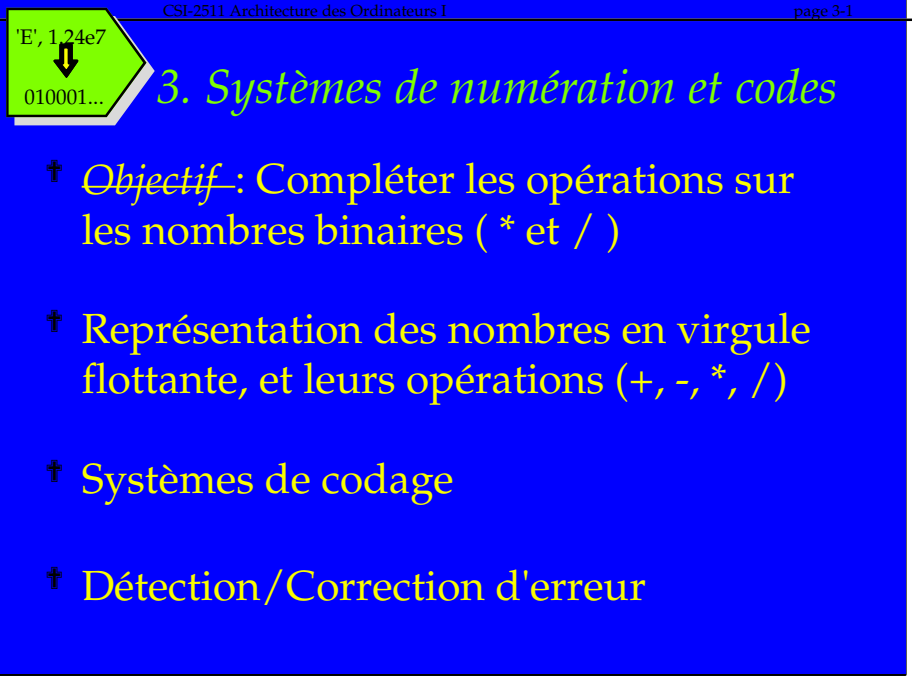
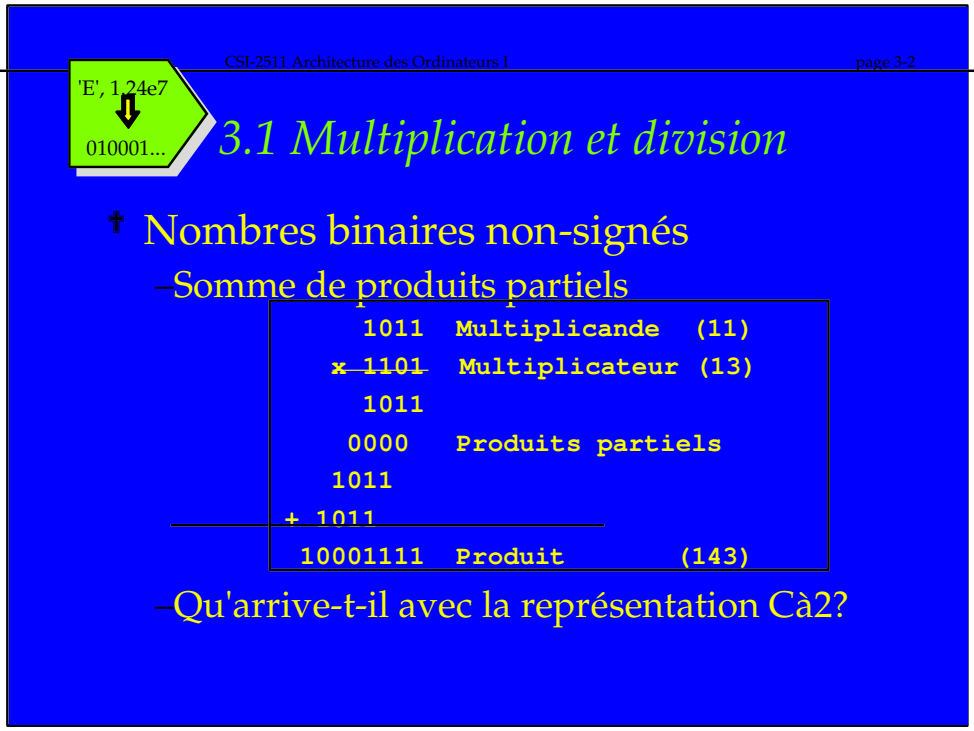


3.1 Multiplication et division
- Multiplication de nombres binaires non-signés
- Multiplication de nombres binaires en Cà2
- Division de nombres binaires non-signés
- Division de nombres binaires en Cà2

3.2 Nombres en virgule flottante
- Représentation et limites
- Addition et soustraction
- Multiplication et division

3.3 Systèmes de codage
- Représentation binaire
- Codages décimaux et alphanumériques

3.4 Détection/Correction d'erreur
- Codage et décodage; pourquoi?
- Bit de parité, 2 parmi 5
- Code Hamming



La multiplication des nombres binaires non-signées est simple:

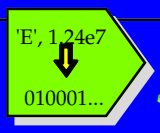
- Nous utilisons la même technique que pour les nombres décimaux: une somme de produits partiels.
- Ces produits sont simples: 0 ou le multiplicande.
- La multiplication de 2 entiers sur n bits donne un produit d'au plus $2n$ bits.
- Un algorithme est présenté à la figure 7-7 en annexe.

