURE 02

BIT MANIPULATION

Big-O Coding

Website: www.bigocoding.com

Giới thiệu tổng quan

Bit viết tắt của chữ Binary Digit là đơn vị nhỏ nhất để biểu diễn thông tin trên máy tính. Một **bit** có thể nhận một trong hai giá trị 0 hoặc 1.

Ví dụ: Một biến "byte" trên máy tính có độ lớn là 8 bit.

a = 12

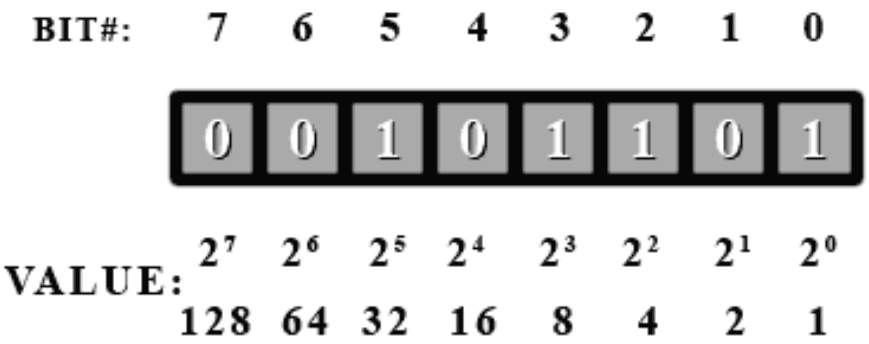
7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0

Biểu diễn số nguyên có dấu

Có nhiều cách được sử dụng để biểu diễn số nguyên có dấu trong máy tính.

Trong đó có 4 phương pháp chủ yếu:

- Phương pháp dấu lượng (Sign and magnitude)
- Phương pháp bù 1 (One's complement)
- Phương pháp bù 2 (Two's complement)
- Phương pháp số quá N (Excess-N)

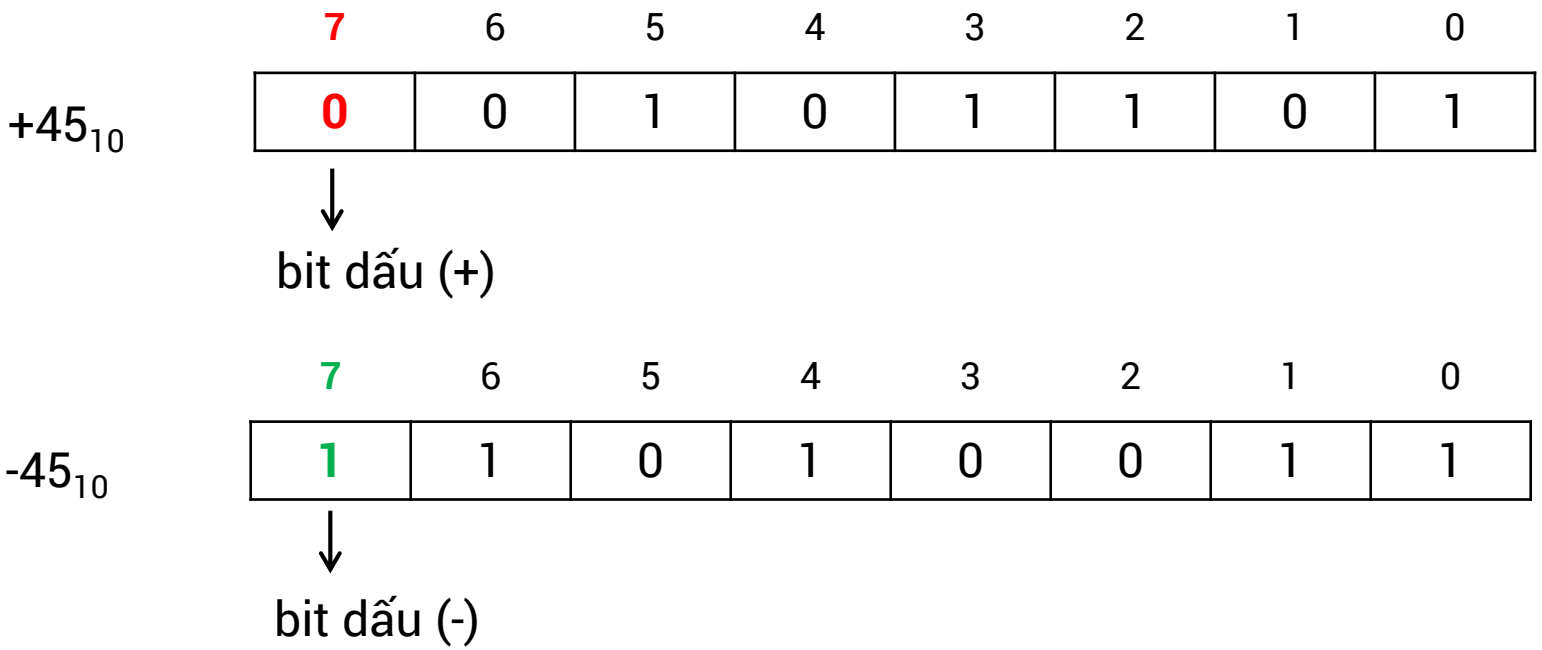


**** Lưu ý: Các máy tính hiện nay hầu hết đều sử dụng **phương pháp biểu diễn số bù 2.***

Phương pháp bù 2

Theo phương pháp này, bit cao nhất hay còn gọi là bit nằm bên trái cùng của byte được sử dụng làm bit dấu (là bit tượng trưng cho dấu của số – sign bit). Người ta quy ước: Nếu bit dấu là 0 thì số là số **dương**, nếu bit dấu là 1 thì số là số **âm**. Ngoài bit dấu này ra, các bit còn lại được dùng để biểu diễn độ lớn của số.

Ví dụ: biểu diễn số +45 và -45 hệ nhị phân theo mẫu 8 bit.



Biểu diễn số dạng bù 2

Biểu diễn số -45_{10} sang hệ nhị phân:

Bước 1: Xác định số nguyên 45 ở hệ thập phân được biểu diễn hệ nhị phân.

0	0	1	0	1	1	0	1

Bước 2: Đảo tất cả các bit lại.

1	1	0	1	0	0	1	0

Bước 3: Cộng thêm 1 vào kết quả thu được ở bước 2.

1	1	0	1	0	0	1	1

Bước 4: Kiểm tra lại bit dấu, để lấy kết quả cuối cùng có bị tràn số hay không.

