DigiPen INSTITUTE OF TECHNOLOGY

CS 176
Advanced Scripting

Binary & Bitwise Operators

Elie Abi Chahine

Binary Numbering Systems

Why Binary?

As human beings, we intuitively count to ten because we have ten fingers and ten toes.

Because of this, our numbering system is based on the number 10; it's a base 10 numbering system.

The computer system is based on electrical wires. The electrical wire can have two states: current is present or current is NOT present. Consequently, binary numbers system is the natural numbering system to use with a computer.

The binary numbering system consists of 2 symbols: 0 and 1. Therefore, the base is 2.

In binary we count like this: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010 …

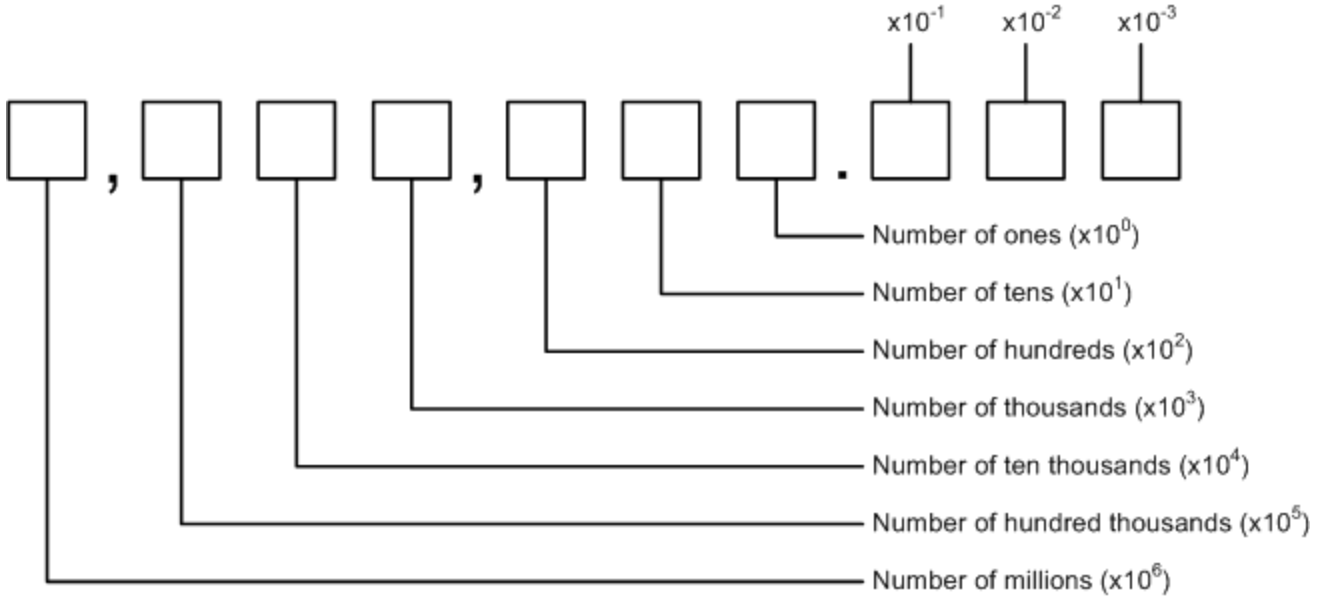
Binary

Here is a decimal and binary table:
MSB: Most Significant Bit
LSB: Least Significant Bit

Decimal	Binary			
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
	MSB			LSB

Positional Numbering Systems

Positional: A weight is given to each digit in the number depending on the position of the digit in the number. The digit '1' in the number 130 and the digit '1' in the number 12 have different meaning. In the first it indicates how many hundreds are in 130; whereas it indicates how many tens are in 12.