Est-ce aussi simple avec la représentation complément à 2?

```
  1011  Multiplicande  (-5)
x 1101  Multiplicateur (-3)
  1011
 0000   Produits partiels
1011
+ 1011
10001111  Produit       (-113)
```

Une solution?

- Convertir les nombres négatifs, multiplier les 2 nombres, ajuster (complémenter) le produit selon les signes des 2 nombres à multiplier
- L'algorithme de Booth.

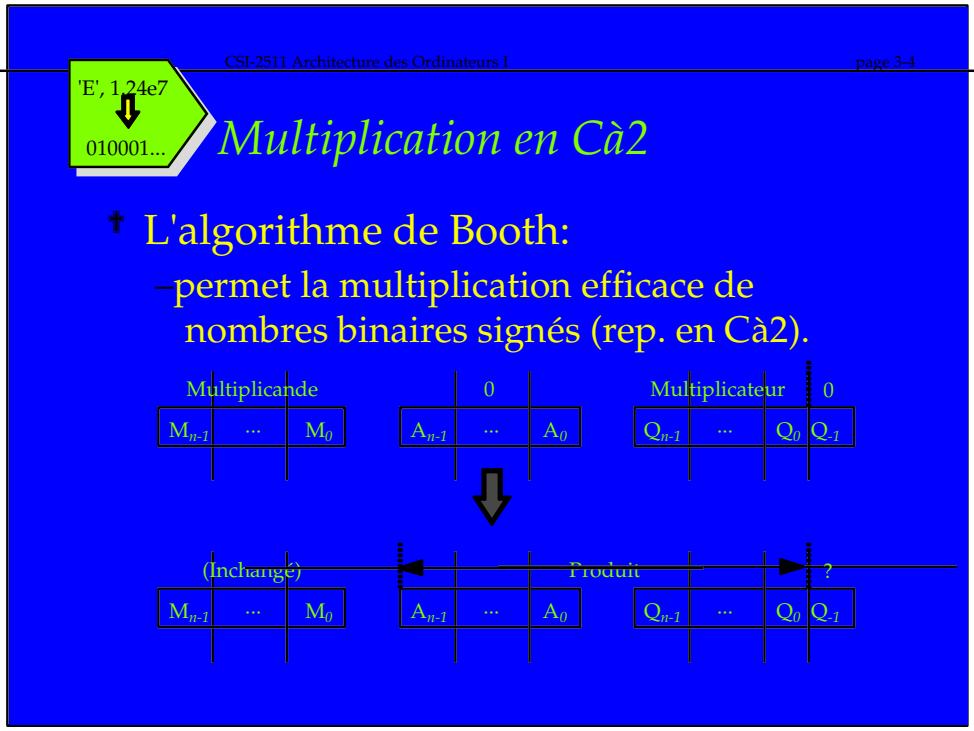


*3.1 Multiplication et division **

- Avec la représentation Cà2?
  - Somme de produits partiels

```
     1011  Multiplicande  (-5)
   x 1101  Multiplicateur (-3)
     1011
    0000   Produits partiels
   1011
+ 1011
 10001111  Produit        (-113)
```

  - Marche pas!!!



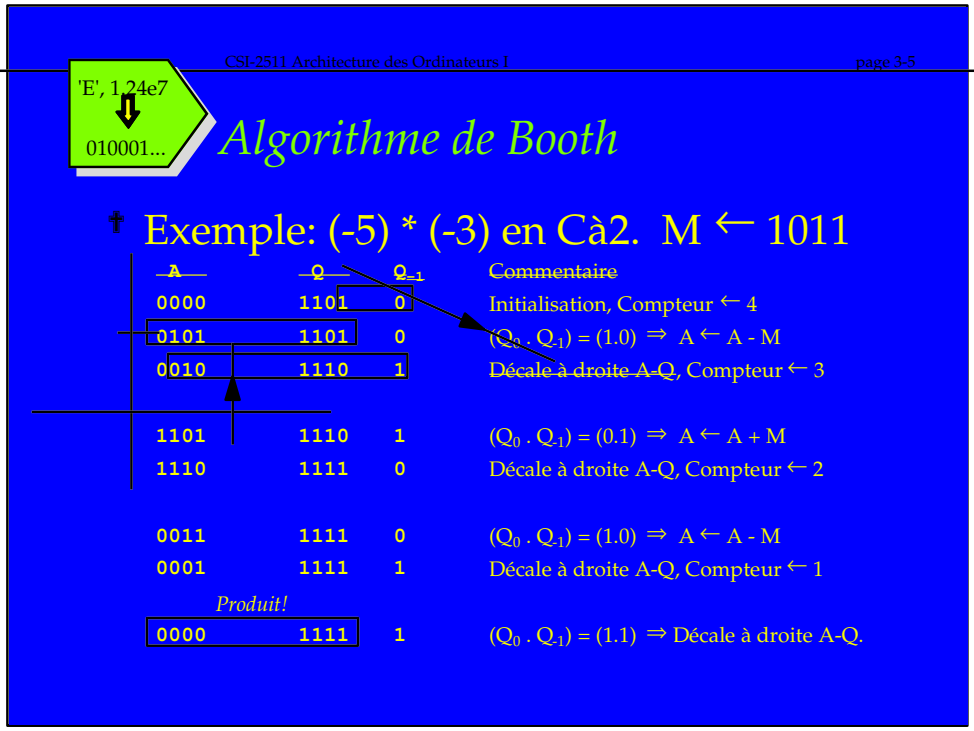
Algorithme de Booth (avec 3 registres: M, A et Q):