1	1	0	1	0	0	1	1

Các thao tác trên bit

Tên phép toán	Ký hiệu
AND	&
OR	|
XOR	^
NOT	~
Shift Left (dịch trái)	<<
Shift Right (dịch phải)	>>
Lấy bit tại vị trí bất kỳ	(X >> k) & 1
Gán giá trị bit 0 tại vị trí bất kỳ	X & (~(1 << k))
Gán giá trị bit 1 tại vị trí bất kỳ	X | (1 << k)
Đảo giá trị bit tại vị trí bất kỳ	X ^ (1 << k)

*** *Lưu ý: X là số cần thao tác, k là vị trí.*

Phép toán AND

Big-O Orange

Kí hiệu: &

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

```
int A = 5;
int B = 6;
int C = A & B;
```

0	0	0	0	0	1	0	1
0	0	0	0	0	1	1	0
0	0	0	0	0	1	0	0

4

**** Lưu ý: chỉ biểu diễn 8 bit cuối của kiểu int 4 byte.*

Phép toán OR

Big-O Orange

Kí hiệu: |

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

```
int A = 5;
int B = 6;
int C = A | B;
```

0	0	0	0	0	1	0	1
0	0	0	0	0	1	1	0
0	0	0	0	0	1	1	1

7

Phép toán XOR

Big-O Orange

Kí hiệu: ^

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

```
int A = 5;
int B = 6;
int C = A ^ B;
```

0	0	0	0	0	1	0	1
0	0	0	0	0	1	1	0
0	0	0	0	0	0	1	1

3

Phép toán NOT

Kí hiệu: ~

A	~A
0	1
1	0

```
int A = 5;
int B = ~A;
```

0	0	0	0	0	1	0	1

1	1	1	1	1	0	1	0

-6

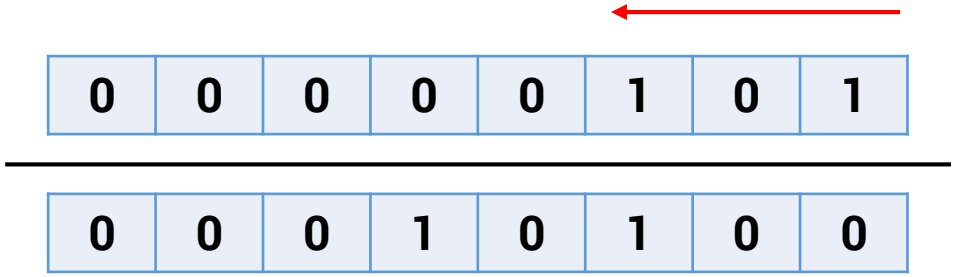
Phép dịch trái (Shift Left)

Kí hiệu: <<

Phép dịch trái tương đương với phép nhân cho 2^n.

Ví dụ: dịch trái 2 bits của A = 5.

```
int A = 5;
int B = A << 2;
```



Hoặc: $5 * 2^2 = 20$.

20

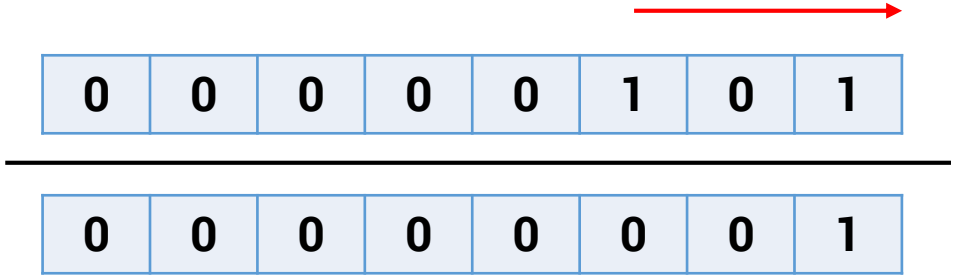
Phép dịch phải (Shift Right)

Kí hiệu: >>

Phép dịch phải tương đương với phép chia nguyên cho 2^n.

Ví dụ: dịch phải 2 bits của A = 5.

```
int A = 5;
int B = A >> 2;
```



Hoặc: $5 / 2^2 = 1$.

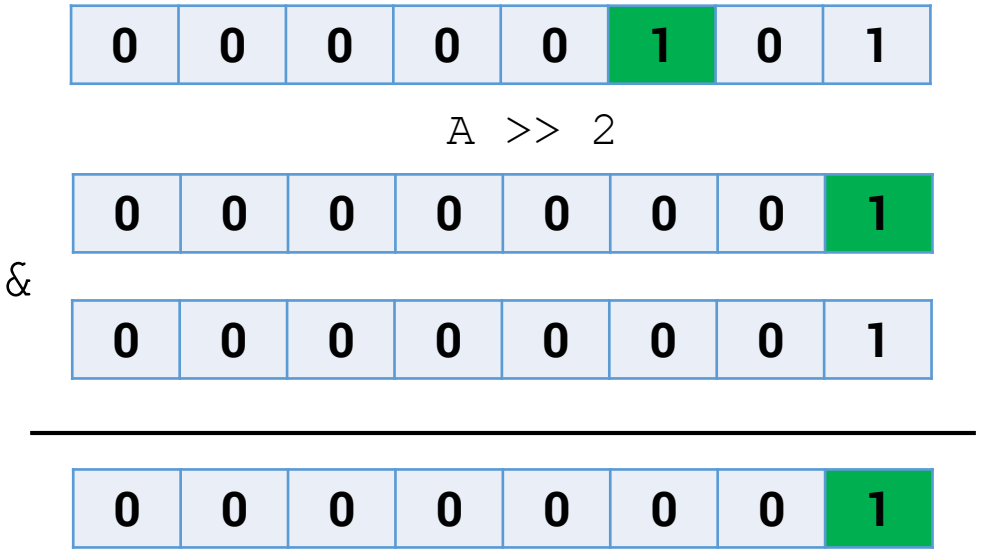
1

Lấy giá trị bit tại vị trí k của X (get bit)

Công thức: (X >> k) & 1

Ví dụ: Lấy bit 2 của A = 5.

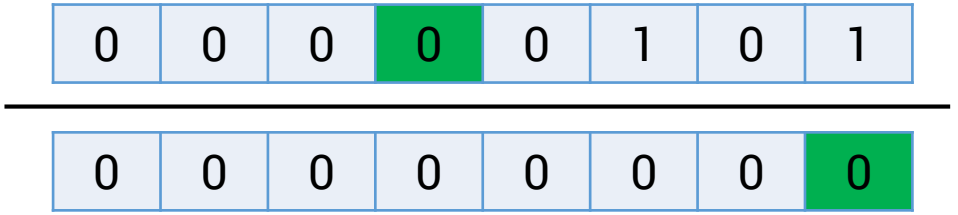
```
int A = 5;
int B = (A >> 2) & 1;
```



B = 1

Tương tự lấy bit 4 của A = 5.

```
int A = 5;
int C = (A >> 4) & 1;
```



C = 0

Gán giá trị bit tại vị trí k của X

Công thức:

- **Gán 0 (clear bit):** X & (~(1 << k))
- **Gán 1 (set bit):** X | (1 << k)

Ví dụ: Cho A = 5 gán 0 ở bit số 2.

