On va terminer sur les concepts arithmétiques, pour passer au codage d'un entier dans un ordinateur.

1. De la base 2 vers la base 16 et inversement

Les chiffres d'un nombre en base 2 sont appelés *bits* (à ne pas confondre avec *byte* qui signifie *octet*, groupe de 8 bits).

Les chiffres de la base 16 sont appelés chiffres *hexadécimaux* : 0…9, A, B, C, D, E, F.

Il est très facile de convertir directement un nombre de la base 2 à la base 16 et inversement. Il y a un lien très fort entre ces deux bases. On va le mettre en évidence.

a. Complétez le tableau suivant

base 10	base 2	base 16
0	0000	0
1	0001	1
...	...	...
14	1110	E
15	1111	F

b. Si c'est possible sur votre poste, ouvrez un shell Python3 et tapez ces deux instructions :

```
for i in range(256):
    print(f"{i:3d}  {i:08b}  {i:02X}")
```

Elles affichent les 256 premiers entiers naturels en base 10, 2 et 16. C'est la continuation du tableau précédent.

On peut constater des régularités. Prenez par exemple le nombre 75. Il s'écrit `01001011` en base 2 et `4B` en base 16. Remarquez que `4` de la base 16 s'écrit `0100` en base 2 et `B` s'écrit `1011`. C'est à dire que le nombre en base 2 est le collage de la traduction en base 2 des chiffres hexadécimaux. Par exemple, le nombre `A5` en base 16 s'écrit en collant `A` écrit en base 2 sur 4 bits : `1010`, et `5` sur 4 bits : `0101`, donc `1010 0101`. Attention à bien écrire par quadruplets de 4 bits.

c. Écrivez les nombres suivants directement en base 2 (ne pas convertir en base 10) :
   i. $(E5)_{16}$
   ii. $(B27)_{16}$
   iii. $(1F38)_{16}$

La conversion de la base 2 vers la base 16 est à peine plus compliquée. Il faut grouper les bits 4 par 4 en commençant *par la droite* et en rajoutant les zéros nécessaires *à gauche*. Et ensuite traduire chaque groupe de 4 bits en hexadécimal.

d. Écrivez les nombres suivants directement en base 16 (ne pas convertir en base 10) :
   i. $(101001101101)_2$
   ii. $(111001111)_2$
   iii. $(111110100010110001)_2$

Pour information
La justification de ces non-calculs réside dans le fait qu'on peut factoriser 4 par 4 les termes de l'écriture polynomiale :

$$\begin{aligned} N &= \ ... \ + \ a_7.2^7 + a_6.2^6 + a_5.2^5 + a_4.2^4 \ + \ a_3.2^3 + a_2.2^2 + a_1.2^1 + a_0.2^0 \\ &= \ ... \ + \ (a_7.2^3 + a_6.2^2 + a_5.2^1 + a_4.2^0).2^4 \ + \ (a_3.2^3 + a_2.2^2 + a_1.2^1 + a_0.2^0) \\ &= \ ... \ + \ A_1.16^1 \ + \ A_0.16^0 \end{aligned}$$

avec
$$\begin{aligned} A_n &= ... \\ A_1 &= a_7.2^3 + a_6.2^2 + a_5.2^1 + a_4.2^0 \\ A_0 &= a_3.2^3 + a_2.2^2 + a_1.2^1 + a_0.2^0 \end{aligned}$$

Les A_i sont la décomposition polynomiale de nombres de 4 chiffres binaires, qu'on peut tous écrire avec un (seul) chiffre de la base 16.

2. Codage d'un entier naturel sur n bits

Dans un ordinateur, en interne, les entiers sont codés en base 2 sur un nombre fixe de bits, n. Ce nombre n qualifie généralement le processeur. Vous avez sûrement entendu parler de Windows 64 ou Windows 32. Le premier est programmé pour des processeurs 64 bits, ceux qu'on trouve actuellement. Windows 32 était pour les anciens processeurs, des années 1990, qui fonctionnaient « en 32 bits ». Les mots « processeur 64 bits ou 32 bits » signifient que c'est la taille des entiers « ordinaires » dans ces processeurs. Ils sont capables de faire des calculs les plus rapides avec ces entiers.

Attention, ces 32 et 64 bits ne sont pas la taille maximale d'un entier. Tous les processeurs, qu'ils soient 8, 16, 32 ou 64 bits sont capables de calculer avec de simples octets (8 bits), des mots de 16 bits (type `short` en langage C), des entiers de 32 bits (`int` en C), 64 bits (`long` en C), 128 bits (`long long` en C). Quand le processeur est trop petit, le compilateur C traduit et adapte les calculs. Il transforme les calculs que le processeur ne sait pas faire directement en algorithmes nettement plus lents. On verra comment dans la suite du cours.

