

Conversions, opérations binaires, décalages, masques

Les méthodes de conversions base10↔hexa et base10↔binaire sont supposées connues (divisions successives ou puissances décroissantes).
Les conversions hexa↔binaire peuvent être réalisées **rapidement** avec la table de correspondance ci-dessous.
Cette table n'est pas à connaître par coeur mais vous devez savoir la refaire (compter en base 2 de 0 à 15)

Correspondances décimal / binaire / hexadécimal

Correspondances		
Décimal	Binaire	Hexa
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Format des données ou variables entières sur une machine 32 ou 64 bits (ordinateurs récents) :

char = 1 octet = 8 bits = 2 digits hexa exemple : **0x2E**
short int = 2 octets = 16 bits = 4 digits hexa exemple : **0x3EFF**
int = 4 octets = 32 bits = 8 digits hexa exemple : **0x0300D010**

Conversion **Hexa ↔ Binaire** : 1 chiffre hexa correspond à un bloc de 4 chiffres binaires

0xE ↔ **0b1110**
0xAE3D ↔ **0b1010111000111101**

Conversion **Hexa ou Binaire ↔ Base 10** : pas de correspondance simple entre chiffres pour les valeurs > 15 ; utiliser une table des puissances de 2 pour décomposer ou utiliser une méthode de conversion par divisions successives

Notations pour préciser la base des constantes numériques en langage C :

- pas de préfixe → **base 10**
 - préfixe 0 → **octal** ATTENTION int x=077; // ici x vaut en fait 63 en base 10 !!
 - préfixe 0x → **hexa**

- préfixe 0b → **binaire** (avec certains outils, **notation non utilisable en C**)

Table puissances de 2

n	2 ⁿ	n	2 ⁿ
0	1	9	512
1	2	10	1024
2	4	11	2048
3	8	12	4096
4	16	13	8192
5	32	14	16384
6	64	15	32768
7	128	16	65536
8	256
			32 ~4,3 10 ⁹

Au sens large, un "mot binaire" est un ensemble de n bits, typiquement 8 bits (octet), 16, 32 ou 64 bits.
 Dans le monde des PCs, "un mot" (*word*) indique un mot sur 16 bits "un double mot" (*double-word*) indique un mot sur 32 bits.

Les bits de **poinds fort** sont les plus gauche, les bits de **poinds faible** sont à droite.

Si on a besoin de repérer la position d'un bit dans un mot binaire on considère une numérotation partant de 0 sur le bit de poids le plus faible (LSB = Least Significant Bit) jusqu'au bit de poids le plus fort (MSB = Most Significant Bit)

Exemple : valeur décimale 1343088032 contenue dans un int (mot de 32 bits)

Représentation compacte : hexadécimal

contenu (4 octets) : **0x500DE5A0**

Représentation détaillée : binaire

contenu (32 bits) : **0b01010000000011011110010110100000**
 n° de bit : 31 ... 0

Il faut bien comprendre qu'au niveau des **stockages**, des **transmissions**, des **traitements** (calculs...) sur les données dans un ordinateurs il n'y a généralement pas de conversions : **tous les nombres sont exploités en binaire selon la représentation en base 2**.

Dans un programme en console, les conversions ne se font généralement qu'au niveau des entrées (saisies) et des sorties (affichages)

// Exemple : programme Somme de 2 entiers saisis par l'utilisateur
 #include <stdio.h>

```
int main()
{
    int x,y,z;           // 3 espaces de stockage de 32 bits (4 octets) chacun

    // SAISIES
    scanf("%d",&x);     // Conversion d'une valeur décimale saisie par l'utilisateur en binaire, stockage binaire
    scanf("%d",&y);     // Conversion d'une valeur décimale saisie par l'utilisateur en binaire, stockage binaire

    // TRAITEMENTS : AUCUNE CONVERSION, POUR LA MACHINE TOUT SE FAIT EN BINAIRE
    z = x+y;           // Calcul de la somme en binaire, résultat en binaire, stockage en binaire

    // AFFICHAGES
    printf("%d\n",z); // Conversion de la valeur binaire en valeur décimale pour affichage (z reste en binaire !)

    return 0;
}
```

C'est la raison pour laquelle il est maladroit pour un programme qui doit déterminer si un nombre est un multiple de 5 ou pas de vouloir tester si le dernier chiffre est un 0 ou un 5 : dès que la saisie est terminée il n'y a plus de représentation décimale dans la machine, il n'y a pas de "dernier chiffre en base 10" disponible puisque ce qui est stocké c'est le codage en base 2 de la valeur.

