Bits, Bytes and Numbers

This document covers

[Bit Operators](#_Bit_Operators)

[Bit Properties](#_Bit_Properties)

[Bit Manipulation](#_Bit_Manipulation)

[Bit Representations](#_Binary_representations)

[Numeric Bit Representations](#_Numeric_Bit_Representations)

## Bit Operators

|  |  |
| --- | --- |
| << | a **11**010101  a << 2 010101**00** |
| >> (Signed Integers) | a 110101**01**  a >> 2 **11**110101  a 010101**01**  a >> 2 **00**010101 |
| ~ | a 00001101  ~a 11110010 |
| & | a 00001101  b 11101011  a&b 00001001 |
| | | a 00001101  b 11101011  a|b 11101111 |
| ^ | a 00001101  b 11101011  a^b 11100110 |

## Bit Properties

|  |  |
| --- | --- |
| a ^ 0s = a | a 00001101  0s 00000000  a ^ 0s 00001101 |
| a ^ 1s = ~a | a 00001101  1s 11111111  a ^ 1s 11110010 |
| a ^ a = 0 | a 00001101  0s 00000000  a & 0s 00000000 |
| a & 0s = 0 | a 00001101  0s 00000000  a & 0s 00000000 |
| a & 1s = a | a 00001101  1s 11111111  a & 1s 00001101 |
| a & a = a | a 00001101  a 00001101  a & a 00001101 |
| a | 1s = 1s | a 00001101  1s 11111111  a | 1s 11111111 |
| a | a = a | a 00001101  a 00001101  a | 1s 00001101 |
| a ^ ~a = 1s | a 00001101  ~a 11110010  a^~a 11111111 |

## Bit Manipulation

|  |  |  |  |
| --- | --- | --- | --- |
| 1 << i | 1 << 3 | 0001000 | Create a mask with all zeros except a single 1 at bit location i |
| ~(1 << i) | ~(1 << 3) | 11110111 | Create a mask with all ones except a single 0 at bit location i |
| ~0 << n | ~0 << 3 | 11111000 | Create a mask of all 1s except for 0s in the n least significant digits |
| (1 << i)-1 | (1 << 3)-1 | 00000111 | Create a mask of all 0s except for 1s in the n least significant digits |
| (1 << j-i+1)-1)<<i | (1 << 4-2+1)-1)<<2 | 00011100 | Create a mask of all 0s except for digits i through j which contain 1s |
| ~(((1 << i - j + 1) - 1) << i)) | ~(((1 << i - j + 1) - 1) << i)) | 111000111 | Create a mask of all 1s except for digits i through j which contain 0s |
| a+(~b+1) | 5 + (~3+1) | 2 | Perform subtraction without using the - key |

