Bits, Bytes and Numbers

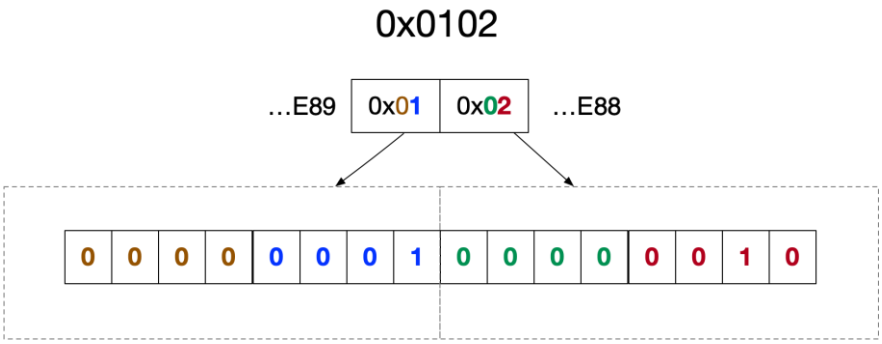
THIS DOCUMENT COVERS

- Introduction

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;
```



Numeric Bit Representations

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

$$\pm(d_\infty\beta^\infty+\ldots d_1\beta^1+d_0\beta^0+b_{-1}\beta^{-1}+b_{-1}\beta^{-2}+\ldots b_{-\alpha}\beta^{-\alpha})=\pm\left(\sum_{k=-\infty}^{\infty}d_k\beta^k\right)$$

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.

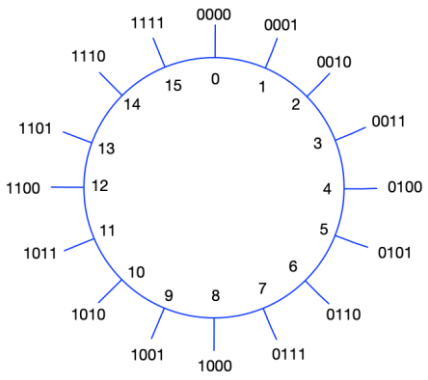
$$(d_{n-1}\ldots d_1d_0)_2=(d_{n-1}2^{n-1}+\ldots d_12^1+d_02^0)=\left(\sum_{k=0}^{n-1}d_kn^k\right)$$

Such a representation can distinguish between 2^n different values which we can use to represent positive integers in the range $[0, 2^n - 1]$ To highlight the approach consider the specific case of $n = 4$

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

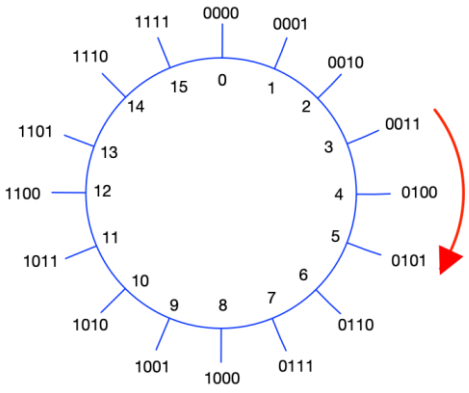
It is useful to visualize such a system as a circle

Figure 1 Unsigned Integers



Unsigned Addition

Addition $(a + b)$ is achieved by starting at a and moving b places clockwise around the wheel. Consider the specific case of $(3 + 2)_{10} = (0011 + 0010)_2$ 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 2 Adding Unsigned Integers

```
public uint Add(uint a, uint b)
{
	uint carry = 0;
	uint result = 0;

	for (int bitIdx = 0; bitIdx < SizeInBits; bitIdx++)
	{
		// We deal in one binary digit at multiplicand time. By right
		// shifting multiplicand and bitIdx times we set the bit we want
		// into the least significant bit.
		uint aShifted = (a >> bitIdx);
		uint bShifted = (b >> bitIdx);