Si la valeur entrée est 35, ce qui est stocké sera 00100011. Ce qu'on peut faire c'est un modulo 5 (constante 00000101), le processeur fait donc 00100011 modulo 00000101 ce qui donne 00000000 dont on peut vérifier dans un test (toujours en base 2) que c'est bien égal à 0 (00000000). Pour les constantes décimales ou hexadécimales (valeurs numériques qui apparaissent littéralement dans le code C) la conversion vers le binaire se fait directement au moment de la compilation.

Les fonctions d'entrées/sorties et le compilateur donnent à l'utilisateur et au programmeur l'illusion et le confort de vivre dans un monde en base 10. Mais dans certaines situations de programmation "bas niveau" il est important de pouvoir étudier l'aspect binaire des opérations réelles sur machine. Malheureusement le codage binaire n'est pas compact donc peu maniable pour les humains : c'est la raison de l'utilisation d'une **représentation hexadécimale** des valeurs, représentation qui reste proche du binaire (les conversions hexa/binaire sont immédiates) mais est beaucoup plus compact.

Représentation des entiers : signés ou non signés, complément à 2

Dans un `unsigned int` on ne peut stocker qu'un entier positif.

Dans un `int` on peut stocker indifféremment un entier positif ou négatif.

La convention qui associe une valeur entière à une et une seule combinaison binaire ne peut donc pas être la même dans les 2 cas.

Pour simplifier l'exposé, on traitera des entiers stockés sur 8 bits (`unsigned char` ou `char`) mais les même définitions sont valables en 16 ou 32 bits.

La convention la plus couramment employée pour représenter les nombres **entiers non signés** est de considérer la décomposition en base 2.

Exemple : on souhaite représenter la valeur 179 en non signé sur 8 bits, quelle est la combinaison binaire associée ?

On peut écrire $179 = 128 + 32 + 16 + 2 + 1 = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \rightarrow 0b10110011$ (0xB3)

Définition générale :

Pour l'octet composé des 8 bits suivants : $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

Valeur non signée = $b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$

La convention la plus couramment employée pour représenter les nombres **entiers signés** est de considérer que le **bit de poids le plus fort (MSB)** d'un mot est affecté d'un coefficient négatif dans la formule de la somme des puissances de 2.

Exemple : on souhaite représenter la valeur -77 en signé sur 8 bits, quelle est la combinaison binaire associée ?

On peut écrire $-77 = -128 + 32 + 16 + 2 + 1 = -1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \rightarrow 0b10110011$ (0xB3)

Définition générale :

Pour l'octet composé des 8 bits suivants : $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

Valeur signée = $-b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$

La raison de ce choix est qu'il permet d'utiliser des circuits arithmétiques compatibles à la fois avec la représentation signée et non-signée.

De plus le signe de la valeur apparaît immédiatement : le bit de poids le plus fort (MSB) est à 1 ssi la valeur est négative, on l'appelle "bit de signe"

Enfin il est facile d'obtenir la soustraction car l'inversion du signe se fait par une opération simple : le **complément à 2** qui consiste à inverser tous les bits du mot binaire (négation "bit à bit", 0 devient 1 et 1 devient 0) puis à ajouter 1 (+1 en binaire).

Exemple de complément à 2 pour inverser le signe :

on part de $0xB3 = 0b10110011$ qui représente -77 en convention signée.

La négation donne $0b01001100$ puis on ajoute 1 ce qui fait $0b01001101 = 0x4D$

Valeur signée résultante : $-0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 64 + 8 + 4 + 1 = +77$

Exercice : refaites un complément à 2 sur le code binaire de +77 pour vérifier qu'on retrouve bien -77

En utilisant si nécessaire le complément à 2 pour inverser le signe, écrire en hexa le contenu de l'octet correspondant aux **valeurs signées** suivantes :

$-12 = 0x..$ $12 = 0x..$ $-0x20 = 0x..$ $-0x56 = 0x..$

Quelles sont les représentations hexa de la **valeur signée -1** et de la **valeur non signée 255** stockées sur un octet, sur 16 bits, sur 32 bits ?

