**INSTITUT DE LA PROMOTION SUPERIEURE DU TRAVAIL**

# ARCHITECTURE DES MACHINES
# &
# SYSTEMES INFORMATIQUES

## SYSTEMES DE NUMERATION ET CODAGE

**A. M'ZOUGHI**

# SYSTEMES DE NUMERATION ET CODAGE

## I. Introduction

Tout système de numération est constitué par un ensemble de chiffres et de lettres qui permettent d'écrire tout nombre à l'aide de ces chiffres ou de ces lettres.
Le cardinal de cet ensemble est la base du système de numération. Tout nombre N peut s'écrire en base b comme suit :

$$N = a_n b^n + a_{n-1} b^{n-1} + a_{n-2} b^{n-2} + \ldots\ldots + a_i b^i + \ldots\ldots + a_1 b^1 + a_0$$

Ce qu'on peut écrire symboliquement lorsqu'on a fixé la base :

$$N = a_n a_{n-1} a_{n-2} \ldots\ldots a_i \ldots\ldots a_1 a_0$$

## II. Les bases.

### II. 1. Numération décimale

Ensemble : {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  Base : 10

Tout nombre décimal peut s'écrire :

$$N = a_n 10^n + a_{n-1} 10^{n-1} + a_{n-2} 10^{n-2} + \ldots\ldots + a_i 10^i + \ldots\ldots + a_1 10^1 + a_0$$

Exemple : l'écriture d'un nombre en base 10, comme par exemple 735, représente en fait :

$$7*10^2 + 3*10^1 + 5*10^0$$

Cette représentation est dite pondérée, car chaque chiffre apporte une contribution au nombre, fonction de son rang : 7 apporte les centaines, 5 les unités d'où la terminologie "position forts poids" et "faibles poids".

### II. 2. Numération binaire

Ensemble : {0, 1}  Base : 2

Tout nombre binaire peut s'écrire :

$$N = a_n 2^n + a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \ldots\ldots + a_i 2^i + \ldots\ldots + a_1 2^1 + a_0$$

Comme les $a_i$ ne peuvent prendre les valeurs 0 ou 1 la représentation est encore plus simple.

Exemple : en base 2, le nombre 100101 représente :

$$1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

***Passage du binaire au décimal :***

La conversion d'un nombre en base 2 en un nombre en base 10 est immédiate : il suffit d'additionner les produits partiels (dans la base 10).

Exemple : $(100101)_2$ est égal à $32(=2^5) + 0 + 0 + 4 (=2^2)+ 0 + 1(=2^0)$ soit $(37)_{10}$.

Si le nombre est fractionnaire, on opère de la même façon, les poids "derrière" la virgule représentant des poids fractionnaires :

Exemple : 10,0101 représente : $1*2^1+ 0*2^{*0}+\mathbf{0*2^{-1}+ 1*2^{-2} + 0*2^{-3} + 1*2^{-4}}$ soit 2 + 1/4 + 1/16 en base 10 qui donne 2+0,25+0,0625 = 2,3125.

***Passage du décimal au binaire :***

La conversion d'un nombre en base 10 en un nombre en base 2 se fait par des divisions successives par 2. En effet, il suffit de diviser le nombre $(N)_{10}$ par 2 et de recommencer cette même opération sur le quotient obtenu. Les restes de la division donnent les $a_i$ du nombre binaire en commençant par i=0.

Exemple : le nombre 37 (100101) pris en exemple ci-dessus peut aussi s'écrire sous la forme suivante :

37 = ((((((1*32) +0)*16)+0*8)+1*4)+0*2)+1

| | | | |
|---|---|---|---|
| d'où : | 37 divisé par 2 donne 18 et reste | 1 | on retrouve |
| | 18 divisé par 2 donne 9 et reste | 0 | la valeur |
| | 9 divisé par 2 donne 4 et reste | 1 | binaire 37 |
| | 4 divisé par 2 donne 2 et reste | 0 | de bas en haut. |
| | 2 divisé par 2 donne 1 et reste | 0 | |
| | 1 divisé par 2 donne 0 et reste | 1 | |

Le quotient 1 est le dernier poids (poids fort) 1.