		// Now we make use of the fact that the number 1 has
		// in our unsigned representation consists of SizeInBits
		// zeros followed by multiplicand solitary one in the least significant
		// position. We can hence take our shifted valued and logically
		// and them with 1 to ensure we only have the digit values in the least
		// significant locations remaining.
		uint aDigit = aShifted & 1;
		uint bDigit = bShifted & 1;

		// We have three binary digits that feed into the current digit
		// {the multiplicand digit, the multiplier digit and the carry}
		//If one or all three
		// of these are one then the digit will be one, otherwise it will be
		// zero.
		uint sumBit = (aDigit ^ bDigit) ^ carry;

		// We now shift the bit into its correct location and add it into the
		// result
		result = result | (sumBit << bitIdx);

		// Finally calculate the carry for the next digit
		carry = (aDigit & bDigit) | (aDigit & carry) | (bDigit & carry);
	}

	return result;
}
```

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.

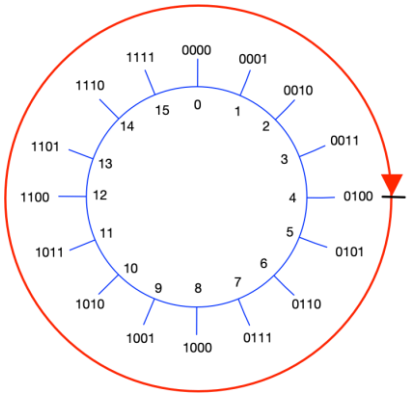
```
    1111
+   0001
   10000
```

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 mod_{2^n}. For any unsigned integers a and m we have

$(a + 2^n)_{mod_2 n} = a_{mod_2 n}$

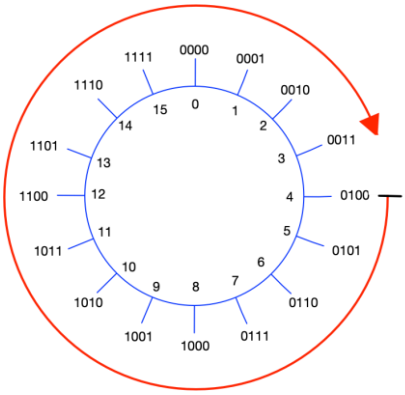
$(a + m.2^n)_{mod_2 n} = a_{mod_2 n}$ for

In our case adding $2^4 = 16$ to any value gets back to the same value. We show $4 + 2^4 = 4$



Unsigned subtraction

In our 4 bit integer notice what happens if we add $2^4 - 1 = 15$ to 4. We only rotate to 3. So adding $2^4 - 1$ is the same as adding -1.



Similarly, adding $2^4 - 2$ is the same as subtracting 2 and adding $2^4 - b$ is the same as subtracting b. We noted in the previous section that $(a + 2^n)_{mod_2 n} = a_{mod_2 n}$ and so it is self evident that

$$(a + [2^n - b])_{mod_2 n} = (a - b)_{mod_2 n}$$

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.

$$b + \sim b = 11 \ldots 1$$

And in our representation we have that $11 \ldots 1 = 2^n - 1$ hence it follows that

$$b + \sim b = 2^n - 1$$

If we substitute this into the expression

$$(a + [2^n - b])_{mod_{2^n}} = (a - b)_{mod_{2^n}}$$

We get

$$(a - b)_{mod_{2^n}} (a + \sim b + 1)_{mod_{2^n}}$$

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	$15 - 2^4 = -1$
1	0001	1110	14	$14 - 2^4 = -2$
2	0010	1101	13	$13 - 2^4 = -3$
3	0011	1100	12	$12 - 2^4 = -4$

Proof that $(a + \sim b + 1)_{mod_{2^n}} = (a - b)_{mod_{2^n}}$ ∴

From properties of modulo numbers we know that

$(a + 2^n)_{mod_{2^n}} = a_{mod_{2^n}}$ and hence

$(a - b + 2^n)_{mod_{2^n}} = (a - b)_{mod_{2^n}}$ rearranging

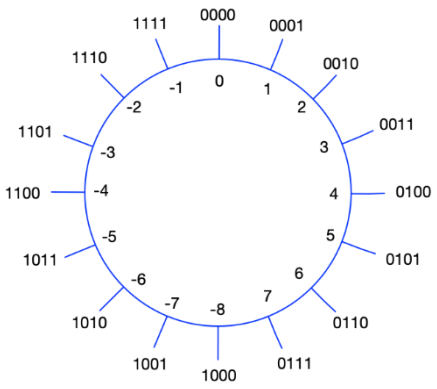
$(a + [2^n - b])_{mod_{2^n}} = (a - b)_{mod_{2^n}}$ adding and subtracting 1

$(a + [2^n - 1 - b] + 1)_{mod_{2^n}} = (a - b)_{mod_{2^n}}$

2s Complement Signed Integers

A n bit 2s complement representation supports the values from $-2^{n-1}\ldots(2^{n-1}-1)$

Figure 3 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.$-a = \sim a + 1$ 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.

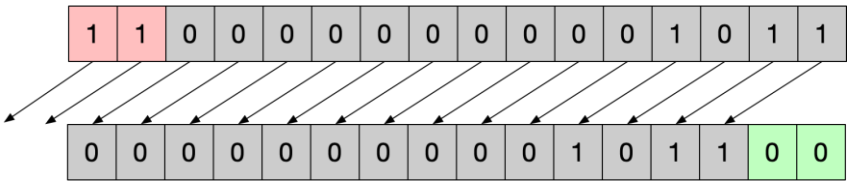
Bit Operators

<< Left Shift

Given our representation above of the bits with least significant to the right a left shift moves everything one place to the left. It is equivanent to multiplying by two in binary