M ← Multiplicande, Q ← Multiplicateur, A ← 0, Q_{-1} ← 0, Compteur ← *n*
TANT QUE (Compteur > 0)
SI (Q_0 . Q_{-1}) = (1.0)
ALORS A ← A - M
SINON SI (Q_0 . Q_{-1}) = (0.1)
ALORS A ← A + M
FIN SI
FIN SI
Décale vers la droite les registres combinés A-Q (incluant Q_1)
Compteur ← Compteur - 1
FIN TANT QUE

Pourquoi cet algorithme fonctionne-t-il?

Sans entrer dans les détails, cet algorithme considère les nombres sous une forme (2^{n+1} - 2^{n-K} + ...) au lieu de (2^n + 2^{n-1} + 2^{n-2} ... + 2^{n-K} + ...) comme c'est le cas habituellement pour les nombres non-signés. Ceci a l'avantage de traiter les nombres en complément à 2 plus facilement (et souvent de façon plus compacte). Par exemple, le nombre 30 sur 8 bits (00011110) est traité comme (2^5 - 2^1) plutôt que comme (2^4 + 2^3 + 2^2 + 2^1).

Avec n=4 bits, on remarque aussi que le codage Cà2 est (-8)421 au lieu de 8421 pour les nombres non-signés (à voir plus loin dans ce chapitre).



Exemple: $(-5)_{Cà2} * (-3)_{Cà2} = (15)_{Cà2}$

Une autre façon de comprendre cet exemple est d'utiliser la théorie derrière cet algorithme (voir annexe):

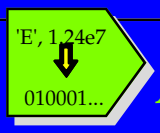
```
     1011    (-5 sur 4 bits)
   x 1101    (-3 sur 4 bits)
 00000101    (Cas 1.0, (-M)*2^0)
 1111011     (Cas 0.1, (+M)*2^1) (M est sur plus de 4 bits)
 000101      (Cas 1.0, (-M)*2^2)
+00000       (Cas 1.1, (0)*2^3)
 00001111    (produit: 15 sur 8 bits).
```

Cet exemple montre bien que le problème de cette représentation trouvé précédemment est réglé. En est-il de même avec:

- $(5)_{Cà2} * (3)_{Cà2} =$

- $(-5)_{Cà2} * (3)_{Cà2} =$

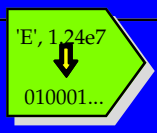
- $(5)_{Cà2} * (-3)_{Cà2} =$



*Algorithme de Booth **

- Exemple: (-5) * (+3) en Cà2. M ← 1011

~~A~~	~~Q~~	~~Q_{-1}~~	~~Commentaire~~
0000	0011	0	Initialisation, Compteur ← 4
0101	0011	0	$(Q_0 . Q_{-1}) = (1.0) \Rightarrow$ A ← A - M
0010	1001	1	~~Décale à droite A-Q~~, Compteur ← 3
0001	0100	1	$(Q_0 . Q_{-1}) = (1.1) \Rightarrow$ Décale à droite A-Q, Compteur ← 2
1100	0100	1	$(Q_0 . Q_{-1}) = (0.1) \Rightarrow$ A ← A + M
1110	0010	0	Décale à droite A-Q, Compteur ← 1
Produit!			
1111	0001	0	$(Q_0 . Q_{-1}) = (1.1) \Rightarrow$ Décale à droite A-Q, Compteur ← 0



Division de nombres binaires

- Un peu plus difficile que la multiplication
- Pour les nombres non-signés
  - –Utiliser la méthode traditionnelle des restes partiels.
- Pour les nombres signés (Cà2)
  - –Faites votre possible...!
  - –Voir exemple en annexe.