## Bit Operations

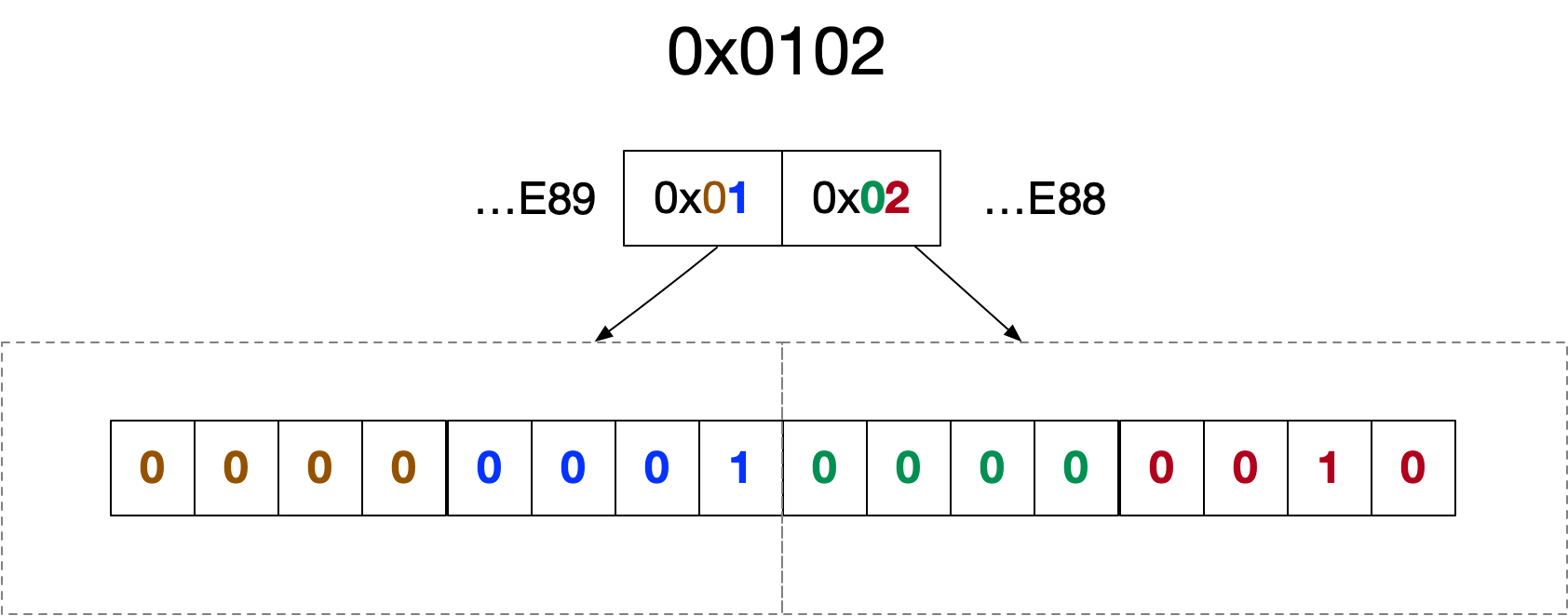
|  |  |  |  |
| --- | --- | --- | --- |
| (1 << i) | a | (1 << 2) | 0b00101000 | 00101100 | Set bit i to 1 |
| ~(1 << i) & a | ~(1 << 2) & 0b00101100 | 00101000 | Set bit i to 0 |
| (a >> i) & 1 | (0b00101100 >> 3) & 1 | 1 | Get the value of bit i |
| (~0 << i) & a | (~0 << 3) & 0b10101111 | 10101000 | Clear i least significant bits |
| ((1 << i)-1) | a | ((1 << 3)-1) | 0b10100000 |  | Set i least signifcant bits |
| a & (a-1) | a 00111100 a-1 00111011 a&(a-1) 00111000 |  | Clear the rightmost (least significant) 1 digit |

## Binary representations

Most numeric types consist of multiple bytes. The order in which the bytes are arranged in memory is known as endianness. On a little endian system, a numeric object’s least to most signifant bytes are arranged in order from lower memory addresses to higher memory addresses. Consider a .NET unsigned short which occupies 2 bytes or 16 bits

ushort a = 0x0102;

Figure Endianess



### Bit Operators

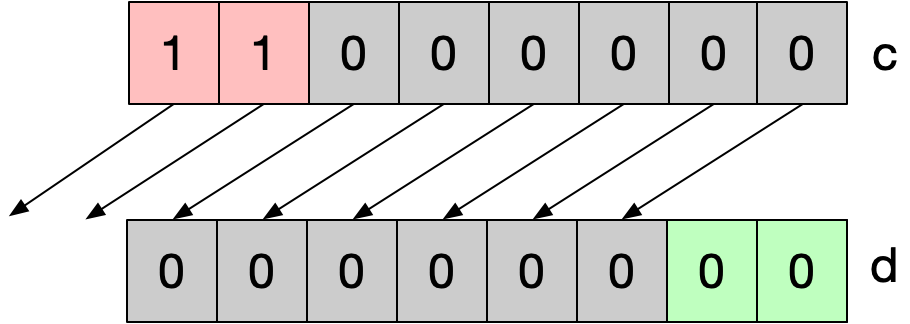
<< Left Shift

A left shift moves everything n place to the left. The left most n bits are dropped and the rightmost n bits are filled with zeros.

byte c = byte.MaxValue;

byte d = (byte)(a >> 2);