Si la valeur à convertir est fractionnaire, la partie fractionnaire doit être multipliée par 2 et on garde chaque fois le bit "**entier**" que l'on enlève pour la prochaine itération 0,3125 donne :

| | | |
|---|---|---|
| 0,3125*2 = | 0,625 | ce poids est le plus fort |
| 0,625*2 = | 1,25 | |
| 0,25*2 = | 0,5 | |
| 0,5*2= | 1,0 | ce poids est le plus faible |

Dans les exemples ci-dessus on a vu comment passer de la base 2 à la base 10 et vice-versa. Pour passer d'une base quelconque **b** à la base 10 et vice-versa il suffit dans les algorithmes ci-dessus de remplacer 2 par **b**.

La base 2 est celle utilisée par l'ordinateur. Elle n'est pas commode du fait du grand nombre de bits à écrire (le bit est à la base 2 ce que le chiffre est à la base 10 : il permet

d'écrire des nombres. **BIT** est la contraction de **BI**nary digi**T**). Prendre la base 10 nous est plus naturel mais le passage entre ces 2 bases n'est pas immédiat.
Si l'on considère une base puissance de 2 alors le problème est beaucoup plus simple : il suffit de grouper les bits ou dégrouper en bits les chiffres par paquets correspondant à la puissance considérée.

***Comptage binaire :***

Avec n bits nous pouvons compter de 0 à $2^n$-1 donc au total $2^n$ combinaisons différentes un nombre binaire N composé par n bits aura ses valeurs comprises entre 0 et $2^n$-1 (représente la plus grande valeur possible de N).

Exemple : pour n = 4, N peut prendre les valeurs de 0 à $2^4$-1 (soit de 0 à 15).

| Poids $2^3 = 8$ | $2^2 = 4$ | $2^1 = 2$ | $2^0 = 1$ | Valeur décimale |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |
| 1 | 0 | 0 | 0 | 8 |
| 1 | 0 | 0 | 1 | 9 |
| 1 | 0 | 1 | 0 | 10 |
| 1 | 0 | 1 | 1 | 11 |
| 1 | 1 | 0 | 0 | 12 |
| 1 | 1 | 0 | 1 | 13 |
| 1 | 1 | 1 | 0 | 14 |
| 1 | 1 | 1 | 1 | 15 |

## II. 3. Numération octale

Ensemble : {0, 1, 2, 3, 4, 5, 6, 7} Base : 8.

Soit le nombre binaire :

$$N = a_n 2^n + a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_i 2^i + \ldots + a_1 2^1 + a_0 2^0$$

Groupons les termes sous la forme :

$$N = \ldots + (a_{3i+2}2^2 + a_{3i+1}2^1 + a_{3i}2^0)2^{3i} + \ldots + (a_5 2^2 + a_4 2^1 + a_3 2^0)2^{3x1} + (a_2 2^2 + a_1 2^1 + a_0 2^0)2^{3x0}$$

$$N = \ldots + (a_{3i+2}2^2 + a_{3i+1}2 + a_{3i})2^{3i} + \ldots + (a_5 2^2 + a_4 2 + a_3)2^3 + a_2 2^2 + a_1 2^1 + a_0$$

On peut alors écrire : $N = b_k 2^{3k} + ..... + b_j 2^{3j} + ..... + b_1 2^3 + b_0$

avec $b_0 = a_i 2^2 + a_1 2^1 + a_0$ et $b_i = a_{3i+2} 2^2 + a_{3i+1} 2 + a_{3i}$

soit encore $N = b_k 8^k + ..... + b_j 8^j + ..... + b_1 8 + b_0$.

Ainsi on peut passer très rapidement du binaire à l'octal et vice-versa.

Exemple : soit le nombre binaire 1010001111011 (13 bits).

Dans la base 8 on groupe les bits par 3 car ($8 = 2^{*3}$) en commençant par les poids faibles.

1 010 001 111 011
ce qui donne 1 2 1 7 3 on a $(12173)_8$ (5 chiffres, de 0 à 7).

Inversement si on part d'un nombre octal $(2347)_8$ par exemple on représente chacun de ses chiffres par 3 chiffres binaires (bits) : 2 3 4 7
010 011 100 111 on donc $(010011100111)_2$

### II. 4. Numération hexadécimale

Ensemble : {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F} Base : 16.