Comment coder un entier naturel (positif ou nul) sur n bits ? C'est ultra simple : on l'écrit en base 2 et on essaie de le mettre sur n bits. Donc on rajoute des zéros à gauche si nécessaire. Par contre, si le nombre est trop grand, alors on ne peut pas le coder.

a. Codez les entiers suivants, écrits en base 2, sur le nombre de bits indiqué :
   i. $(1111001111010)_2$ sur 16 bits
   ii. $(0011011101)_2$ sur 8 bits
   iii. $(110110110)_2$ sur 12 bits
   iv. $(101001101)_2$ sur 8 bits
b. Codez les entiers suivants, écrits en base 10, sur le nombre de bits indiqué. Pour cela, il faut d'abord écrire chaque nombre en base 2, avec la méthode des divisions successives vue dans le TD1, puis mettre le résultat sur le nombre de bits indiqué.
   i. 29 sur 8 bits
   ii. 21 sur 4 bits
   iii. 13 sur 6 bits
c. Quel est le nombre N_{max} le plus grand qu'on peut coder sur n bits ? Pour le déterminer,

réfléchissez à son écriture. En base 10 par exemple, avec 2 chiffres, quel est le plus grand nombre qu'on peut écrire ? 99. Alors en base 2, si on a n bits, quel est ce nombre ? Déduisez la formule qui donne $N_{max} = f(n)$ soit par démonstration directe avec la décomposition en polynôme (pensez à lui ajouter 1 pour tout simplifier, comme quand on fait 99+1), soit par quelques exemples et une idée de la généralisation.

3. Additions

On veut maintenant essayer des additions en base 2. Ça marche comme en base 10 : addition des chiffres rang par rang, transfert de la retenue sur le rang suivant. La table d'addition, sans et avec retenue, a été donnée dans le cours d'amphi n°1.

a. Additionnez les nombres suivants :
   i. $(1100)_2 + (10)_2$
   ii. $(1011)_2 + (1010)_2$
   iii. $(1011)_2 + (1111)_2$
   iv. $(11110101)_2 + (10111)_2$

On se pose maintenant la question de limiter tous les nombres, valeurs et résultat, à n bits. Il peut se produire une erreur de calcul quand la dernière retenue ne peut pas être écrite, car la taille du résultat est limitée.

b. Additionnez les nombres suivants, sur le nombre de bits indiqué. Quand ce n'est pas possible, il faut garder seulement les n bits de poids faible, et enlever les bits de poids fort. Ces derniers sont perdus, et donc le résultat est faux.
   i. $(1101)_2 + (1100)_2$ sur 4 bits
   ii. $(0110)_2 + (1101)_2$ sur 4 bits
   iii. $(01110100)_2 + (11010110)_2$ sur 8 bits

Le résultat est faux, mais l'est-il tant que ça ?

c. Pour les deux premières question i. et ii., convertissez les nombres et leur somme sur 4 bits en base 10 et regardez ce qui manque pour que le résultat soit correct : seulement un bit de plus en poids fort : c'est justement cette retenue.

On verra que les processeurs gèrent cette retenue dans le cas d'un dépassement. Cela permet d'écrire des programmes qui font quand même des calculs justes.

4. Multiplications en base 2 (optionnel)

Contrairement à ce qu'on peut penser, c'est assez simple. Voici le principe, mais d'abord un concept : quels sont les rangs des bits à 1 dans un nombre ?

Soit $B = (b_{n-1}b_{n-2}...b_2b_1b_0)_2$. Les rangs des bits à 1 sont les indices i des b_i valant 1. Par exemple, pour $(1101)_2$, c'est 3,2 et 0, parce que b_3, b_2 et b_0 sont les 1 de $(1101)_2$.

a. Donner les rangs des bits à 1 dans les nombres suivants :
   i. $(110101)_2$
   ii. $(100110)_2$
   iii. $(101011)_2$

Soit A un nombre à multiplier par B. Le produit est une grande addition de plusieurs fois le nombre A auquel on rajoute des 0 à la droite (ce qui revient à le multiplier par une puissance de 2) ; le nombre de ces 0 sont les rangs des bits à 1 dans B. Soit par exemple $B = (1101)_2$, on a des 1 aux rangs 3, 2 et 0. Donc on va additionner $A[000] + A[00] + A$.

Voici des exemples, les zéros ajoutés sont soulignés :

- $A = (1101)_2$, $B = (101)_2$. Le résultat est $(1101\,\underline{00})_2 + (1101)_2 = (1000001)_2$
- $A = (101)_2$, $B = (1101)_2$. Le résultat est $(101\,\underline{000})_2 + (101\,\underline{00})_2 + (101)_2 = (1000001)_2$

b. Voici des exercices :
   i. $(111)_2 \times (11)_2$
   ii. $(110)_2 \times (101)_2$
   iii. $(11)_2 \times (111)_2$