0	0	0	0	0	1→0	0	1

```
int A = 5;
int B = A & (~(1 << 2));
```

0	0	0	0	0	0	0	1

1 << 2

0	0	0	0	0	1	0	0

~(1 << 2)

1	1	1	1	1	0	1	1

&

0	0	0	0	0	1	0	1

0	0	0	0	0	0	0	1

B = 1

Gán giá trị bit tại vị trí k của X

Công thức:

- **Gán 0 (clear bit):** X & (~(1 << k))
- **Gán 1 (set bit):** X | (1 << k)

Ví dụ: Cho A = 5 gán 1 ở bit số 3.

0	0	0	0	0→1	1	0	1

```
int A = 5;
int C = A | (1 << 3);
```

0	0	0	0	0	0	0	1

1 << 3

0	0	0	0	1	0	0	0

|

0	0	0	0	0	1	0	1

0	0	0	0	1	1	0	1

C = 13

Đảo giá trị bit tại vị trí k của X (flip bit)

Công thức: X ^ (1 << k)

Ví dụ: đảo bit 3 của A = 5.

0	0	0	0	0→1	1	0	1

```
int A = 5;
int B = A ^ (1 << 3);
```

0	0	0	0	0	0	0	1

1 << 3

0	0	0	0	1	0	0	0

^

0	0	0	0	0	1	0	1

0	0	0	0	1	1	0	1

B = 13

Bài toán minh họa 1

Find the Missing Number: Cho mảng một chiều các số từ 1 đến n, và bị thiếu đi một phần tử. Hãy tìm phần tử bị thiếu đó.

Ví dụ: Cho mảng một chiều.

0	1	2	3	4	5	6	7
1	2	3	4	6	7	8	9

→ 5 là phần tử bị thiếu.

Bài toán minh họa 1

Cách 1: Dùng công thức tính tổng:

- Tính tổng từ 1 ➔ n: $\text{sum} = \frac{n(n+1)}{2}$
- Tính tổng giá trị trong mảng: sum(array)

➔ Giá trị cần tìm = sum – sum(array)

0	1	2	3	4	5	6	7
1	2	3	4	6	7	8	9

sum = $\frac{9*(9+1)}{2}$ = 45

sum(array) = 40.

➔ Giá trị cần tìm = 45 – 40 = 5

Độ phức tạp: O(N)

Bài toán minh họa 1

Cách 2: Dùng công thức XOR:

- Tính XOR từ 1 ➔ n: xor = (1 ⊕ 2 ⊕ 3 ⊕ … ⊕ n)
- Tính XOR giá trị trong mảng: xor(array) = (1⊕ …⊕ n)

➔ Giá trị cần tìm = xor ⊕ xor(array)

0	1	2	3	4	5	6	7
1	2	3	4	6	7	8	9

xor = 1 ⊕ 2 ⊕ 3 ⊕ 4 ⊕ 5 ⊕ 6 ⊕ 7 ⊕ 8 ⊕ 9

xor(array) = 1 ⊕ 2 ⊕ 3 ⊕ 4 ⊕ 6 ⊕ 7 ⊕ 8 ⊕ 9

➔ Giá trị cần tìm = xor ⊕ xor(array) = 5

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Độ phức tạp: O(N)

Source Code Find the Missing Number

```cpp
#include <iostream>
using namespace std;
int getMissingNo(int a[], int n)
{
    int xor_number = 1;
    int xor_array = a[0];
    for (int i = 2; i <= n+1; i++)
        xor_number = xor_number ^ i;
    for (int i = 1; i < n; i++)
        xor_array = xor_array ^ a[i];
    return (xor_number ^ xor_array);
}
```


```cpp
int main()
{
    int a[] = {1, 2, 3, 4, 6, 7, 8, 9};
    int result = getMissingNo(a, 8);
    cout << result;
    return 0;
}
```

Source Code Find the Missing Number

```python
def getMissingNo(a, n):
    xor = 1
    xor_array = a[0]
    for i in range(2, n + 2):
        xor = xor ^ i
    for i in range(1, n):
        xor_array = xor_array ^ a[i]
    return xor ^ xor_array

if __name__ == "__main__":
    a = [1, 2, 3, 4, 6, 7, 8, 9]
    result = getMissingNo(a, 8)
    print(result)
```

Source Code Find the Missing Number

```java
public class Main {
    static private int getMissingNo(int a[], int n) {
    int xor = 1;
    int xor_array = a[0];
    for (int i = 2; i <= n + 1; i++)
        xor ^= i;
    for (int i = 1; i < n; i++)
        xor_array ^= a[i];
    return xor ^ xor_array;
    }

    public static void main(String[] args) {
        int a[] = {1, 2, 3, 4, 6, 7, 8, 9};
        int result = getMissingNo(a, 8);
        System.out.println(result);
        return;
    }
}
```

Cải tiến cách 2

Cách 2: Dùng công thức XOR (tiết kiệm chi phí 1 vòng lặp)

- **Tính XOR từ 1 ➔ n: xor = (1 ⊕ 2 ⊕ 3 ⊕ … ⊕ n)**

Thập phân	Nhị phân	Kết quả	Nhận xét
1	0001	0001	1
2	0010	0011	n+1
3	0011	0000	0
4	0100	0100	n
5	0101	0001	1
6	0110	0111	n+1
7	0111	0000	0
8	1000	1000	n
9	1001	0001	1
10	1010	1011	n+1
11	1011	0000	0
12	1100	1100	n

Source Code FMN cải tiến

```cpp
#include <iostream>
using namespace std;
int computeXOR(int n)
{
    //If n is a multiple of 4
    if (n % 4 == 0)
        return n;
    //If n % 4 gives remainder 1
    if (n % 4 == 1)
        return 1;
    //If n % 4 gives remainder 2
    if (n % 4 == 2)
        return n + 1;
    //If n % 4 gives remainder 3
    return 0;
}
```

Source Code FMN cải tiến

```cpp
int getMissingNo(int a[], int n)
{
    int xor_array = a[0];
    int xor_number = computeXOR(n + 1);
    for (int i = 1; i < n; i++)
        xor_array = xor_array ^ a[i];
    return (xor_number ^ xor_array);
}
```


```cpp
int main()
{
    int a[] = {1, 2, 3, 4, 6, 7, 8, 9};
    int result = getMissingNo(a, 8);
    cout << result;
    return 0;
}
```