Les valeurs 10, 11 .... 15 sont codées A, B, ....F. Dans la base 16, on procède exactement comme précédemment mais on groupe les bits par 4 car ($16=2^{*4}$).

Exemple : soit le même nombre binaire de l'exemple ci-dessus 1010001111011 (13 bits).

En groupant les bits par 4 en commençant par les poids faibles.

1 0100 0111 1011
ce qui donne 1 4 7 B on a $(147B)_{16}$ (4 chiffres de 0 à F)

La base 16 est la plus utilisée aujourd'hui. Un nombre de 32 bits n'est plus codé que par 8 symboles dans la base 16, avec un passage d'une base à l'autre immédiat.

### III. Table de conversion

| Décimal | Binaire | Octal | Hexadécimal |
|---|---|---|---|
| 0 | 0000 | 00 | 0 |
| 1 | 0001 | 01 | 1 |
| 2 | 0010 | 02 | 2 |
| 3 | 0011 | 03 | 3 |
| 4 | 0100 | 04 | 4 |
| 5 | 0101 | 05 | 5 |
| 6 | 0110 | 06 | 6 |
| 7 | 0111 | 07 | 7 |

| | | | |
|---|---|---|---|
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |

## IV. Représentation des nombres signés dans un système digital

Les deux formes les plus utilisées pour représenter des nombres binaires signés sont les suivantes :

1) valeur absolue plus signe
2) complément à deux.

Comme les représentations sont différentes, les opérations arithmétiques sont également différentes.

Définition : Le complément logique ou complément à 1 d'un nombre est le nombre obtenu en remplaçant les uns par des zéros et inversement.

### IV. 1. Représentation en valeur absolue plus signe

Par définition, un nombre est représenté par un signe suivi de sa valeur absolue en binaire. La convention consiste à coder le signe par le bit de fort poids (0 pour le signe (+) et 1 pour le signe moins( -)).

Exemple : +7 est représenté par : 0000 0111
- 9 est représenté par : 1000 1001

il y a + 0 et - 0 qui sont respectivement 0000 0000 et 1000 0000.

#### IV. 1. 1. Addition de deux nombres N1 et N2 de même signe

***Règle 1 :*** Lorsque N1 et N2 ont le même signe, on additionne les bits des valeurs absolues et on prend le signe commun comme signe de la somme.

Exemple : (+19) + (+10) ou (-19) + (-10)

```
        001 0011     | N1 |
+       000 1010     | N2 |
        001 1101     | N1 | + | N2 |
```

Résultat +29 : 0001 1101 ou -29 : 1001 1101

### IV. 1. 2. Addition de deux nombres N1 et N2 de signes différents

***Règle 2 :*** Lorsque N1 et N2 ont des signes différents, un des deux nombres doit être complémenté avant l'addition. Ensuite, la valeur absolue du nombre non complémenté est ajoutée à la valeur absolue du nombre complémenté.

- S'il y a une retenue au delà du plus fort poids, elle est ajoutée au bit de plus faible poids de la somme. Le signe du résultat sera celui du nombre non complémenté.

- S'il n'y a pas de retenue, la somme sera complémentée et le signe du résultat sera celui du nombre qui a été complémenté.

Exemple : considérons l'addition de +19 (0001 0011) à -10 (1000 1010).

L'addition réalisée sera la suivante :

- On complémente -10 (0111 0101)

```
        001 0011      | N1 |
+       111 0101      | N2' |
    1   000 1000
                1
        000 1001      | N1 | + | N2' |
```

Résultat +9 : 0000 1001

- On complémente +19 (1110 1100)

```
        110 1100      | N1' |
+       000 1010      | N2 |
        111 0110      il n'y a pas de retenue  on complémente le résultat
        000 1001
        000 1001      | N1' | + | N2 |
```

Résultat +9 : 0000 1001

La représentation en valeur absolue plus signe d'un nombre comporte de nombreux inconvénients pour la réalisation des opérations et des opérateurs arithmétiques.

### IV. 2. Représentation en complément à 2

Dans ce système un nombre positif est représenté dans la même forme que dans n'importe quel autre système de représentation. Toutefois, les nombres négatifs se présenterons sous la forme du complément à 2. (c'est à dire le complément à la base). Elle possède toutes les propriétés voulues pour bien fonctionner.