$-1 = 0x..$ $-1 = 0x....$ $-1 = 0x.....$ $255 = 0x..$ $255 = 0x....$ $255 = 0x.....$

Donner un extrait de la table d'interprétation sémantique des entiers sur 8 bits avec une colonne pour les valeurs en signé et une colonne en non signé : calculer ces valeurs pour : 00000000 00000001 00000010 00000011 ... 01111110 01111111 10000000 10000001 ... 11111101 11111110 11111111

Additions et soustractions binaires

Ici nous considérons une largeur fixe de 8 bits pour les mots binaires mais les processeurs récents fonctionnent plutôt en 32 bits ou 64 bits.

L'UAL du processeur comporte un circuit additionneur de largeur fixe, c'ad que le mot binaire résultant de l'addition est de même taille que les 2 mots opérands. Dans le cas où le résultat dépasse (la somme de 2 nombres à 8 chiffres pouvant parfois faire 9 chiffres) alors le chiffre le plus significatif qui dépasse est tout simplement ignoré. Dans ce cas le processeur émet un signal de retenue mais celui-ci est généralement ignoré par les programmes en C. Faire $250 + 250$ dans un `unsigned char` ne génère pas d'erreur, simplement le résultat est faux puisqu'il ne peut tenir sur 8 bits¹. Notons qu'en 32 bits (taille du int usuel sur nos machine) des valeurs < 1 Milliard ne poseront pas de problème de dépassement.

Les soustractions sont calculées par l'UAL du processeur en appliquant d'abord un complément à 2 pour inverser le signe puis une addition
Ainsi $9 - 3$ se calcul comme l'addition $9 + (-3)$

On peut additionner à la main les nombres binaires comme les décimaux, en opérant sur les chiffres de droite à gauche avec retenues éventuelles.

Sans retenue en haut de la colonne $0+0 \rightarrow 0$ $0+1 \rightarrow 1$ $1+0 \rightarrow 1$ $1+1 \rightarrow 0 + \text{retenue de } 1$

Avec retenue 1 en haut de la colonne $1+0+0 \rightarrow 1$ $1+0+1 \rightarrow 0 + \text{retenue de } 1$ $1+1+0 \rightarrow 0 + \text{retenue de } 1$ $1+1+1 \rightarrow 1 + \text{retenue de } 1$

Exemples (la retenue est indiquée avec ^)

	non signé	signé		non signé	signé
0b10010110 = 0x96 =	150	-106	0b00010010 = 0x12 =	18	18
+ 0b01001110 = 0x4E =	+ 78	+ 78	+ 0b11101011 = 0xEB =	+ 235	+ -21
0b11100100 = 0xB4 =	228	-28	0b11111101 = 0xFD =	253	-3

Exercices :

Compléter

0b00110011 = 0x.. =
+ 0b01011001 = 0x.. =
0b..... = 0x.. =

En non signé sur 8 bits combien fait 255 + 1 ? 0 - 1 ?

En signé sur 8 bits combien fait 127 + 1 ? -128 - 1 ?

1 En fait en unsigned char $250 + 250 = 244 = 500 + k \times 256$ avec $k = -1$: le résultat est correct mais considéré dans une arithmétique modulo 256...

Opérateurs Binaires "bit à bit" (*bitwise*)

Tableaux de vérité

Ces opérateurs binaires permettent de modifier et de tester un ou plusieurs bits d'une donnée.

Ce sont les opérateurs usuels de la logique booléenne : ET & OU | OUx (Exclusif) ^ NON (complément à 1) ~
L'UAL d'un processeur peut appliquer **simultanément** ces opérations entre tous les bits de 2 mots opérands.
Sur nos exemple nous utilisons des octets (variables de type `char`) mais ces opérateurs s'utilisent aussi bien sur les mots de 2 octets (variables de type `short int`) ou 4 octets (variables de type `int`)

1 0 1 0 & 1 1 0 0 1 0 0 0	1 0 1 0 1 1 0 0 1 1 1 0
1 0 1 0 ^ 1 1 0 0 0 1 1 0	~ 1 0 0 1

Exemples 0xE3 = 0b11100011 0xE3 = 0b11100011 0xE3 = 0b11100011
ET 0x25 = 0b00100101 OU 0x25 = 0b00100101 OUx 0x25 = 0b00100101 NON 0x25 = 0b00100101
0x21 = 0b00100001 0xE7 = 0b11100111 0xC6 = 0b11000110 0xDA = 0b11011010

Exercices 0x5A = 0b..... 0x5A = 0b..... 0x5A = 0b.....
ET 0x79 = 0b..... OU 0x79 = 0b..... OUx 0x79 = 0b..... NON 0x79 = 0b.....
0x.. = 0b..... 0x.. = 0b..... 0x.. = 0b..... 0x.. = 0b.....