Source Code FMN cải tiến

```python
def computeXOR(n):
    #If n is a multiple of 4
    if n % 4 == 0:
        return n
    #If n % 4 gives remainder 1
    if n % 4 == 1:
        return 1
    #If n % 4 gives remainder 2
    if n % 4 == 2:
        return n + 1
    #If n % 4 gives remainder 3
    return 0
```

Source Code FMN cải tiến

```python
def getMissingNo(a, n):
    xor_array = a[0]
    xor = computeXOR(n + 1)
    for i in range(1, n):
        xor_array = xor_array ^ a[i]
    return xor ^ xor_array
```

```python
if __name__ == "__main__":
    a = [1, 2, 3, 4, 6, 7, 8, 9]
    result = getMissingNo(a, 8)
    print(result)
```

Source Code FMN cải tiến

```java
public class Main {
    static private int computeXOR(int n) {
    //If n is a multiple of 4
    if (n % 4 == 0)
        return n;
    //If n % 4 gives remainder 1
    if (n % 4 == 1)
        return 1;
    //If n % 4 gives remainder 2
    if (n % 4 == 2)
        return n + 1;
    //If n % 4 gives remainder 3
        return 0;
    }
```

Source Code FMN cải tiến

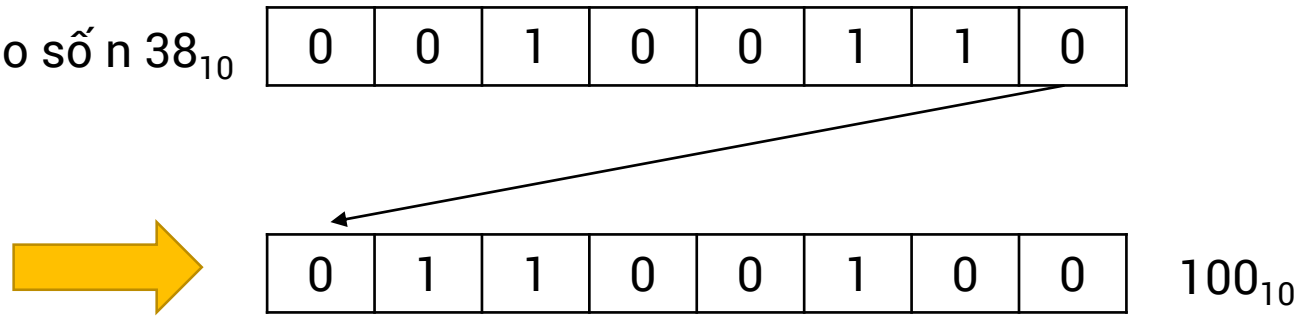
```
15.     static private int getMissingNo(int a[], int n) {
16.     int xor = computeXOR(n + 1);
17.     int xor_array = a[0];
18.     for (int i = 1; i < n; i++)
19.         xor_array ^= a[i];
20.     return xor ^ xor_array;
21.     }
```


```
22.     public static void main(String[] args) {
23.         int a[] = {1, 2, 3, 4, 6, 7, 8, 9};
24.         int result = getMissingNo(a, 8);
25.         System.out.println(result);
26.         return;
27.     }
28. }
```

Bài toán minh họa 2

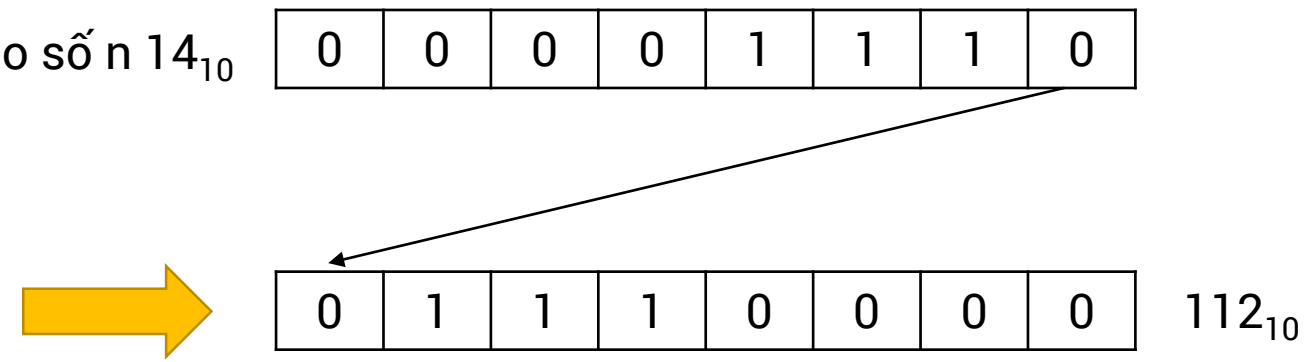
Reverse bits of the given number: Đảo ngược thứ tự dãy bit của một số nguyên dương (mẫu 8 bit).

Ví dụ 1: cho số n 38_{10}



100_{10}

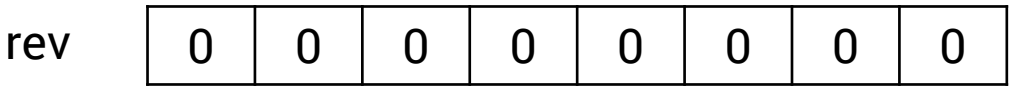
Ví dụ 2: cho số n 14_{10}



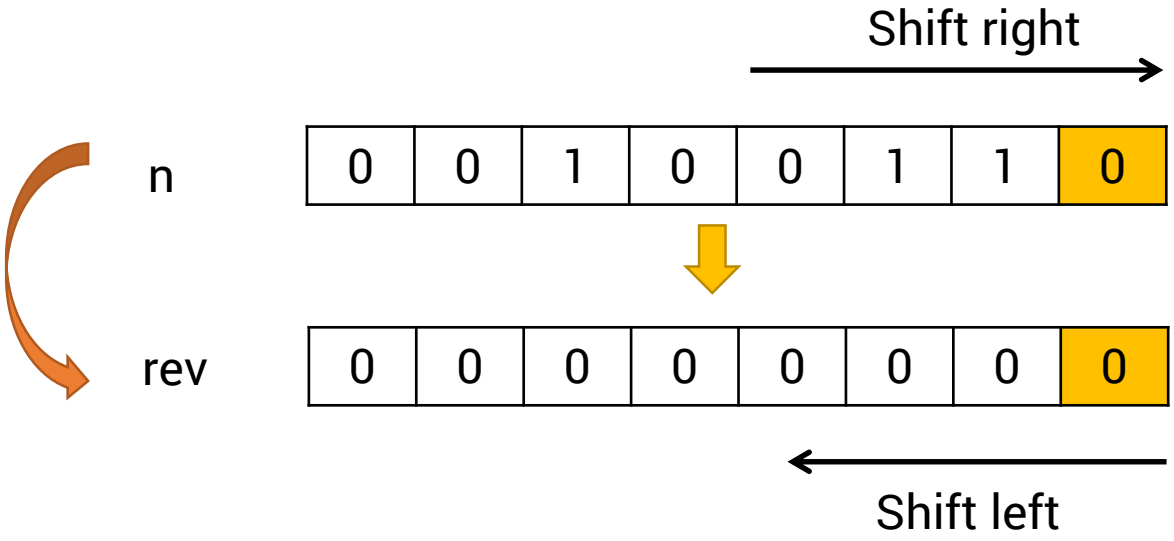
112_{10}

Ý tưởng thực hiện

Bước 1: Tạo ra số "kết quả" là số 0.