Considérons des nombres de 8 bits. La moitié des représentations possibles code alors les positifs (0 à 127 soit 128 valeurs de 00000000 à 01111111), et l'autre moitié les négatifs (-128 à -1, soit les 128 valeurs restantes 10000000 à 11111111).

Exemple : -19 représenté sur 8 bits aura la forme suivante : 1110 1101. Elle est aussi obtenue ainsi : $-19 = 2^8 - 19 = 256 - 19 = 237 = 1110\ 1101 = 255 - 19 + 1$.

on sait que +19 = 00010011

```
                  11111111    On remarquera que la soustraction
                - 00010011    se fait sans jamais de retenue
                  11101100    C'est aussi le complément logique
et ajoutons 1 :   +        1
                  11101101    pour obtenir le complément à 2.
```

alors -19 = 11101101

***Remarques :***

- Dans ce type de représentation il n'y a qu'un zéro : 0000 0000
- Pour un nombre N codé sur n bits, le complément à 2 de N est égal à $2^n - |N|$.

Il y trois méthodes pour obtenir le complément à 2 d'un nombre binaire représenté sur n bits :

* on le soustrait à $2^n$ ;
* on prend le complément logique de ce nombre et on ajoute 1 ;
* on complémente à la base à partir des poids faibles jusqu'au premier chiffre non nul inclus et on complémente tous les autres chiffres à la base moins un.

### IV. 2. 1. Addition de deux nombres positifs

Cette addition ne présente pas d'intérêt particulier.

***Règle 1 :*** lorsque deux nombres N1 et N2 positifs sont additionnés (signe + valeur), si le bit de signe du résultat est égal à 1, c'est que nous avons un dépassement de capacité (oVerflow).

Exemples :

- addition de +19 et +10

```
        0001 0011        +19
+       0000 1010      + +10
        0001 1101        +29
```

- addition de +127 et +2

```
        0111 1111        +127
+       0000 0010      + + 2
        1000 0001        +129 dépassement de capacité.
```

### IV. 2. 2. Addition de deux nombres négatifs

***Règle 2 :*** lorsque deux nombres N1 et N2 négatifs sont additionnés (signe+valeur), la retenue peut être ignorée car elle est non significative. Cette retenue provient de l'addition des deux bits de signe. Toutefois, le bit de signe de la somme doit être égal à 1 puisqu'elle est négative. Si le bit de signe du résultat est égal à 0, c'est que nous avons un dépassement de capacité (oVerflow).

Exemples :

| | | | | |
|---|---|---|---|---|
| - addition de -19 et -10 | | 1110 1101 | -19 | |
| | + | 1111 0110 | + -10 | |
| | 1 | 1110 0011 | -29 | la retenue est sans valeur |
| - addition de -127 et -2 | | 1000 0001 | -127 | |
| | + | 1111 1110 | + - 2 | |
| | 1 | 0111 1111 | +129 | dépassement de capacité. |

### IV. 2. 3. Addition de deux nombres de signes opposés

***Règle 3 :*** Si une retenue est générée quand la somme est positive, elle est ignorée. Il n'y a pas de retenue quand la somme est négative.

La propriété du complément à 2 est de coder le signe dans le nombre et de suivre les règles de l'addition et de la soustraction.

Exemples : faire 19 - 10 revient à faire 19 + (-10) :

| | | | | |
|---|---|---|---|---|
| - addition de +19 et -10 | | 0001 0011 | +19 | 0001 0011 |
| | + | 1111 0110 | + - 10 | - 0000 1010 |
| la retenue est ignorée | 1 | 0000 1001 | + 9 | 0000 1001 |
| - addition de -19 et +10 | | 1110 1101 | - 19 | |
| | + | 0000 1010 | + +10 | |
| | | 1111 0111 | - 9 | |

**Remarque :** cette représentation des nombres négatifs peut être étendue à d'autres bases.

### V. Nombres signés et nombres non signés. Débordements.

On sait faire une addition de nombres non signés. Avec 8 bits on peut compter de 0 à 255, soit de 00000000 à 11111111 en base 2.
Considérons diverses additions binaires, en considérant les deux représentations, signée et non signée :

| | | |
|---|---|---|
| 5 | 00000101 | +5 |
| 9 | + 00001001 | +9 |
| 14 | 00001110 | +14 |