Positional Numbering Systems

Decimal:

$$1384.29 = 1 * 10^3 + 3 * 10^2 + 8 * 10^1 + 4 * 10^0 + 2 * 10^{-1} + 9 * 10^{-2}$$

Binary:

$$100.11_2 = 1*2^2 + 0*2^1 + 0*2^0 + 1*2^{-1} + 1*2^{-2}$$

Conversion: Binary to Decimal

Example1: 111_2 to Decimal

$$111_2 = (2^2*1) + (2^1*1) + (2^0*1)$$
$$= 4 + 2 + 1$$
$$= 7_{10}$$

Example2: 1001_2 to Decimal

$$1001_2 = (2^3*1) + (2^2*0) + (2^1*0) + (2^0*1)$$
$$= 8 + 0 + 0 + 1$$
$$= 9_{10}$$

Example3: 1010_2 to Decimal

$$1010_2 = (2^3*1) + (2^2*0) + (2^1*1) + (2^0*0)$$
$$= 8 + 0 + 2 + 0$$
$$= 10_{10}$$

Positional Numbering Systems

Another way to do a binary to decimal conversion is by using this template:

1	0	1	0	1	1	0	1
x 128	x 64	x 32	x 16	x 8	x 4	x 2	x 1

128	+	0	+	32	+	0	+	8	+	4	+	0	+	1	=	173

Conversion: Decimal to Binary

Example1: 7_{10} to Binary

7/2 = 3 Remainder = 1

3/2 = 1 Remainder = 1

1/2 = 0 Remainder = 1 (last remainder)

Decimal $7_{10} = 111_2$ Binary

Example2: 9_{10} to Binary

9/2 = 4 Remainder = 1

4/2 = 2 Remainder = 0

2/2 = 1 Remainder = 0

1/2 = 0 Remainder = 1 (last remainder)

Decimal $9_{10} = 1001_2$ Binary

Example3: 10_{10} to Binary

10/2 = 5 Remainder = 0

5/2 = 2 Remainder = 1

2/2 = 1 Remainder = 0

1/2 = 0 Remainder = 1 (last remainder)

Decimal $10_{10} = 1010_2$ binary

Positional Numbering Systems

Another way to do a decimal to binary conversion is by using this template:

173	45	45	13	13	5	1	1
÷ 128	÷ 64	÷ 32	÷ 16	÷ 8	÷ 4	÷ 2	÷ 1
1	0	1	0	1	1	0	1

Addition in Binary

Arithmetic calculation in all positional numbering systems follows the same principles, so the addition table looks like this:

Operands	Sum
0 + 0	0
0 + 1	1
1 + 0	1
1 + 1	**0 (carry 1)**

Addition in Binary

Example 1: $01_2 + 01_2 = 10_2$

Binary Addition		
Carry	1	
	0	1
+	0	1
Sum	1	0

Example 2: $01_2 + 01_2 + 01_2 = 11_2$

Binary Addition		
Carry	1	
	0	1
	0	1
+	0	1
Sum	1	1

Addition in Binary

Example 3: **$1011101+1111001=11010110_2$**

Binary Addition								
Carry	1	1	1	1			1	
		1	0	1	1	1	0	1
	+	1	1	1	1	0	0	1
Sum	1	1	0	1	0	1	1	0

Signed binary

Before going through the bitwise operators, let us learn a little about signed binary numbers.

For the machine, a negative number is represented by the 2's complement of it's corresponding positive number:

-5 = 2's complement of (5)

2's Complement

2's complement of M = 1's complement of M + 1

In order to compute the 1's complement of M, you convert M to binary and then flip all the bits (1 -> 0 and 0 -> 1)

Example:

3 in base 10 = **00000011** in base 2

1's complement of **3** = 1's complement of **00000011** = **11111100**

-3 = 2's complement of **3** = 1's complement of **00000011 + 1**
= **11111100 + 1 = 11111101**

2's Complement

An easier way to obtain the 2's complement of a binary number is to parse that number starting with the LSB until you find the rightmost digit that is equal to 1. Leave that digit and all other digits to the right of it unchanged, and then complement all digits to the left of that one digit.