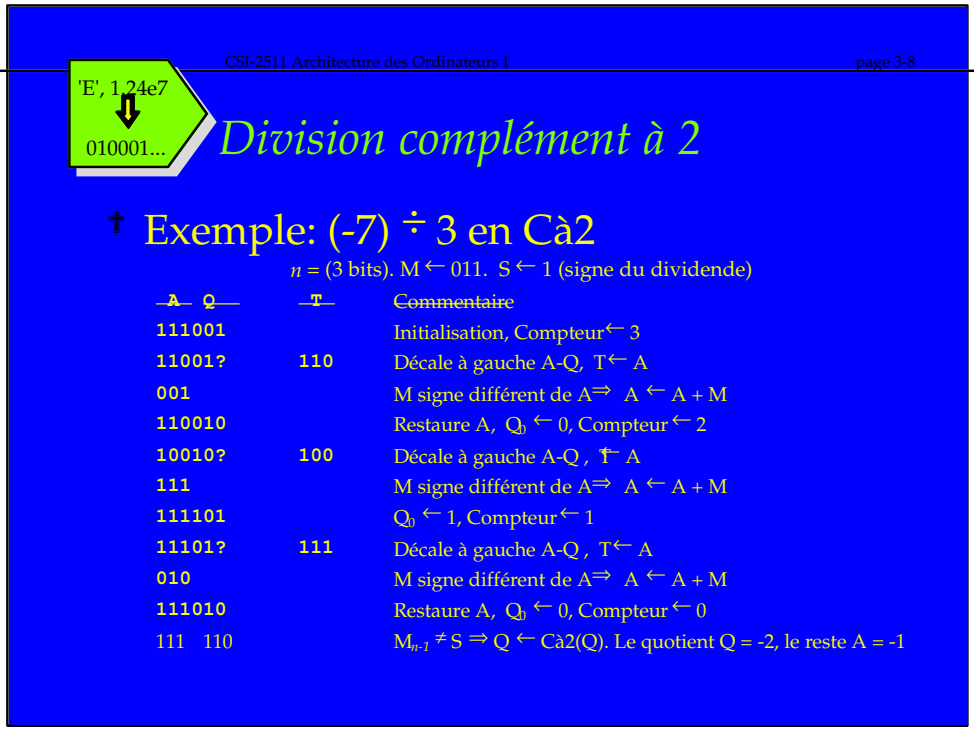
Traditionnellement: 177 ÷ 8 = 22 reste 1

Dividende	10110001	1000	*Diviseur*
	−1000	10110	*Quotient*
	1100		
	−1000		
	1000		
	−1000		
	01	*Reste*	

Algorithme de division: non-signés (3 registres: A, M, Q sans Q_{-1})

M ← Diviseur, Q ← Dividende, A ← 0, Compteur ← *n*
TANT QUE (Compteur > 0)
    Décale à gauche les registres combinés A-Q
    SI A < M
        ALORS Q_0 ← 0
        SINON A ← A - M, Q_0 ← 1
        FIN SI
    Compteur ← Compteur -1
    FIN TANT QUE
(*Le quotient se retrouve dans Q et le reste dans A*)

Exercice: Sur 4 bits, essayez 11 ÷ 3, 4 ÷ 5, et 13 ÷ 7.

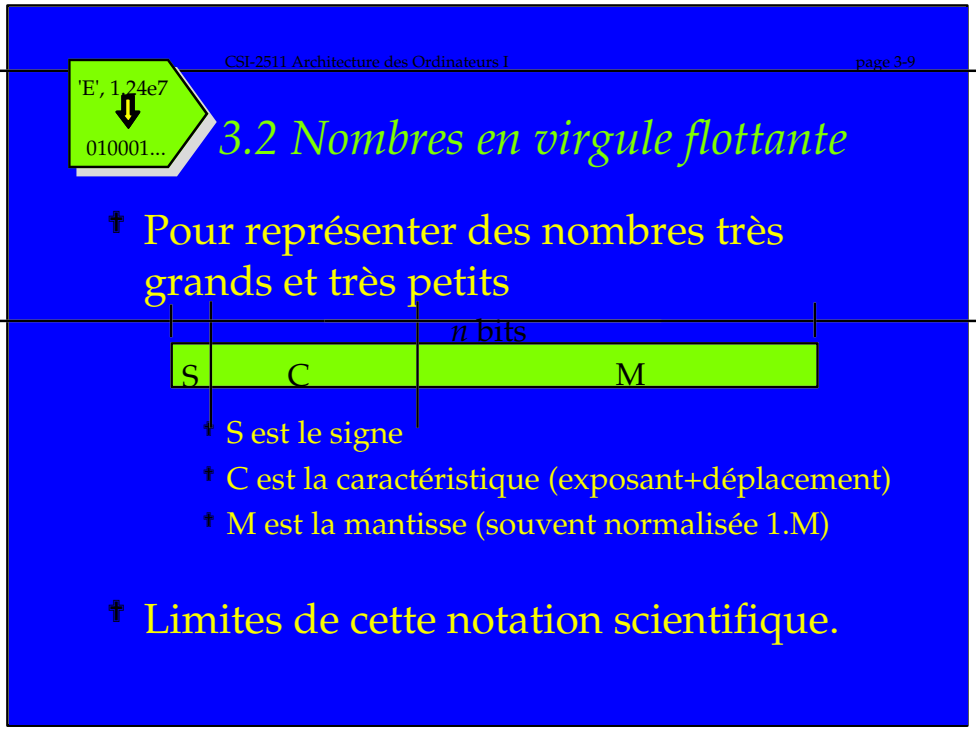


Algorithme de division: signés Cà2 (4 registres: A, M, Q sans Q_{-1}, T)

Le dividende est représenté en Cà2 sur $2n$ bits. Les registres A-Q sont combinés initialement pour recevoir ce dividende.

