

# Opérations bit à bit

## Les bases de numérations

La base que l'on utilise tout le temps pour compter, c'est la base 10 que l'on appelle décimale (qui comprend les chiffres 0,1, 2,3,4,5,6,7,8,9). Mais il existe d'autres tel que l'hexadécimal (base 16), le binaire (base 2), l'octal (base 8) et bien d'autres.

## Base quelconque vers décimal

Pour convertir un nombre d'une base quelconque vers le décimal (base 10) on peut appliquer la formule (que l'on répète pour chaque chiffre)

$$\sum a_n B^n$$

- $n$  est le numéro de la position du chiffre dans le nombre (unité  $\rightarrow 0$ , dizaines  $\rightarrow 1$ , centaines  $\rightarrow 2$ , dixièmes  $\rightarrow -1$ , centièmes  $\rightarrow -2$ )
- $a_n$  correspond à la valeur du chiffre en position  $n$  du chiffre dans sa base d'origine
- $B$  est la base (exemple hexadécimal  $\rightarrow 16$ )

Par exemple, pour convertir le nombre `CAFE` (hexadécimal) en décimal, on peut faire

$$\text{CAFE}_{16} = 12 * 16^3 + 10 * 16^2 + 15 * 16^1 + 14 * 16^0 = 51866_{10}$$

## Décimal vers binaire (ou autre base)

Pour convertir le décimal en binaire, on peut utiliser le resserrement d'intervalle, mais qui plus simplement peut consister à écrire le nombre sur sa calculatrice, puis `EXE`, `÷R`, `2`. Ensuite on peut appuyer sur `EXE` et à chaque fois noter le nombre en "R" (reste) qui correspond aux chiffres binaires **de droite à gauche**. Mais il en va aussi pour convertir vers d'autres bases que le binaire.

Par exemple, si je veux transformer 42 en binaire :

Ans = 42	(42 EXE)
Ans ÷R 2	(÷R 2 EXE)
R=0	(EXE)

R=1	(EXE)
R=0	(EXE)
R=1	(EXE)
R=0	(EXE)
R=1	(EXE)
0	

Donc si on remet le nombre à l'endroit (donc de haut en bas) ça donne **101010** qui correspond à  $2^5 + 2^3 + 2^1 = 32 + 8 + 2 = 42$

## Binaire vers hexadécimal (ou autre base de puissance de 2)

Convertir du binaire en hexadécimal (ou toute autre base en puissance de 2), il suffit d'aligner le binaire et l'hexadécimal et de répartir les bits autrement.

Donc si on veut convertir **101010** en hexadécimal, il faut trouver le nombre de bits (chiffre binaire) auquel un nombre hexadécimal correspond, soit 4, car  $\sqrt{16} = 4$ . On va donc diviser le nombre en groupe de 4 en partant de la droite (**bit de poids faible** → de plus petite valeur, l'inverse est appelé **bit de poids fort**)

10 1010

Ensuite on peut compléter à gauche avec des **0**

0010 1010

Enfin on peut convertir chaque bloc en chiffre hexadécimal :

BIN	0010	1010
DEC	2	10
HEX	2	A

Donc le nombre hexadécimal final est **2A**, qui correspond exactement à **101010** en binaire, soit **42** en décimal.

## Opérations bit à bit (bitwise)

Une opération bit à bit est une opération qui manipule les représentations binaires des données.

Il existe 2 catégories d'opérations, les opérations **logiques** (NOT AND OR et XOR) et les opérations de **décalage** (shift left, shift right)

Les opérations logiques vont être effectuées sur chaque bit, par exemple :

```
a 10101011
b 11001101
↓↓↓↓↓↓↓↓ l'opération AND est effectuée sur chaque bit
a&b 10001001
```

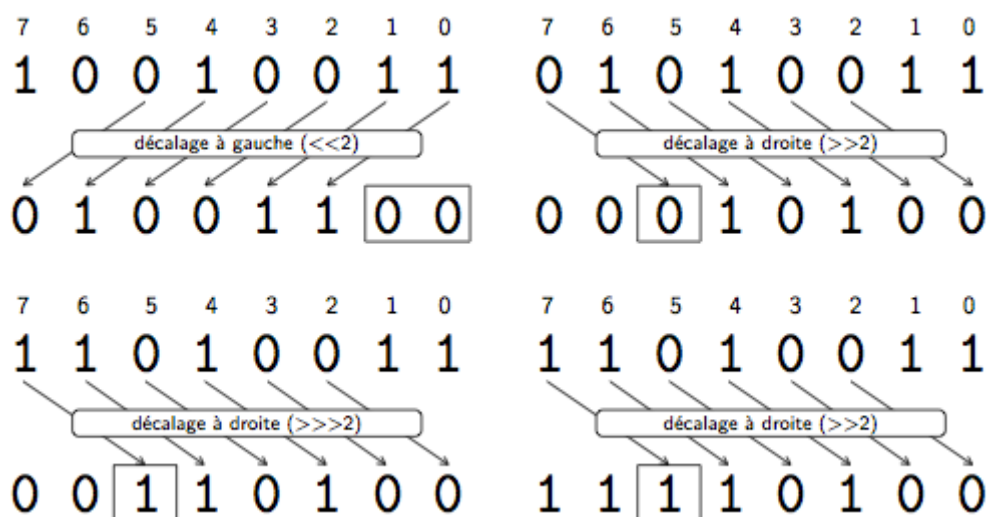
Pour voir comment fonctionne les autres connecteurs logiques, voir sur [cette synthèse du Q1](#)