Bits	8(MSB)	7	6	5	4	3	2	1(LSB)
+72	0	1	0	0	1	0	0	0
	Flip	Flip	Flip	Flip	No Flip	No Flip	No Flip	No Flip
-72	1	0	1	1	1	0	0	0

The 2's complement of a negative number represented in its 2's complement form is equal to its corresponding positive number.

Bits	8(MSB)	7	6	5	4	3	2	1(LSB)
-72	1	0	1	1	1	0	0	0
	Flip	Flip	Flip	Flip	No Flip	No Flip	No Flip	No Flip
+72	0	1	0	0	1	0	0	0

Signed binary addition

5 – 3 = 5 + (-3) = 5 + (2's complement of 3)

5 base 10 = 00000101 in base 2
3 base 10 = 00000011 in base 2
-3 = 2's complement of 3 = 11111101

5 – 3 = 00000101 + 11111101 = 00000010 in base 2 = 2 in base 10

Bitwise Operators

Some Operators

Operator	Name
<<	Shift Left
>>	Shift Right
>>>	Shift Right with Zero Fill
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR

Shift Left (<<)

The shift left operator **(<<)** shifts all the bits of a number **n** times to the left.

```
var a:int = 10;          /* 00001010  */
a = a << 1;              /* 00010100  */
trace(a);                /* 20        */

a = 10;                   /* 00001010  */
a = a << 3;               /* 01010000  */
trace(a);                /* 80        */
```

Explanation

```
     Binary:   Decimal:

     00001010       10

 <<         1

_____________________
     00010100       20
```

As you can see, all the bits moved one space to the left resulting in a different integer value. Every time we shift the bits, a zero value is placed at the least significant bit (the one on the far right).

```
     Binary:   Decimal:

     00001010       10

 <<         3

_____________________
     01010000       80
```

Shifting a number by 3 moves all its bits by 3 spaces.

Shifting to left by **n** is equivalent to multiplying the number by $\mathbf{2^n}$

10 << 1 = 20 and 10 * 2^1 = 20

10 << 3 = 80 and 10 * 2^3 = 80

Shift Right (>>)

The shift right operator **(>>)** shifts all the bits of a number **n** times to the right.

```
var a:int = 10;          /* 00001010  */
a = a >> 1;              /* 00000101  */
trace(a);                /* 5         */
```

Explanation

```
    Binary:   Decimal:

    00001010       10

>>        1

_____________________

    00000101        5
```

As you can see, all the bits moved one space to the right resulting in a different integer value. The most significant bit got replaced with a **0**.

This illustrates that, like a left shift by **n** is equivalent to multiplying a number by 2^n, a right shift is equivalent to dividing a number by 2^n.

Shift Right (>>)

Let's test the Shift Right operator with negative values.

```
var a:int = -10;        /* 11110110 */
a = a >> 1;             /* 11111011 */
trace(a);               /* -5       */
```

Explanation

```
    Binary:   Decimal:

    11110110      -10

 >>        1

_____________________

    11111011       -5
```

All the bits moved one space to the right resulting in a different integer value. Unlike the example above, the most significant bit got replaced with a **1**. This is something special with the **Shift Right** operator, when shifting it replaces the most significant bit by **0** for **positive numbers** and by **1** for **negative numbers** in order to preserve the sign.

Shift Right With Zero Fill (>>>)

The shift right with zero fill operator (>>) shifts all the bits of a number n times to the right, but unlike the shift right operator (>>) it does not preserve the sign.

```
var a:int = -10;
a = a >> 1;
trace(a);                /* -5 */

var a:int = -10;
a = a >>> 1;
trace(a);                /* 2147483643 */
```

Explanation

Binary:	Decimal:
11111111111111111111111111110110	-10
>> 1	
11111111111111111111111111111011	-5

With the shift right operator (>>) all the bits moved one space to the right with a most significant bit value equal to 1 in order to preserve the sign.