M ← Diviseur, A-Q ← Dividende, Compteur ← n, S ← A_{n-1} (*signe du dividende, 1 bit*)
TANT QUE (Compteur > 0)
    Décale à gauche les registres combinés A-Q
    T ← A (*registre temporaire, pourrait être réalisé autrement*)
    SI $A_{n-1} = M_{n-1}$
        ALORS A ← A - M
        SINON A ← A + M
        FIN SI
    SI ($A_{n-1} = T_{n-1}$) OU (A = 0 ET Q = 0)
        ALORS Q_0 ← 1
        SINON Q_0 ← 0, A ← T (*restaure le contenu de A*)
        FIN SI
    Compteur ← Compteur -1
    FIN TANT QUE
SI ($S \neq M_{n-1}$)
    ALORS Q ← Complément-à-2(Q)
    FIN SI

(*Le reste dans A. Le quotient se retrouve dans Q. Algorithme à tester...!*)



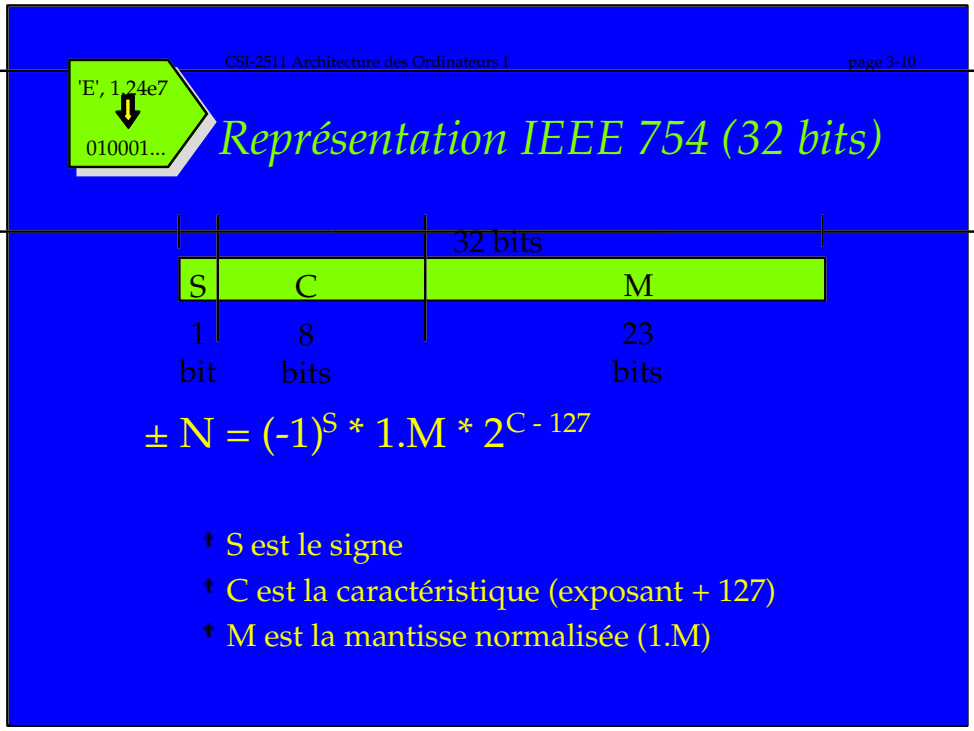
La représentation en *virgule flottante* se veut le reflet de la notation scientifique utilisée courrament: $4,5 \times 10^{27}$; $-1,3453 \times 10^{-43}$...

Le plus souvent, nous rencontrons:

- Un bit de signe (S), permettant les nombres positifs et négatifs.
- Quelques bits pour l'exposant (C). La *caractéristique* est l'exposant auquel un nombre (*bias*) a été ajouté. Par exemple: C = exposant + 127. Ceci permet la représentation d'exposants négatifs et positifs.
- Beaucoup de bits pour la mantisse (M). Celle-ci est bien souvent normalisée afin d'éviter d'avoir de multiples représentations du même nombre. Par exemple, le premier nombre est normalisée: $1,23 \times 10^{27}$ vs $12,3 \times 10^{26}$.