Figure Left Shift

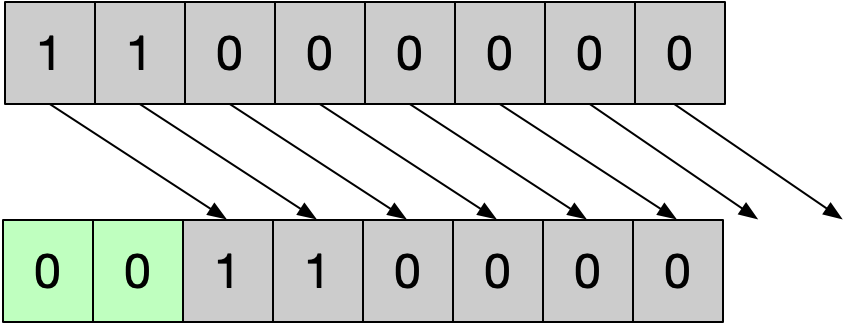


>> Right Shift

The right shift operator shifts all bits n places to the right. The rightmost n digits are dropeed and the leftmost n digits are filled as follows. If the operand is unsigned the left n bits are filled with zeros

byte c = 128+64;

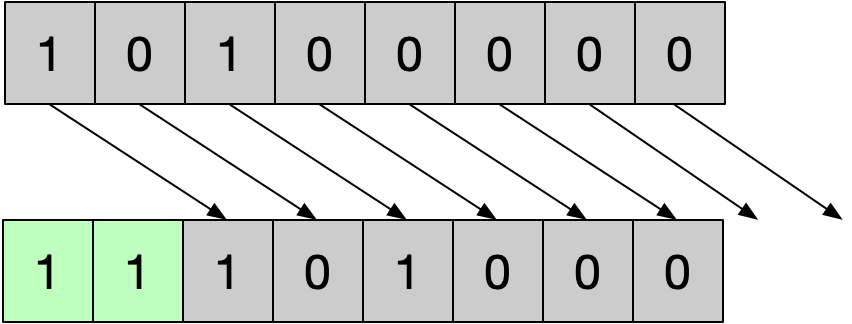
byte d = (byte)(c >> 2);



If the operand is signed the sign of the bits filled on the left matches the sign bit in the most significant position.

sbyte a = -96;

sbyte b = (sbyte)(a >> 2);



This extra complexity ensures that shifting right 1 place is equivalent to divinding by two when the operand is negative.

~ bitwise complement

Inverts all the bits

a 00001101

~a 11110010

& bitwise and

Copies a 1 into the result if the corresponding bits in each operand are 1

a 00001101

b 11101011

a&b 00001001

| bitwise or

a 00001101

b 11101011

a|b 11101111

^ exclusive or

a 00001101

b 11101011

a^b 11100110

## Numeric Bit Representations

We now move on to show how number systems of the following form are represented in .NET

#### Unsigned Integers

If we only need to represent positive whole numbers, that is to say unsigned integers, we can use a n-bit binary representation. We don’t need any bits to represent fractions.

Such a representation can distinguish between different values which we can use to represent positive integers in the range To highlight the approach consider the specific case of