Binary:	Decimal:
11111111111111111111111111110110	-10
>>> 1	
01111111111111111111111111111011	2147483643

With the shift right with fill zero operator (>>>) all the bits moved one space to the right with a most significant bit value equal to 0 which will not preserve the sign hence the new value of 2147483643.

Note: The result of this operation will always be positive since the most significant bit will always be a zero.

AND (&)

The bitwise AND operator (&) is a binary operator that works on the bit level. Given two bits, the AND operator will return a 1 if, and only if, both operands are equal to 1.
In other words, the AND operator follows the following table:

&	0	1
0	0	0
1	0	1

AND (&)

```
trace("0 & 0 = " +  (0 & 0));        /* 0 */
trace("0 & 1 = " +  (0 & 1));        /* 0 */
trace("1 & 0 = " +  (1 & 0));        /* 0 */
trace("1 & 1 = " +  (1 & 1));        /* 1 */

var one:int = 8;                     /* 00001000 */
var two:int = 168;                   /* 10101000 */
trace(one & two);                    /* 00001000 */
```

Explanation

```
     Binary:   Decimal:

     00001000         8

&    10101000       168

-------------------------

     00001000         8
```

Each bit in the **one** variable will be AND-ed with its respective bit in the **two** variable. The returned value will be a new integer.

OR (|)

The bitwise OR operator (|) is a binary operator that works on the bit level. Given two bits, the OR operator will return a 1 if either of the operands is equal to 1.
In other words, the OR operator follows the following table:

|	0	1
0	0	1
1	1	1

OR (|)

```
trace("0 | 0 = " +  (0 | 0));      /* 0 */
trace("0 | 1 = " +  (0 | 1));      /* 1 */
trace("1 | 0 = " +  (1 | 0));      /* 1 */
trace("1 | 1 = " +  (1 | 1));      /* 1 */

var one:int = 72;                  /* 01001000 */
var two:int = 168;                 /* 10101000 */
trace(one | two);                  /* 11101000 */
```

Explanation

```
    Binary:   Decimal:

    01001000        72

|   10101000       168

-------------------------

    11101000       232
```

Each bit in the **one** variable will be OR-ed with its respective bit in the **two** variable. The returned value will be a new integer.

XOR (^)

The bitwise XOR operator (^) is a binary operator that works on the bit level. Given two bits, the XOR operator will return a 1 if, and only if, one of the operands is equal to 1.
In other words, the XOR operator follows the following table:

^	0	1
0	0	1
1	1	0

XOR (^)

```
trace("0 ^ 0 = " +  (0 ^ 0));     /* 0 */
trace("0 ^ 1 = " +  (0 ^ 1));     /* 1 */
trace("1 ^ 0 = " +  (1 ^ 0));     /* 1 */
trace("1 ^ 1 = " +  (1 ^ 1));     /* 0 */

var one:int = 72;                 /* 01001000 */
var two:int = 168;                /* 10101000 */
trace(one ^ two);                 /* 11100000 */
```

Explanation

```
    Binary:   Decimal:

    01001000        72

^   10101000       168

-------------------------

    11100000       224
```

Each bit in the **one** variable will be XOR-ed with its respective bit in the **two** variable. The returned value will be a new integer.

Using the bits of an integer as flags

Now that we know how all the bitwise operators work, let's see how we can benefit from them in code.

Assume we have an integer variable and the size of an integer is 32bits.

By using bitwise operators we can treat the bits inside that integer as 32 flags which can replace 32 booleans. By doing that, we are saving a lot of memory since the size of one integer is definitely less than the size of 32 booleans.