```
ushort a = 0x0005;
ushort b = (ushort)(a << 2);
```

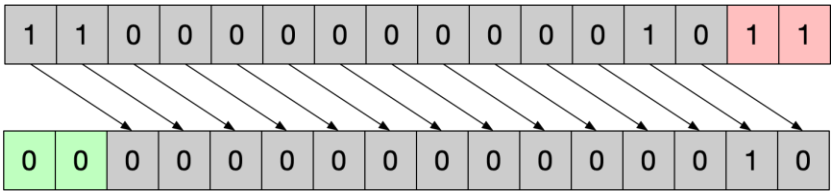
Figure 4 0x05 << 2 = 0x = 0xA



>> Right Shift

```
ushort a = 0x0005;
ushort b = (ushort)(a >> 2);
```

Right shift is equivalent to dividing by 2



~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
```

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
a            00001101
a ^ a        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 | 0s = a

```
a            00001101
0s           00000000
a | 0s       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 Tricks

GetBit

The following code shows how to get a bit at a particular index in an integers binary representation. We illustrate the idea with the specific example of getting the bit at index 2 on the number 27 which is zero.

```
27                  00011011
1 << 2              00000100
27 & (1 << 2)       00000000
```

```
public bool GetBit(int number, int bitIdx)
{
    return ((1 << bitIdx) & number) != 0;
}
```

SetBit

The following code shows how to set a bit at a particular index in an integers binary representation. We illustrate the idea with the specific example of setting the bit at index 2 on the number 27 to give the number 31

```
27                  00011011
1 << 2              00000100
27 | (1 << 2)       00011111
```

```
public int SetBit(int number, int bitIdx)
{
    return (1 << bitIdx) | number;

}
```

ClearBit

The following code shows how to clear a bit at a particular index in an integers binary representation. We illustrate the idea with the specific example of clearing the bit at index 3 on the number 27 to give the number 19

```
27                  00011011
1 << 2              00001000
~(1 << 2)           11110101
27 & ~(1 << 2)      00010011
```

```
public int ClearBit(int number, int bitIdx)
{
    int mask = ~(1 << bitIdx);
    return mask & number;
}
```

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.

```
      00001101
+     00001101
      ---------
      000110100
```

Multiplication

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

```
      00001101
*     00000011
      ---------
      01101000
```

Clear rightmost bits a & (~0 << n)

```
a                   00011011
~0 << 2             11111100
a & (~0 << 2)       00011000
```