Dépassement de capacité (figure 7-17 en annexe)

- Comme il y a un nombre fini de bits, il y a un nombre fini de nombres pouvant être représentés. Les limites sont atteintes lorsque les valeurs à représenter sont trop grandes (*overflow*) ou trop près de 0 (*underflow*), et ce qu'elles soient positives ou négatives.
- Notez aussi que la densité des valeurs représentées varie selon la magnitude du nombre (fig. 7-18). Ceci cause certains problèmes de précision comme: $1 \times 10^{35} + 1 \times 10^{-35} = 1 \times 10^{35}$!!!



La norme IEEE 754 permet de représenter un nombre *réel* sous la forme:

$$\pm N = (-1)^S * 1.M * 2^{C - 127}$$

S (1 bit) est le bit signe: 0 pour positif et 1 pour négatif

C (8 bits) est la caractéristique (exposant augmenté de 127)

$0 < C < 255$ ($C = 0 \rightarrow$ zéro, $C = 255 \rightarrow$ infini)

$-126 \leq C - 127 \leq +127$ (exposant réel)

M (23 bits) est la mantisse normalisée de forme 1.M

Le bit (1) le plus significatif (à gauche) est caché et le point binaire est situé immédiatement à sa droite.

$0 \leq M \leq 2^{23} - 1$

$1.0 \leq 1.M \leq 1.(2^{23} - 1)$

Ex.: 1 00000000 00000000000000000000000 = -0

1 01111111 00000000000000000000000 = -1

0 01111111 11111111111111111111111 = _____

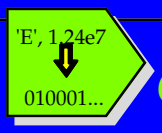
0 10000001 11100000000000000000000 = _____

_____ = 8

_____ = 35.5

0 11111111 10111100100011011111001 = _____

Plusieurs autres normes existent (fig. 7-19). La table 7-2 indique leurs limitations. La table 7-3 montre la représentation des 0, ∞ et *NotANumber*.



*Conversion IEEE754 - Réel **

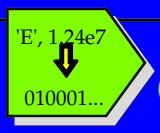
0 10000001 11100000000000000000000

S = 0, donc nombre positif

C = 129, donc exposant = C-127 = 2

1.M = 1.111

+ $1.111 \times 2^2 = 111.1 \times 2^0 =$ ~~7.5~~



*Conversion Réel - IEEE754 **

$35.5 = (?)_{\text{IEEE 754}}$

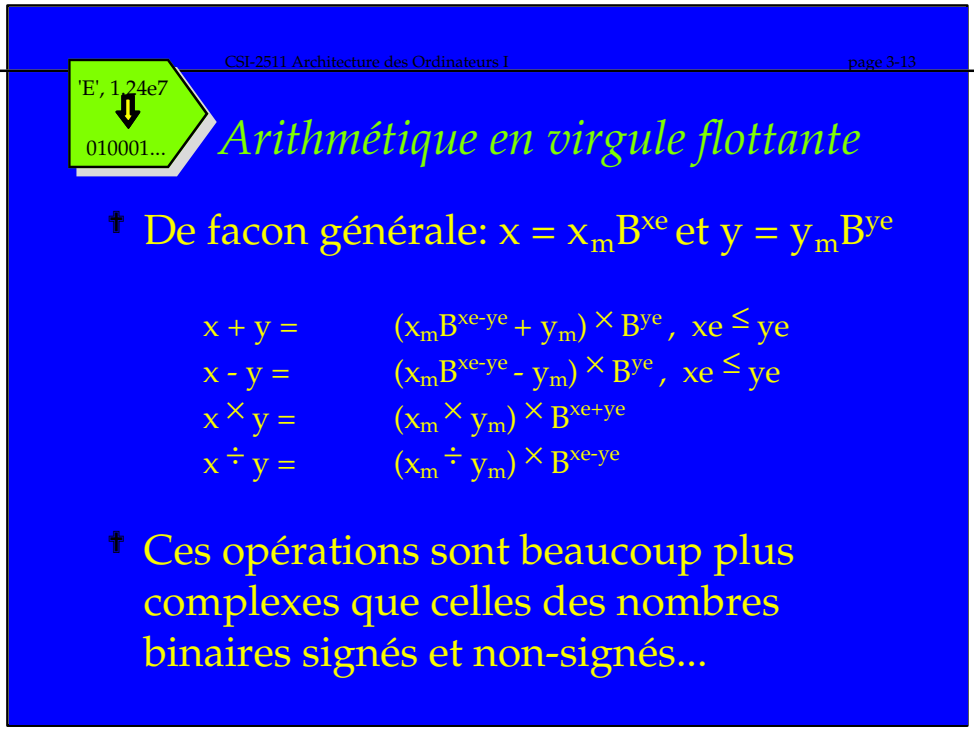
Nombre positif, donc S = 0

$35.5 = 10\ 0011.1 = 1.000111 \times 2^5$

Exposant = C-127 = 5, donc C = 132

1.M = 1.000111, donc M = 0001 1100...