Using the bits of an integer as flags

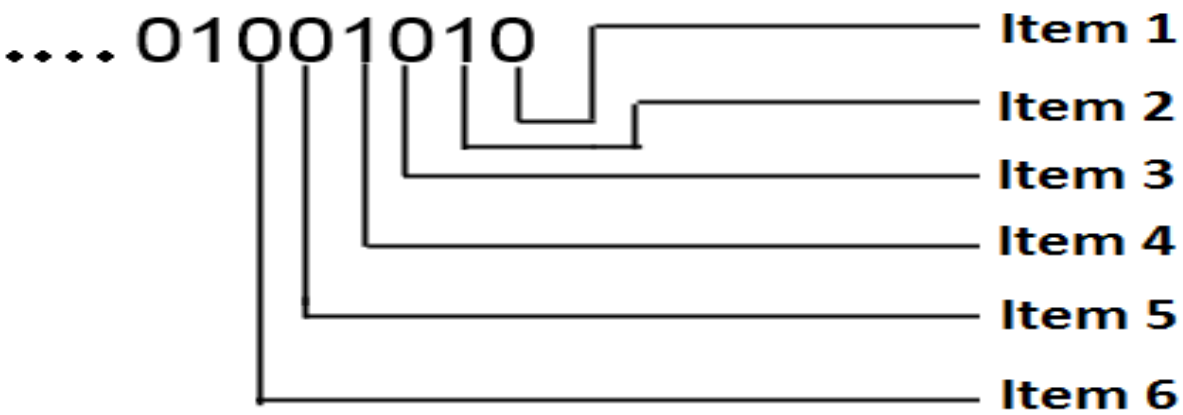
Say we are designing a video game where each player can collect a total of 32 items.

One way of designing our code is to have an array of 32 booleans for each character.

Each boolean will represent a specific item, and when the player collects that specific item we change the boolean to true.

Or, we can instead have one integer and use its bits as 32 flags, one for each item.

Using the bits of an integer as flags



From the picture above we know that the player collected items 2, 4 and 7.

Using the bits of an integer as flags

Let's say the game is running and for some reason we need to check if the player collected item 4, the only thing we need to do is AND (&) our items integer variable with the value 8 (which is called the item's mask) and if the result is anything other than 0 then the player has the item.

```
   01001010  Items
             Integer

&  00001000  Item4
             Mask
-------------------
   00001000
```

Let's do the same test but with item 1:

```
   01001010  Items
             Integer

&  00000001  Item1
             Mask
-------------------
   00000000
```

The result is 0 which indicates that the player hasn't collected item 1 yet.

Note: An item's mask is the integer value that has all the bits set to 0 except for the respective item's flag.

Using the bits of an integer as flags

How about setting a bit flag on when the player collects an item. In order to do so, we OR (|) the items integer with the item's mask.

Assume the player just collected item 1:

```
   01001010  Items
             Integer

|  00000001  Item1
             Mask
-------------------
   01001011
```

As you can see, all the items integer's flags remained the same except for item 1's flag that got set.

Very Important Note	*When we want to set a bit flag to true, we need to change the items integer variable.* *ItemsInteger = ItemsInteger | ItemMask* *It is not the case when we check if an item is collected or not:* *if((ItemsInteger & ItemMask) != 0)* *{* *...* *}*

Using the bits of an integer as flags

Last but not least, we can use the XOR (^) operator to toggle a bit ON and OFF. Assume that the 7th bit controls if a light is turned on or off:

```
   01001010  Items
             Integer

^  01000000  Bit7
             Mask
-------------------
   00001010
```

As you can see, XOR-ing the items integer variable with the "Bit7Mask" variable changed the 7th bit from 1 to 0. Let's XOR it again with the same mask:

```
   00001010  Items
             Integer

^  01000000  Bit7
             Mask
-------------------
   01001010
```

The 7th bit is changed from 0 to 1. Toggling is done with the following equation:

$$ItemsInteger = ItemsInteger \ \hat{} \ bitMask$$

The End ☺