Le résultat est exact dans les deux représentations. Est-ce toujours le cas ?

autre exemple :

| | | |
|---|---|---|
| 100 | 01100100 | +100 |
| 100 | + 01100100 | +100 |
| 200 | 11001000 | - 56 ! |

l'addition non signée est exacte car le résultat ne dépasse pas 255 mais l'addition signée est fausse car on a débordé, le résultat ne devant pas dépasser 127.
On trouvera des exemples ou le résultat est inverse : débordement en non signé, exact en signé. Et également des exemples où le résultat déborde dans les deux cas.
On voit ainsi qu'une addition binaire est indépendante de l'interprétation que l'on fait des nombres. On dit souvent qu'un calculateur "travaille" en complément à 2. En fait il travaille simplement en binaire et c'est nous, utilisateurs, qui donnons une interprétation aux nombres : signés ou non signés.
Le seul problème concerne les débordements, qui ne sont pas les mêmes dans les deux cas. Comme l'ordinateur n'est pas en mesure de connaître notre interprétation, on verra qu'il nous donne cette information de débordement pour les deux cas, à l'aide de ce que l'on appellera indicateurs de retenue et de débordement.

Exercice : effectuer en complément à 2 les additions suivantes :
a) 57 + 28; b) (-57) + (-28); c) 127 + 8; d) (-1) + (-2); e) (-127) + (-65)

## VI. Les nombres Décimaux Codés Binaire.

Le passage de la base 10 à la base 2 (et réciproquement) n'est pas immédiat. Une représentation intermédiaire est possible qui code le nombre en base 10 chiffre à chiffre en base 2 :

| 3 | 2 | 7 |
|---|---|---|
| 0011 | 0010 | 0111 |

Ce passage semble équivalent au passage de la base 16 à la base 2, mais seuls les chiffres 0 à 9 sont utilisés. La lecture de ce nombre codé en "Décimal Codé Binaire ou DCB" est immédiate, mais on voit que, pour chaque chiffre, seules 10 combinaisons sur les 16 possibles sont utilisées. Ce codage a donc un mauvais rendement et de plus les opérations arithmétiques, lorsqu'elles sont possibles dans cette représentation, sont plus lentes qu'en binaire.
Cette représentation est utile lorsque l'on a peu d'opérations à effectuer sur les nombres, ce qui est le cas en gestion par exemple. On évite alors les passages par le binaire pur.

## VII. les nombres fractionnaires.

Coder des nombres fractionnaires revient à tenir compte de la position d'une virgule, fictive. Il faut simplement vérifier que lors d'une addition où d'une soustraction les virgules soient "alignées". La multiplication et la division mènent à des règles différentes, mais simples.
Cette virgule fictive revient simplement à changer d'échelle : il suffit de gérer un compte en centimes pour éviter la virgule (elle n'apparaît qu'à l'édition du compte, ou le symbole, est ajouté en bonne position), de calculer une valeur de courant en milliampère pour éviter les 3 chiffres après la virgule, ...etc.

cependant si les valeurs à représenter sont infiniment petites ou infiniment grandes une autre représentation est utilisée, dite "flottante" ou "réelle".

**VIII. Les nombres réels.**

Les nombres réels ou flottants permettent de travailler avec des nombres fractionnaires du type :

$$\pm m,n * 10^{\pm e}$$

Cette représentation est surtout utilisée en calcul scientifique. Un mot mémoire code alors la mantisse **(m,n)**, son signe et la position de la virgule, la caractéristique **(e)** et son signe.

Exemple : la représentation normalisée de $125,34*10^{13}$ est $0,12534*10^{16}$.
Ce codage est réalisé soit sur 32 bits (format court), soit sur 64 bits (format long).

- format court :

31 30 23 22 0

| S | EXPOSANT (e) | MANTISSE (m,n) |
|---|---|---|

- format long :

63 62 52 51 0

| S | EXPOSANT (e) | MANTISSE (m,n) |
|---|---|---|

Les opérations sur ce type de nombres sont plus difficile à réaliser, prennent un temps beaucoup plus long, et les conversions ne sont pas simples.
La présence d'un co-processeur flottant permet de réaliser ces opérations de façon efficace.