Bước 2: Lần lượt chép ngược toàn bộ bit của số n vào rev:



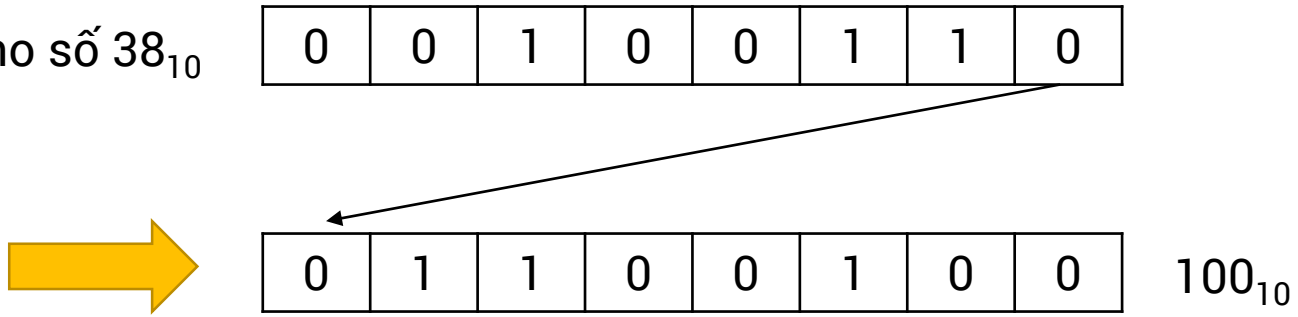
Bước 3: Dịch trái số bit còn lại nếu chưa đủ số bit mẫu.

Bước 4: Xuất kết quả cần tìm.

Độ phức tạp: **O(log(n))**

Bước 1: Tạo ra số kết quả

Đề bài: Cho số 38_{10}



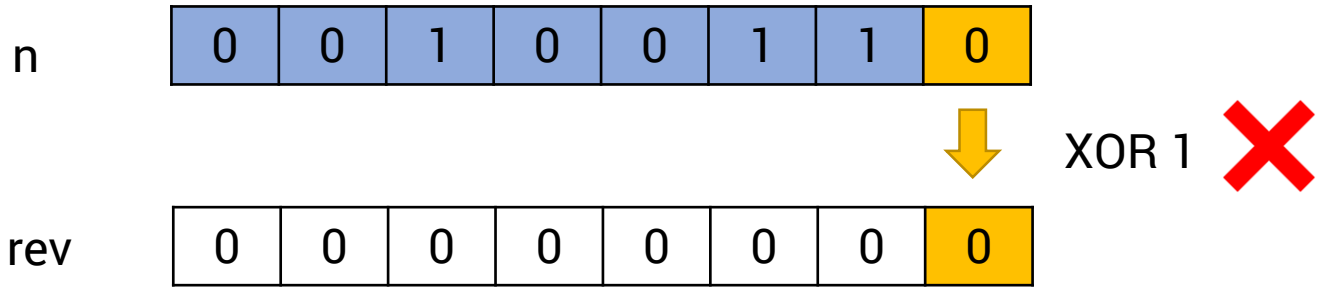
Tạo ra số "kết quả" là số 0.

rev	0	0	0	0	0	0	0	0

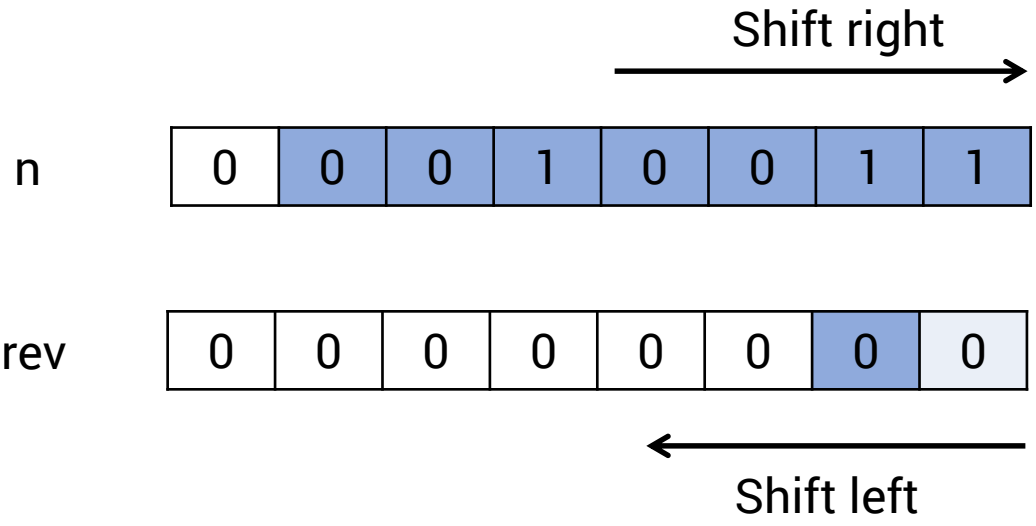
Khai báo biến số lượng bit, s = 8.