00 0000

01 0001

02 0010

03 0011

04 0100

05 0101

06 0110

07 0111

08 1000

09 1001

10 1010

11 1011

12 1100

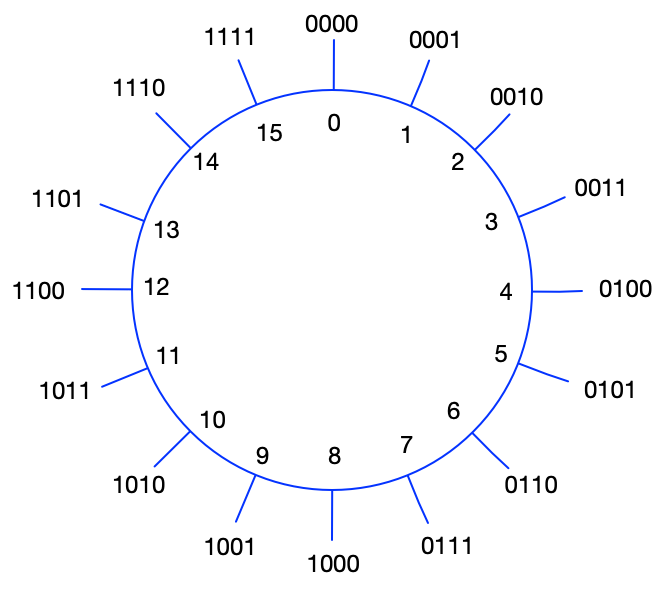
13 1101

14 1110

15 1111

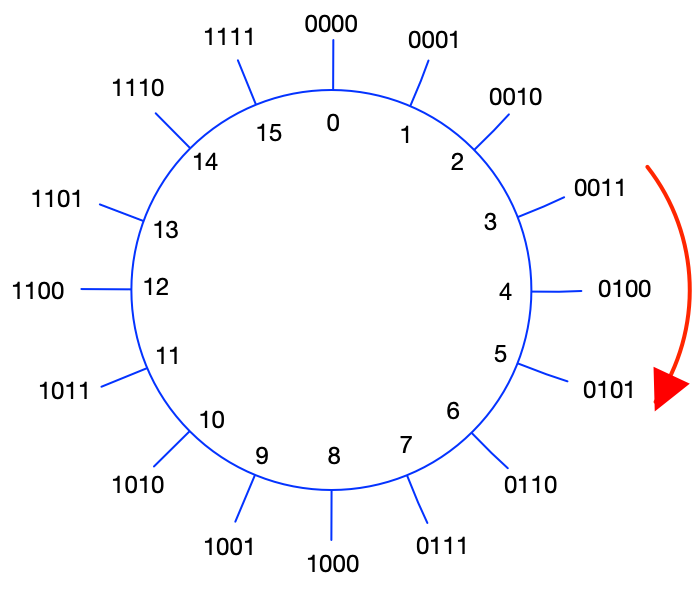
It is useful to visualize such a system as a circle

Figure Unsigned Integers



##### Unsigned Addition

Addition is achieved by starting at a and moving b places clockwise around the wheel. Consider the specific case of We visualize this as follows



This is very simple binary addition

0011

+ 0010

0101

The following C# code shows how we might achieve such addition

Figure Adding Unsigned Integers



Notice in our add method we do not deal with the overflow from the most significant bit. When we add one to the largest representable binary digit which consists of all ones the result is the smallest binary digit consisting of all zeros. In a four bit unsigned integer we would have as follows. Note the bold red overflow is discarded.

1111

+ 0001

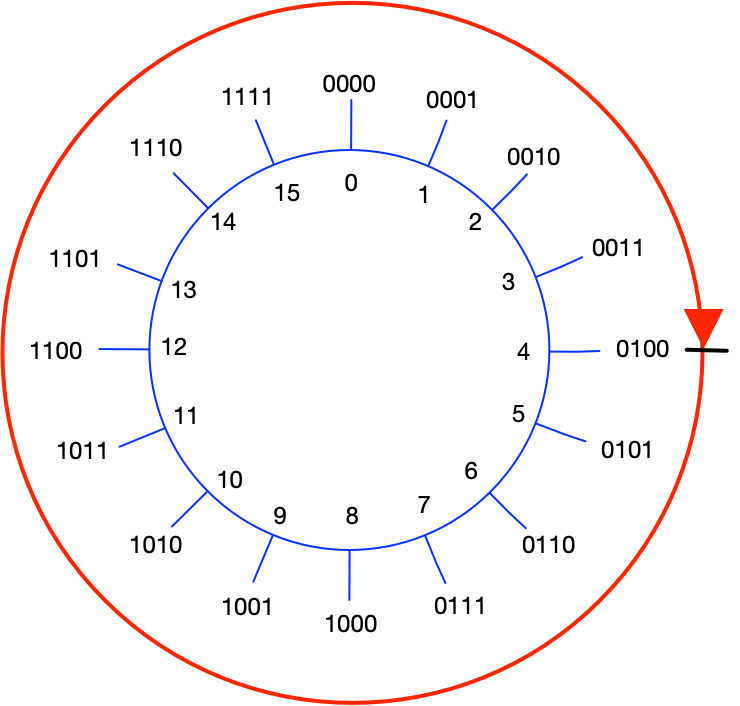
**1**0000

By implementing add in this way we have created a modulo number system. If there are n bits in our unsigned integer then addition is . For any unsigned integers a and m we have