Exemple en C : `printf("%x\n", 0xE3 ^ 0x25); // Affiche C6` (%x affiche un entier en hexadécimal)

Attention à ne pas confondre avec les opérateurs logiques pour les tests `&&` `||` `!`

ces derniers considèrent des valeurs de vérité sur la totalité des mots selon la convention mot=0 → Faux mot≠0 → Vrai

0xF0 & 0x0F donne 0x00 : opération bit à bit

0xF0 && 0x0F donne 0x01 : 0xF0≠0 (Vrai) ET 0x0F≠0 (Vrai) donne résultat≠0 (Vrai)

Décalages (*Shift*). Opérateurs << : décalage à gauche >> : décalage à droite

Le décalage logique à gauche déplace les valeurs des bits à l'intérieur du mot vers la gauche de n positions. Les valeurs précédentes dans les n bits de poids forts sont perdues et les n bits de poids faibles sont mis à 0. Le décalage logique à droite réalise l'opération en sens inverse.

Exemple : Valeur (en hexa) du mot 0xF0ABCD0E après un décalage logique de 4 bits à gauche : 0X0ABCD0E0

Que vaut (en hexa) le mot 0xF0ABCD0E après un décalage logique de 8 bits à droite ?

Que vaut (en hexa) le mot 0xF0ABCD0E après un décalage logique de 7 bits à gauche ?

Exemple en C : `printf("%x\n", 0xF0ABCD0E << 4); // Affiche ABCD0E0`

Utilisation des décalages pour faire des divisions ou des multiplications rapides :

- En décalant à gauche un nombre en base 10 de n positions et en complétant à droite par des 0 on réalise facilement une multiplication par 10ⁿ

Le même principe est valable en binaire et on obtient ainsi une multiplication par 2ⁿ avec un décalage à gauche de n bits

- En décalant à droite un nombre en base 10 de n positions et en jetant les chiffres qui étaient à droite on réalise facilement une division par 10ⁿ

Le même principe est valable en binaire et on obtient ainsi une division par 2ⁿ avec un décalage à droite de n bits (quotient entier, tronqué)

Exercice : convertir 17 en binaire, décaler à gauche d'une position, reconverter en base 10. Idem en partant de 17 avec un décalage à droite d'1 bit.

Cette idée est importante en programmation bas niveau car les opérations de décalage sont plus efficaces pour les circuits de calcul du processeur que les multiplications et surtout les divisions (le résultat s'obtient plus rapidement). Mais ça ne marche qu'avec des valeurs qui sont des puissances de 2.

Autre problème : le bit de signe en représentation signée sur les décalages à droite. Si on décale à droite une valeur signée négative et qu'on injecte des 0 sur les bits de poids fort alors le signe est perdu... on n'obtient pas la valeur correspondant au quotient d'une division par une puissance de 2... Pour cette raison le langage C tient compte du type signé ou non lors d'un décalage à droite : pour un type signé, si le bit de signe est à 1 (valeur effectivement négative) alors les n bits de poids fort sont mis à 1 et non à 0 (de cette manière on conserve le bon signe). En dehors de l'utilisation des décalages à droite pour faire des divisions, ce comportement est plutôt gênant, **on préférera donc déclarer les variables sur lesquelles on fait des décalages comme non signées** pour éviter de voir des 1 surgir inopinément par la gauche lors des décalages à droite. Politique à surveiller.

Attention erreur fréquente : comme pour toute opération le décalage **ne modifie pas** la variable sur laquelle il opère, comme pour toute opération il faut utiliser la valeur résultante (affecter ou afficher ou continuer à faire d'autres opérations avec...)

```
unsigned int j,k;
j=0x000000FF;

j>>1; // NE FAIT RIEN : j non modifié
k=j>>1; // k prend la valeur 0x0000007F, j non modifié
j=j<<2; // j modifié car apparaît explicitement à gauche de l'affectation (prend la valeur 0x000003FC)
```

Notations condensées pour les opérateurs en C

Il est fréquent d'écrire en C `x=x+1`; ou `x=x-1`; une notation spécifique existe rien que pour ça : `x++`; ou `x--`; ²

Il est aussi fréquent d'écrire `x=x+2`; ou `x=x*4`; ... ou de manière générale `x = x opération autreChose`;

Une écriture condensée est alors `x+=2`; ou `x*=4`; ... ou de manière générale `x opération= autreChose`;

Les opérations compatibles avec cette écriture sont arithmétiques et logiques : `+=` `--` `*` `/=` `%=` `<<=` `>>=` `&=` `|=` `^=`

`j=j<<2`; peut donc s'écrire `j<<=2`; Ce qui peut se voir dans du code professionnel, mais n'abusez pas de ça au détriment de la lisibilité ...