**IX. Le code ASCII.**

Le code ASCII ( American Standard Code Interface Interchange) est le code unanimement reconnu pour coder les divers symboles alphanumériques (alphabet plus chiffres plus symboles spéciaux), il comporte :

1) 26 lettres minuscules
2) 26 lettres majuscules
3) les 10 chiffres 0 à 9.
4) environ 25 caractères spéciaux comprenant +, -, *, /, #, % …

Ceci représente 87 caractères qui nécessitent au moins 7 bits pour leur codage. En réalité un bit supplémentaire est rajouté pour coder la parité permettant de détecter les erreurs lors de la transmission du code. Donc 8 bits sont utilisés.

Dans ce code le chiffre 0 à pour valeur $30 ($ désigne la base 16, soit 48 en base 10), le chiffre 1 vaut $31 et ainsi de suite jusqu'à 9 qui vaut $39.

Une chaîne de caractères ASCII code un nombre si chacun de ses caractères est un code compris entre $30 et $39. Cette chaîne ASCII représente un nombre "rentré" à partir d'un clavier, un nombre prêt à être édité ou prêt à être envoyé sur une liaison téléinformatique.

***Parité :***

Détection des erreurs par la méthode de parité

**Parité paire :** prend "1" si la somme des bits à "1" avec bit de parité inclus est **paire** dans le cas contraire elle prend "0".

| | | | |
|---|---|---|---|
| Exemple : | C = 100 0011 | 1 | 100 0011 |
| | A = 100 0001 | 0 | 100 0001 |

**Parité impaire :** prend "1" si la somme des bits à "1" avec bit de parité inclus est **impaire** dans le cas contraire elle prend "0".

| | | | |
|---|---|---|---|
| Exemple : | C = 100 0011 | 0 | 100 0011 |
| | A = 100 0001 | 1 | 100 0001 |

**X. Algorithmes de conversions.**

Considérons le nombre 247.
Il vient d'être "tapé" au clavier et se retrouve donc sous forme d'une chaîne ASCII dans 3 octets : $323437
En éliminant le "3" du demi-octet de forts poids et en compactant les demi-octets restants on code le nombre en DCB : $0247
qui occupe 2 octets (on considère que l'unité minimale de manipulation est l'octet. Le demi-octet de forts poids est donc ici à 0).
Pour passer ce nombre en binaire, on peut multiplier chaque chiffre DCB par son poids et cumuler tous les résultats obtenus.
Pour un nombre d'un chiffre le code binaire est le même que le code DCB. Chaque chiffre pris individuellement est donc binaire et s'il est multiplié par son poids en binaire, le résultat est binaire et le cumul également par conséquences :

| | | | |
|---|---|---|---|
| 7*1=7 | soit | 0111*1 = | 0111 |
| 4*10=40 | soit | 0100*1010 = | 101000 |
| 2*100=200 | soit | 0010*11001000 = | 11001000 |
| | d'où un total cumulé de | | 11110111 |

qui représente 247 en base 2 (128 + 64 + 32 + 16 + 4 + 2 + 1 ou 255 - 8), résultat recherché et qui est codé sur un octet dont la valeur binaire est 1111 0111 soit F7 en base 16 (F7 représente 15*16 + 7 soit 247).

La procédure inverse se fait par divisions successives :
11110111 divisé par 1100100 (soit 100 en base 10) donne un quotient de 2 (en binaire ou DCB puisque le quotient est toujours d'un chiffre, le nombre considéré étant à 3 chiffres).

Le reste 42 c'est à dire 101111 est divisé par 1010 (soit 10 en base 10) qui donne un quotient de 4 (binaire ou DCB) et un reste de 2 (binaire ou DCB).
En regroupant les résultat on obtient le chiffre $247 codé en DCB.
En plaçant un chiffre par octet et en ajoutant 3 dans le demi-octet en position forts poids on retrouve la chaîne ASCII de départ, que l'on peut éditer.

Les algorithmes ci-dessus sont parfaitement exploitables sur calculateur. Il existe cependant des algorithmes plus systématiques, par exemple en utilisant le schéma de Horner, ou en faisant usage d'une table de pondération.