=

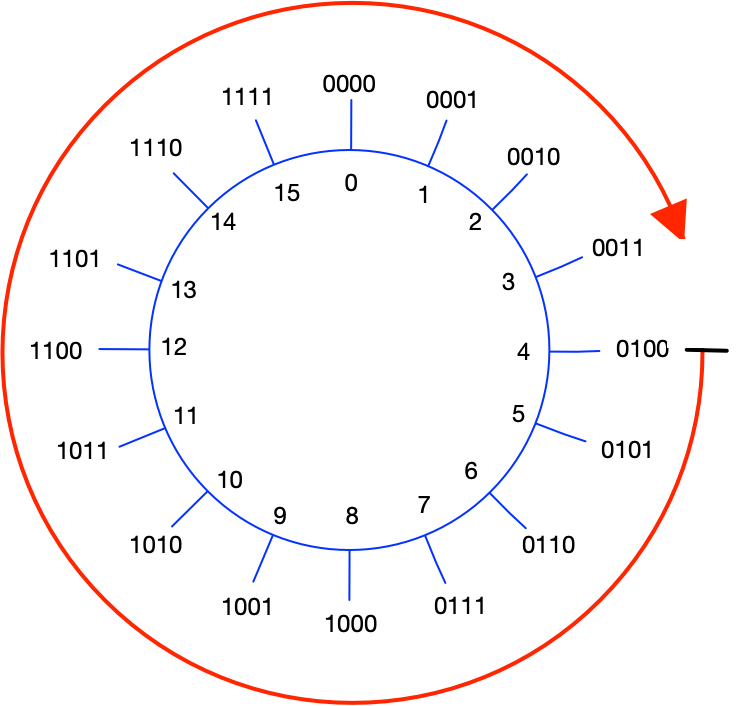
=for

In our case adding to any value gets back to the same value. We show 4



##### Unsigned subtraction

In our 4 bit integer notice what happens if we add to 4. We only rotate to 3. So adding is the same as adding -1.



Similarly, adding is the same as subtracting 2 and adding is the same as subtracting b. We noted in the previous section that = and so it is self evident that

This is a very useful result if we combine it with the following observation. Adding any binary number to its complement gives a number consisting solely of 1s.

And in our representation we have that hence it follows that

If we substitute this into the expression

We get

This means we can use our method for addition of unsigned integers to perform subtraction of unsigned integers. The following shows the simple C# code

public uint Subtract(uint a, uint b) => Add(a, Add(~b, 1));

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| b  (Decimal) | B  (Binary) | ~b  (Binary) | Adding  (Clockwise) | Subtraction  (Anticlockwise) |
| 0 | 0000 | 1111 | 15 |  |
| 1 | 0001 | 1110 | 14 |  |
| 2 | 0010 | 1101 | 13 |  |
| 3 | 0011 | 1100 | 12 |  |

Proof that =

From properties of modulo numbers we know that

= and hence

= rearranging

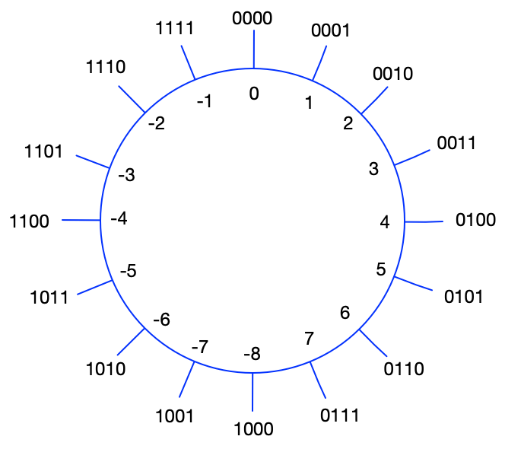
= adding and subtracting 1

=

#### 2s Complement Signed Integers

A n bit 2s complement representation supports the values from …

Figure 2s Complement Signed Integers



+00 0000

+01 0001

+02 0010

+03 0011

+04 0100

+05 0101

+06 0110

+07 0111

-08 1000

-07 1001

-06 1010

-05 1011

-04 1100

-03 1101

-02 1110

-01 1111

The table shows that to negate a number we complement it and add one. We say we are taking the twos complement. When adding a pair of twos complement numbers where one of them is negative we simply move around the wheel the number of places in the positive direction of the twos complement binary representation.

## Manipulating Binary

### Tricks

Adding the same number

Performing integer addition where both operands are the same equal to multiplying by two which is equal to shifting left one place.

0000**1101**

+ 0000**1101**

000**1101**0

Multiplication

In binary multiplication is simply shiting the multiplicand left by a number of digits equal to the multiplier.

0000**1101**

\* 00000011

0**1101**000

## Problems and Interview Questions