2 ATTENTION `x=x++`; est FAUX, ne MARCHE PAS, car l'incrémentation se fait après que la variable donne sa valeur à l'expression à droite, mais la variable à gauche reçoit cette valeur en dernier, donc **après** avoir été incrémenté x reçoit finalement la valeur d'avant l'incrémentation (d'où une valeur finalement inchangée)

Masques binaires

Les masques binaires permettent d'extraire l'état de certains bits d'un mot, ou de modifier l'état de certains bits sans modifier les autres.

Exemples : On dispose d'un octet déclaré en C par

```
unsigned char p;
```

On souhaite tester le bit 5 de l'octet et déclencher un traitement si il vaut 1

```
...  
if (p & 0x20) { traitement ... }
```

On souhaite mettre à 1 (*set*) les bits 1 et 2 de l'octet

```
p = p | 0x06;
```

On souhaite mettre à 0 (*reset*) les bits 1 et 2 de l'octet

```
p = p & 0xF9; // 0xF9 = ~0x06
```

On appelle masque la valeur constante binaire (généralement exprimée en hexa dans le code C) qui permet de sélectionner ou d'effectuer une opération sur des bits particulier d'un mot binaire.

L'opération au niveau du mot est une opération avec masque ou "masquage".

Ce sont des techniques fondamentales en programmation bas niveau

(programmation de systèmes embarqués, de pilotes de périphériques ...)

Exercices :

- Ré-écrivez les exemples précédents en utilisant les notations condensées

- On souhaite tester le bit 0 de l'octet et déclencher un traitement si il vaut 1

- On souhaite mettre à 1 (*set*) les bits 3, 4 et 5 de l'octet

- On souhaite mettre à 0 (*reset*) les bits 3, 4 et 5 de l'octet

- On souhaite inverser (*invert, toggle*) le bit 7 de l'octet

Il est fréquent en programmation bas niveau d'avoir dans un fichier en-tête (un `#include "bidule.h"`) une liste de `#define` qui donnent les masques correspondant aux positions de bits qu'on veut tester ou modifier.

Par exemple on suppose qu'un octet `unsigned char plane;` permet de contrôler les systèmes de sécurité d'un avion Certains bits sont lus par le programme (entrées) et certains bits commandent des actions (sorties) :

```
// Fichier Securite_Aviation.h  
#define PLANE_HIGH 0x01 // Le bit 0 correspond à l'avion en altitude  
#define PLANE_FIRE 0x02 // Le bit 1 correspond à l'avion en feu  
#define PLANE_FUMES 0x04 // Le bit 2 correspond à la cabine pleine de fumées  
#define PLANE_LGEAR 0x08 // Le bit 3 correspond au train d'atterrissage sorti  
#define PLANE_WARNING 0x10 // Le bit 4 déclenche l'allumage des lumières rouges qui clignotent  
#define PLANE_ALARM 0x20 // Le bit 5 déclenche une sirène stridente
```

ce qui permet ensuite d'écrire :

```
// Si en altitude et train d'atterrissage sorti, allumer lumières rouges  
if ( (plane & PLANE_HIGH) && (plane & PLANE_LGEAR) ) plane |= PLANE_WARNING;  
  
// Si l'avion est en feu ou la cabine pleine de fumée, allumer lumières rouges et sirène  
if ( plane & (PLANE_FIRE | PLANE_FUMES) ) plane |= PLANE_WARNING|PLANE_ALARM;
```

ceci est équivalent mais plus lisible que

```
if((plane&1)&&(plane&8)) plane|=16;  
if(plane&6) plane|=48;
```

Exercice : Ecrivez le code qui indique au système d'éteindre la lumière rouge quand elle est allumée mais que l'avion n'est plus en altitude ou que le train d'atterrissage est rentré.