Mais en résumé voici :

a	b	~a (NOT)	a & b (AND)	a   b (OR)	a ^ b (XOR)
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Pour ce qui est des opérations de décalages :

- Dans un **décalage à gauche (<<)** de **\$x\$ positions**, on supprime les **\$x\$ premiers bits** (si on est sur un nombre de bits limités) et on ajoute **\$x\$ 0** à la fin (bits de poids faible)
- Dans un **décalage à droite signé (ou arithmétique) (>>)** de **\$x\$ positions**, on copie le bit de poids fort (le plus à gauche) **\$x\$ fois** devant et on supprime les **\$x\$ derniers bits** (bits de poids faible)
- Dans un **décalage à droite non signé (>>>)** de **\$x\$ positions**, on ajoute **\$x\$ 0** devant et on supprime les **\$x\$ derniers chiffres**



Donc si on fait un `>>> 2` on doit faire un décalage à droite non signé 2 fois de suite. Par exemple pour faire `42 >>> 2`, on doit convertir 42 en binaire `101010` puis ajouter deux `0` au début (ce qui donne `00101010`) et supprimer les deux derniers chiffres (ce qui donne `001010`) puis reconvertir en décimal (ce qui donne 10)

```
42 >>> 2
= 101010 >>> 2
= 001010
= 10
```

## Les masques binaires

Les opérateurs sont souvent utilisés avec des "masques" les masques sont des successions binaires destinées à indiquer la position du/des bit(s) concernés par une certaine opération. On peut donc appliquer une opération sur une certaine partie de bits uniquement.

Les 2 masques les plus simples sont les masques qui utilisent des `1` pour signifier les positions particulières telles que `1000` pour spécifier la 3e position (les positions commencent par 0), un tel masque peut être généré avec le décalage `1 << 3`

L'autre type sont les masques inversés qui utilisent des `0` pour signifier les positions telles que `0111` pour spécifier la 3e position, un tel masque peut être généré en inversant le précédent `~(1 << 3)`

Les masques sont généralement désignés par leur valeur hexadécimale, donc `1000` sera appelé `0x8` et `0111` sera appelé `0x9`

## Extraire une information

Pour un exemple plus concret, disons que l'on a une couleur RGB. Une couleur RGB sur 32 bits où les octets (soit 8 bits chacun) représentent respectivement rouge, vert et bleu.

```
0x F66FFF
0b 11110110 01101111 11111111
```

Si on veut uniquement avoir le vert (soit les 8 bits du milieu, on peut d'abord utiliser un décalage à droite pour éliminer le bleu (la couleur de droite)

```
11110110 01101111 11111111 >> 8
=      11110110 01101111
```

Ensuite on peut utiliser un masque pour garder seulement les 8 premiers bits de poids faible (donc le vert). On veut donc avoir le masque `11111111` qui peut être représenté comme étant `0xFF`. Enfin, on peut extraire la couleur verte en faisant un opérateur AND entre notre couleur décalée et notre masque

```
11110110 01101111
&    11111111
    ↓↓↓↓↓↓↓↓
=    01101111
```

On sait donc que la couleur verte a une valeur binaire de `01101111`, soit une valeur hexadécimale de `6F` soit une valeur décimale de 111.

Pour faire tout ce qui est décrit plus tôt en Java :

```
int color = 0xF66FFF;
int mask = 0xFF;
int greenValue = (color >> 8) & mask; // donnera 0x6F
//      ^      ^ Ajout du masque pour uniquement avoir les 8 derniers bits (vert)
//      ^ Suppression des 8 derniers bits (bleu)
```

## Remplacer une information

On peut aussi utiliser les masques pour remplacer une information à une certaine position. Si on veut remplacer la couleur verte sans changer la couleur rouge ou bleu par exemple. Disons que l'on veut que la valeur de vert soit `0x45`

Pour cela, il va falloir reprendre notre nombre de départ.

```
0x F66FFF
0b 11110110 01101111 11111111
```

Puis on peut appliquer le masque `11111111 00000000 11111111` (`0xFF00FF`) pour faire en sorte que seul la couleur verte soit à 0. En utilisant l'opérateur AND on peut donc mettre le vert à 0.

```
11110110 01101111 11111111
& 11111111 00000000 11111111
  ↓↓↓↓↓↓↓↓ ↓↓↓↓↓↓↓↓ ↓↓↓↓↓↓↓↓
= 11110110 00000000 11111111
```

Ensuite on peut prendre notre nouveau vert `0x45` et le décaler pour qu'il soit à la même place que le vert dans la couleur de base

```
01000101 << 8
= 01000101 00000000
```

Enfin on peut combiner les deux avec un OR

```
11110110 00000000 11111111
| 01000101 00000000
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
= 01000101 11111111
```

Pour faire la même chose que tout ça en Java :

```
int color = 0xF66FFF; // Couleur de départ
int greenValue = 0x45; // Nouvelle valeur de vert à écrire
int mask = 0xFF00FF; // Masque pour remettre le vert d'origine à 0
int newColor = (color & mask) | (greenValue << 8)
//      ^      ^ ^ Décalage du vert au bon endroit
//      ^      ^ Ajout du vert
//      ^ Suppression du vert dans la couleur d'origine
```

## Tester la valeur / égalité de 2 choses

Si on veut savoir si les bits 8 et 9 du nombre de départs valent 1, on peut créer un masque `11 00000000` (`0x300`) et l'appliquer avec.

```
int color = 0xF66FFF;
int mask = 0x300;
boolean isEqual = (color & mask) == mask
// isEqual is True
```

Revision #8

Created 27 May 2023 08:42:15 by SnowCode

Updated 8 June 2023 06:21:39 by SnowCode