~~0 10000100 0001 1100 0000 0000 0000 000~~



Addition/Soustraction:

La Figure 7-20 présente un algorithme possible pour la réalisation de ces opérations en virgule flottante. On peut remarquer 4 grandes étapes:

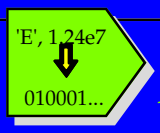
- Vérifications des 0
  - Les 0 doivent avoir un traitement séparé à cause de leur représentation
  - Permet de court-circuiter une addition (ex: $x + 0 = x$)
- Alignement des mantisses
  - Les exposants doivent être égaux pour pouvoir additionner les mantisses.
  - Le nombre avec l'exposant le plus petit voit son exposant être incrémenté et sa mantisse décalée vers la droite.
  - Ex: $1{,}23 \times 10^2 - 4{,}5 \times 10^4 \Rightarrow 0{,}0123 \times 10^4 - 4{,}5 \times 10^4$
- Addition ou soustraction des mantisses
- Normalisation
  - Le résultat doit toujours être normalisé. Par exemple:
  - $123{,}4 \times 10^2 \Rightarrow 1{,}234 \times 10^4$, $0{,}001234 \times 10^2 \Rightarrow 1{,}234 \times 10^{-1}$

Notez bien comment la détection d'*overflow/underflow* s'effectue.

Exercices:

$1{,}25 \times 10^2 - 4{,}5 \times 10^4$ en IEEE 754 = ?

$1{,}25 \times 10^2 + (-1{,}25 \times 10^2)$ en IEEE 754 = ?



*Exemple: Addition **

0 10000100 1000 0000 0000 0000 0000 000

\+ ~~0 10000011 1100 0000 0000 0000 0000 000~~

- Zéro? Non. Signe du résultat: positif, donc S=0
- C = Max(Car1, Car2) = Max(132, 131) = 132 = Car1
- $1.M_1 + (1.M_2 \times 2^{4-5}) = 1.\ 1+0.111 = 10.011$
- Normalisation: $10.011*2^5 = 1.0011*2^6$, donc C = 133 et 1.M = 1.0011

= ~~0 10000101 0011 0000 0000 0000 0000 000~~

(Correspond à: 48 + 28 = 76)

Multiplication

La figure 7-21 présente un algorithme simple pour la multiplication (toutes les considérations à prendre n'y sont pas). Certaines étapes sont présentées:

- Vérification des 0

  Permet encore une fois de court-circuiter la multiplication (ex: y * 0 = 0)

- Addition des exposants (caractéristiques)

  Le *bias* est soustrait par la suite. Par exemple, en IEEE 754, il faut soustraire 127 pour obtenir le bon exposant avec bias:

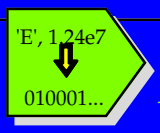
  Car1 + Car2 - 127 = (Exp1 + 127) + (Exp2 + 127) - 127 = Exp1 + Exp2 + 127

- Vérification des dépassement de capacité
- Multiplication des mantisses
- Normalisation
- Arrondissement (selon le nombre de bits dans la mantisse)

Division

Un algorithme est fourni à la figure 7-22. Les opérations effectuées sont très similaires à celles pour la multiplication. Cependant:

- Les 0 sont vérifiés aussi afin de détecter une division par 0
- Les exposants sont soustraits, et le bias additionné
- Les mantisses sont divisées.