Bước 2: Chép số n vào rev (1)

Xét **n != 0** ➔ Chép bit đầu tiên của số n vào rev:



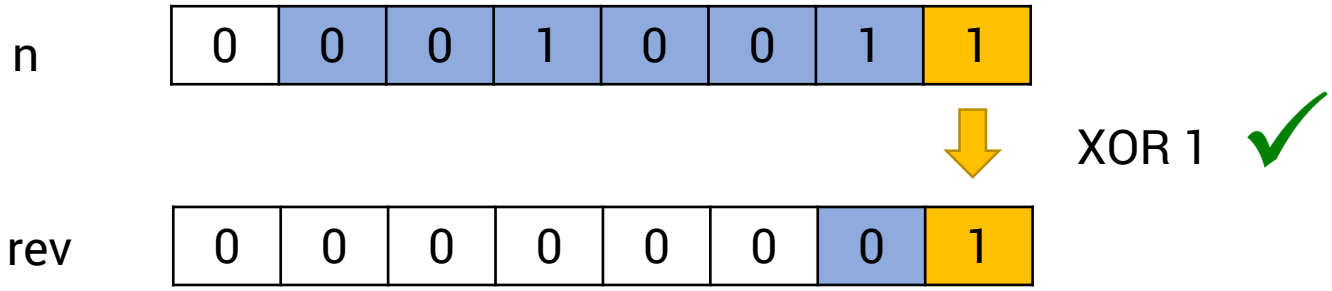
Dịch phải 1 bit số n và dịch trái 1 bit số rev:



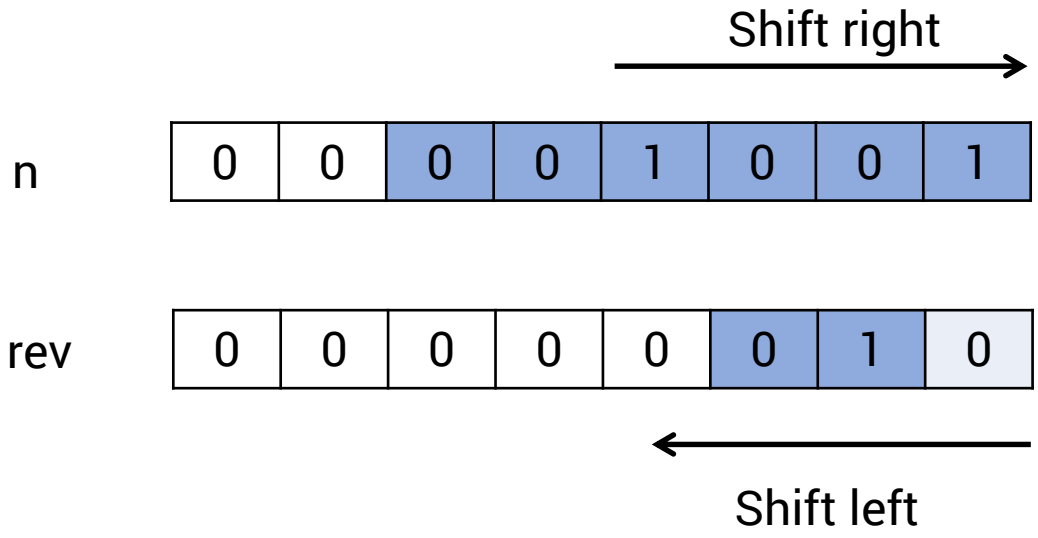
Giảm biến s = 7

Bước 2: Chép số n vào rev (2)

Xét **n != 0** ➔ Chép bit tiếp theo của số n vào rev:



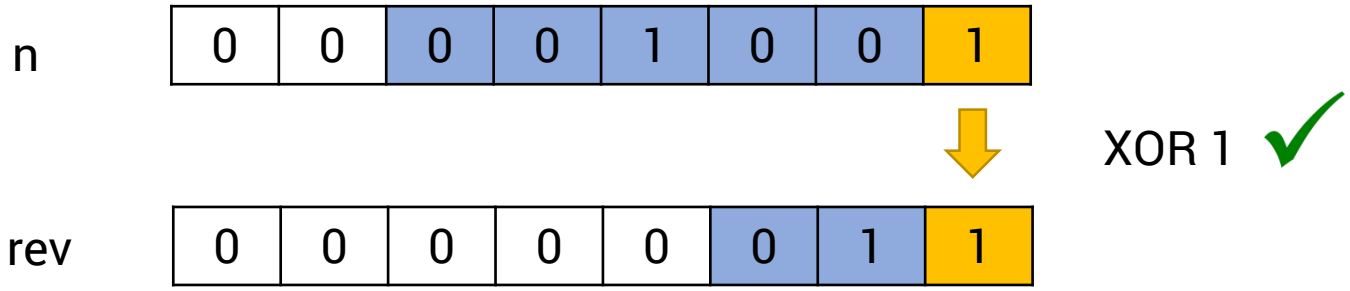
Dịch phải 1 bit số n và dịch trái 1 bit số rev:



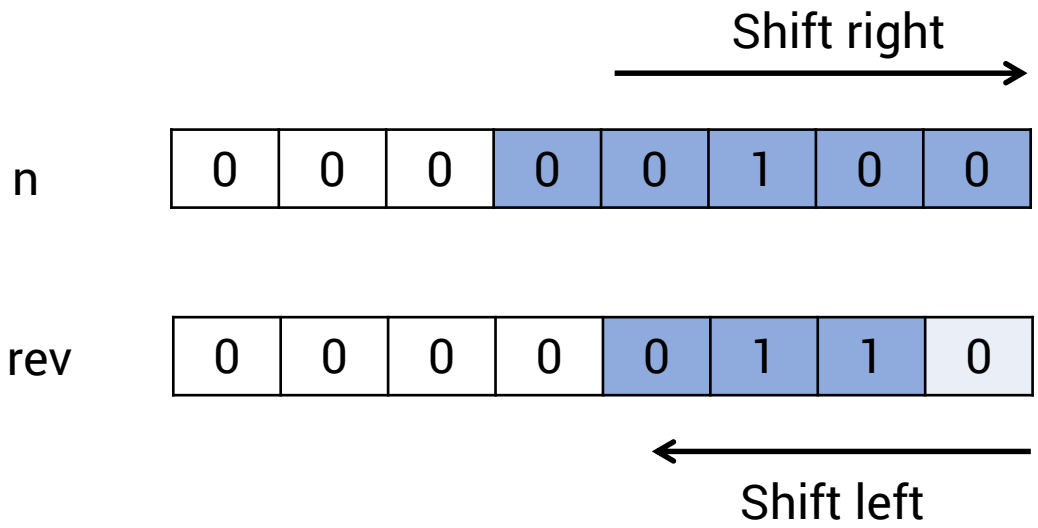
Giảm biến s = 6

Bước 2: Chép số n vào rev (3)

Xét **n != 0** ➔ Chép bit tiếp theo của số n vào rev:



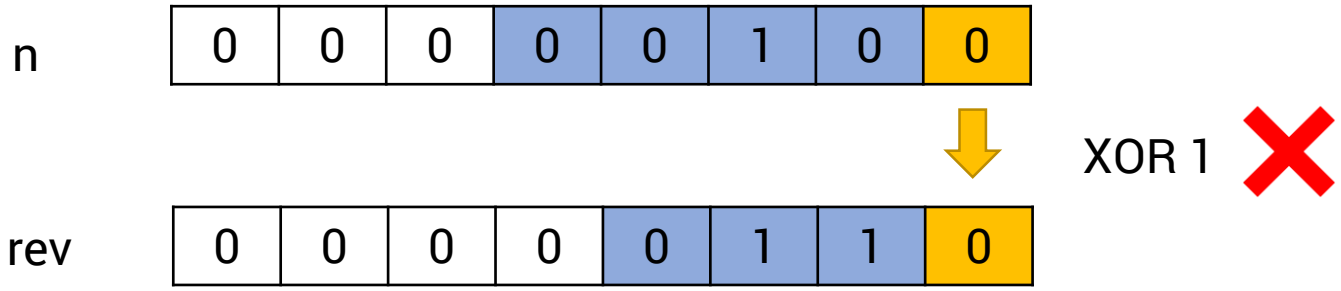
Dịch phải 1 bit số n và dịch trái 1 bit số rev:



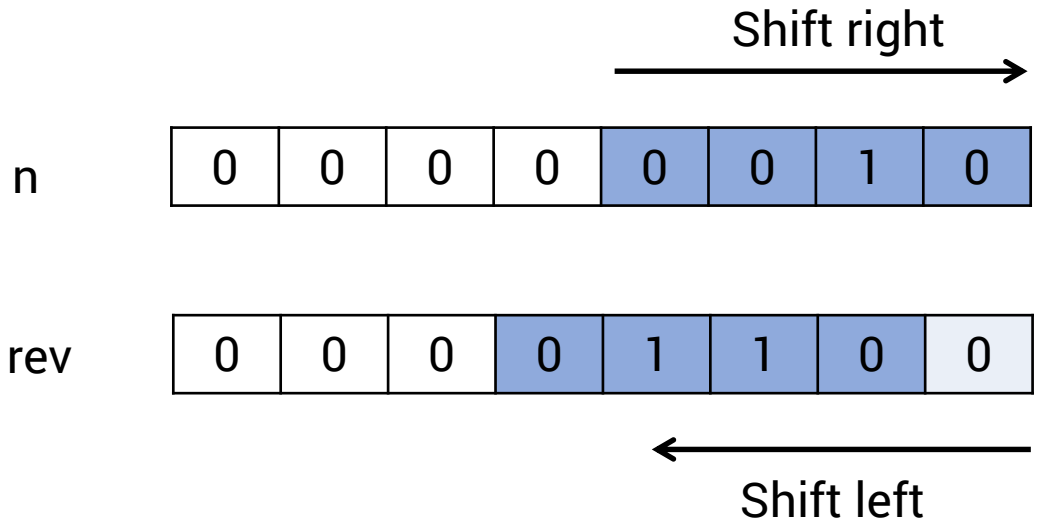
Giảm biến s = 5

Bước 2: Chép số n vào rev (4)

Xét **n != 0** ➔ Chép bit tiếp theo của số n vào rev:



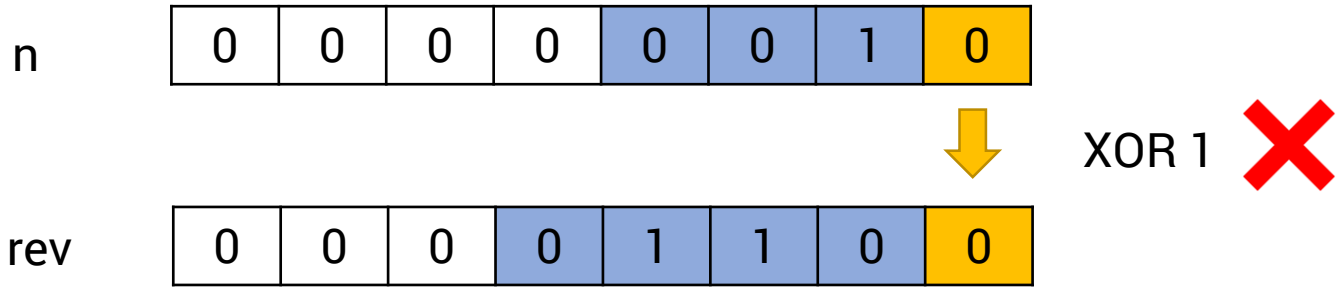
Dịch phải 1 bit số n và dịch trái 1 bit số rev:



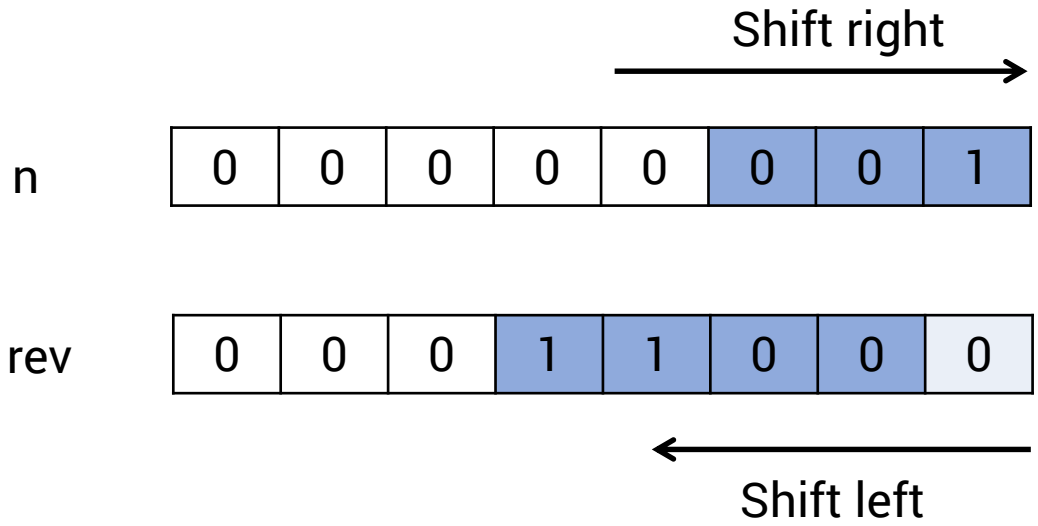
Giảm biến s = 4

Bước 2: Chép số n vào rev (5)

Xét **n != 0** ➔ Chép bit tiếp theo của số n vào rev:



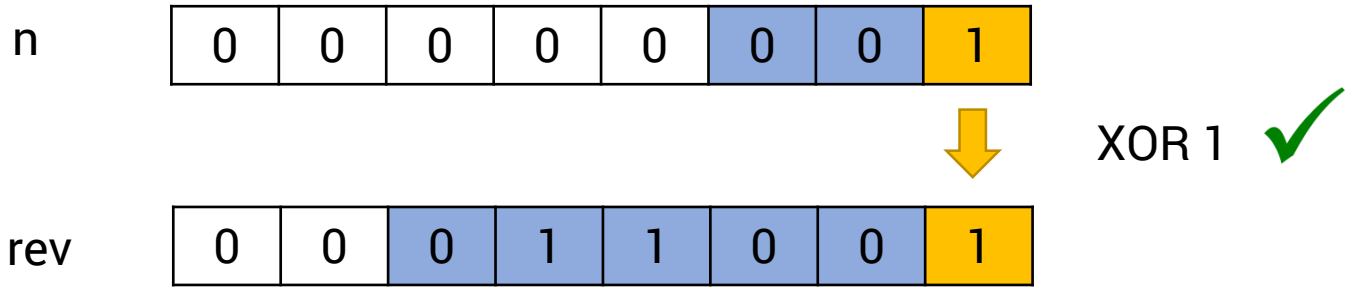
Dịch phải 1 bit số n và dịch trái 1 bit số rev:



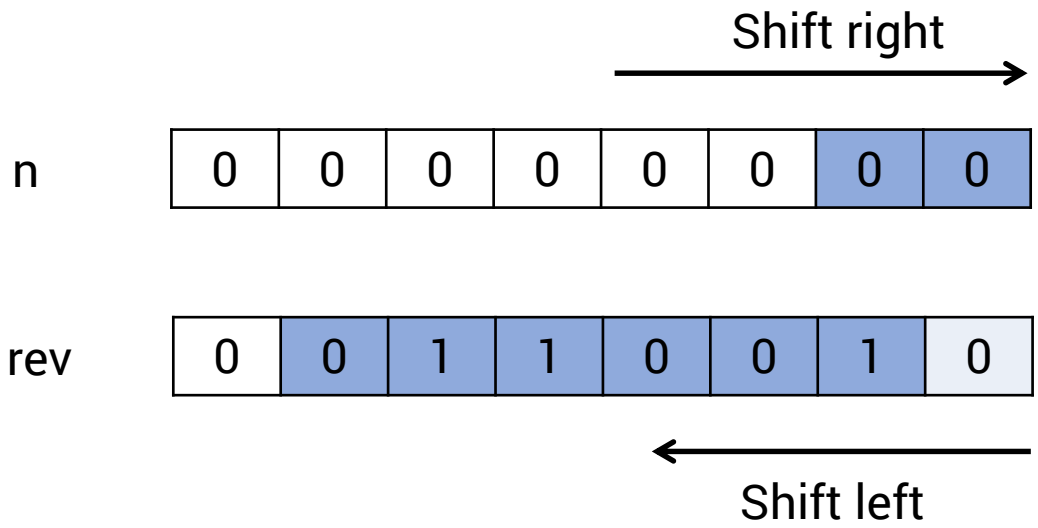
Giảm biến s = 3

Bước 2: Chép số n vào rev (6)

Xét **n != 0** ➔ Chép bit tiếp theo của số n vào rev:



Dịch phải 1 bit số n và dịch trái 1 bit số rev:



Giảm biến s = 2

Bước 2: Chép số n vào rev (7)

Xét **n == 0** ➔ Dừng thuật toán

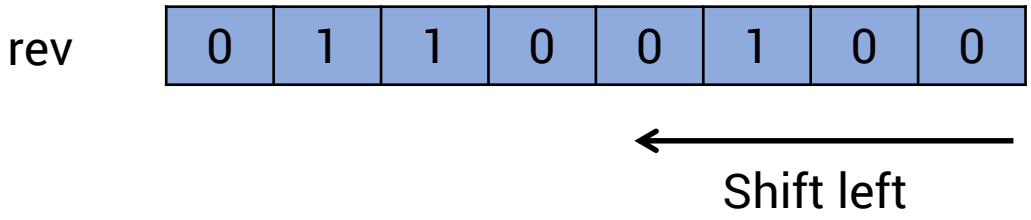
n

0	0	0	0	0	0	0	0

Bước 3 & 4: Dịch trái các bit còn lại

Dịch trái số bit còn lại nếu chưa đủ số bit mẫu.

Do s = 2 dịch trái s - 1 bit.



Kết quả rev = 100

Source Code Reverse bits

```cpp
#include <iostream>
#include <bitset>
using namespace std;
unsigned int bit_reversal(unsigned int n)
{
    int s = 8;
    unsigned int rev = 0;
    while (n != 0)
    {
        // Shift left
        rev <<= 1;
        // If current bit is '1'
        if (n & 1 == 1)
            rev ^= 1; // XOR 1
       // Shift right
       n >>= 1;
       s--;
    }
    if (s > 0)
        rev <<= s - 1;
    return rev;
}
```

Source Code Reverse bits

```cpp
int main()
{
    unsigned int n = 38;
    unsigned int rev = bit_reversal(n);
    cout << n << " (" << bitset<8>(n) << ")" << endl;
    cout << rev << " (" << bitset<8>(rev) << ")" << endl;
    return 0;
}
```

Source Code Reverse bits

```python
def bit_reversal(n):
    s = 8
    rev = 0
    while n != 0:
        # Shift left
        rev <<= 1
        # If current bit is '1'
        if n & 1 == 1:
            rev ^= 1   # XOR 1
        # Shift right
        n >>= 1
        s -= 1
    rev <<= s - 1
    return rev
```

Source Code Reverse bits

```
15. if __name__ == "__main__":
16.     n = 38
17.     rev = bit_reversal(n)
18.     print(n, ' (', bin(n), ')', sep = '')
19.     print(rev, ' (', bin(rev), ')', sep = '')
```

Source Code Reverse bits

```java
public class Main {
    static private int bit_reversal(int n) {
        int s = 8;
        int rev = 0;
        int count = 32;
        while (n != 0 && s != 0) {
            // Shift left
            rev <<= 1;
            // if current bit is '1'
            if ((n & 1) == 1)
                rev ^= 1; // XOR 1
            // Shift right
            n >>= 1;
            s--;
        }
        if (s > 0)
            rev <<= s - 1;
        return rev;
    }
```

Source Code Reverse bits

```
20. public static void main(String[] args) {
21.     int n = 38;
22.     int rev = bit_reversal(n);
23.     System.out.printf("%d (%s)\n", n, Integer.toBinaryString(n));
24.     System.out.printf("%d (%s)\n", rev, Integer.toBinaryString(rev));
25.     return;
26. }
```

Nhận xét

Ta có thể thay thế dòng lệnh

```
// if current bit is '1'
if (n & 1 == 1)
    rev ^= 1; // XOR 1
```

```
rev ^= (n & 1);
```

Do:

- Nếu bit đầu tiên của n là 1 thì (n & 1) == 1 → rev ^= (n & 1) có giá trị là 1.
- Nếu bit đầu tiên của n là 0 thì (n & 1) == 0 → rev ^= (n & 1) có giá trị là 0.

➔ rev ^= (n & 1)

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Hỏi đáp