*Exemple: Multiplication **

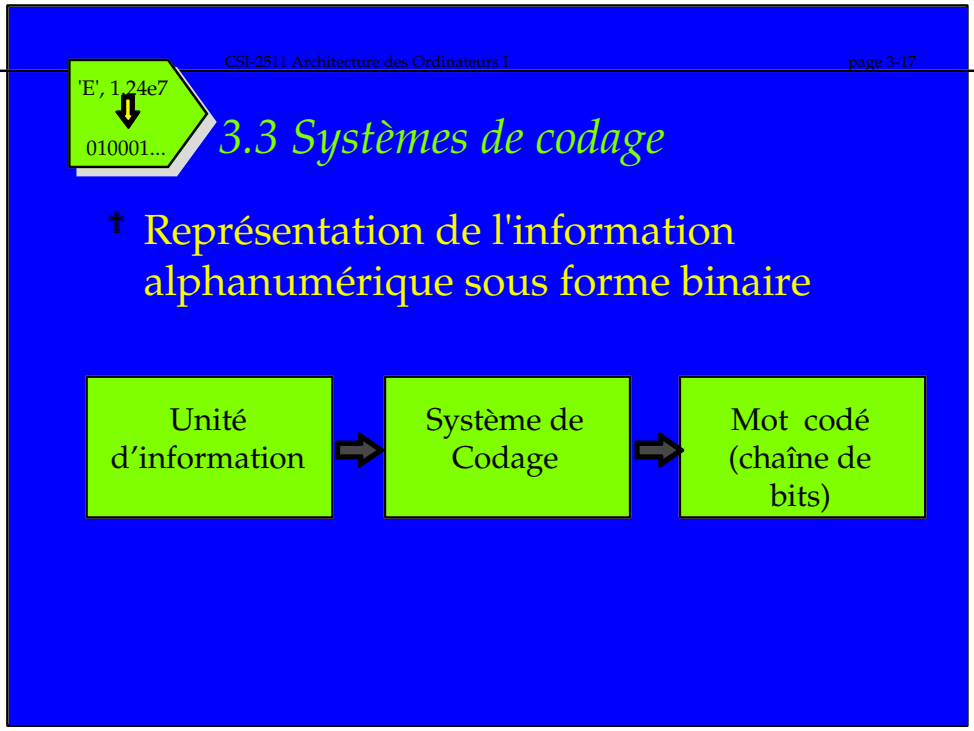
0 10000100 0001 1100 0000 0000 0000 000

x ~~0 10000001 0010 0000 0000 0000 0000 000~~

- Zéro? Non.
- Signe du résultat: positif, donc S=0
- Car1+Car2-127 = 132+129-127 =134 = 10000110 = C
- $1.M_1 * 1.M_2 = 1.000111 \times 1.001 = 1.001111111 = 1.M$

= ~~0 10000110 0011 1111 1000 0000 0000 000~~

(Correspond à: 35.5 * 4.5 = 159.75)



Si nous voulons entre $2^{n-1}+1$ et 2^{n} unité distinctes d'information alphanumérique, n est le nombre minimum de bits nécessaires pour représenter chaque mot codé.

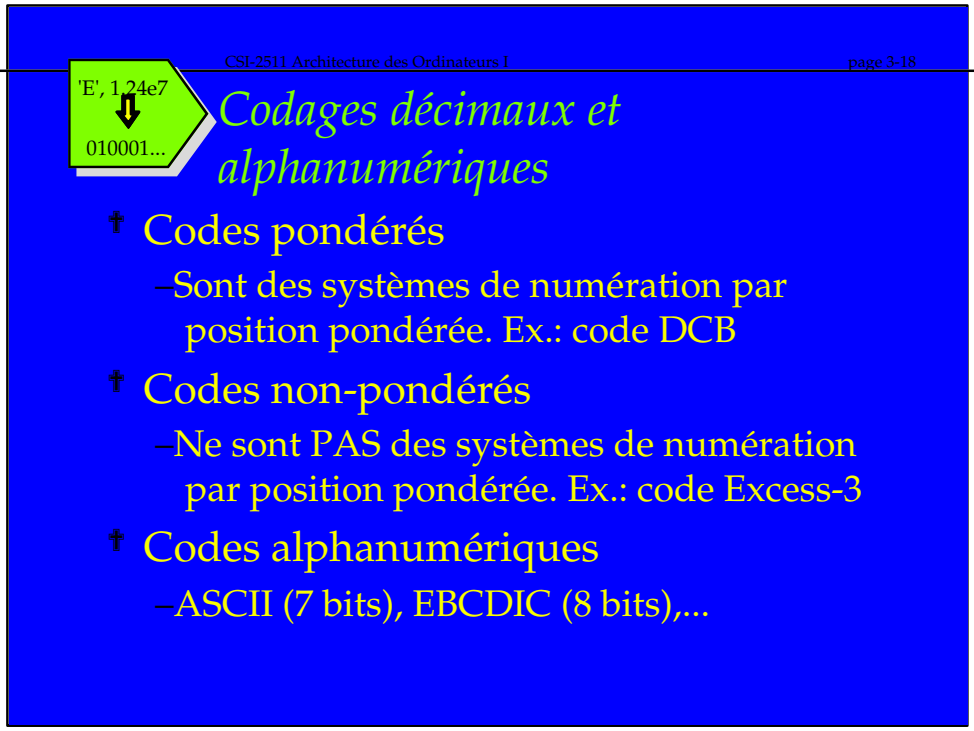
Exemple:

Code *421*: 8 digits nécessaires (0 à 7) $\rightarrow$ $8 = 2^3$ $\rightarrow$ mots de 3 bits

~~Digit~~	~~Mot codé~~
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Combien de bits pour l'alphabet? _____

Combien de bits pour l'alphabet et les chiffres de 0 à 9? _____



Code DCB (Décimal Codé Binaire, ou 8421, ou BCD)

Permet d'éviter la conversion binaire-décimale traditionnelle. Chaque chiffre est représenté par 4 bits et un nombre est codé chiffre par chiffre.

Exemple: $4839_{10} = 0100\ 1000\ 0011\ 1001_{DCB}$

Code Excess-3 (ou XS3)

Le mot codé pour un digit décimal en Excess-3 est le mot codé pour le digit décimal en 8421 + 0011.

Exemple: $4839_{10} = 0111\ 1011\ 0110\ 1100_{XS3}$

~~Décimal~~	~~BCD~~	~~XS3~~
0		
1		
	...	
7		
8		
9		

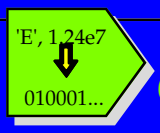
Pour les codes alphanumériques, nous retrouvons les codes:

ASCII (American Standard Code for Information Interchange) sur 7 bits,

EBCDIC (Extended Binary Coded Decimal Interchange Code) sur 8 bits,

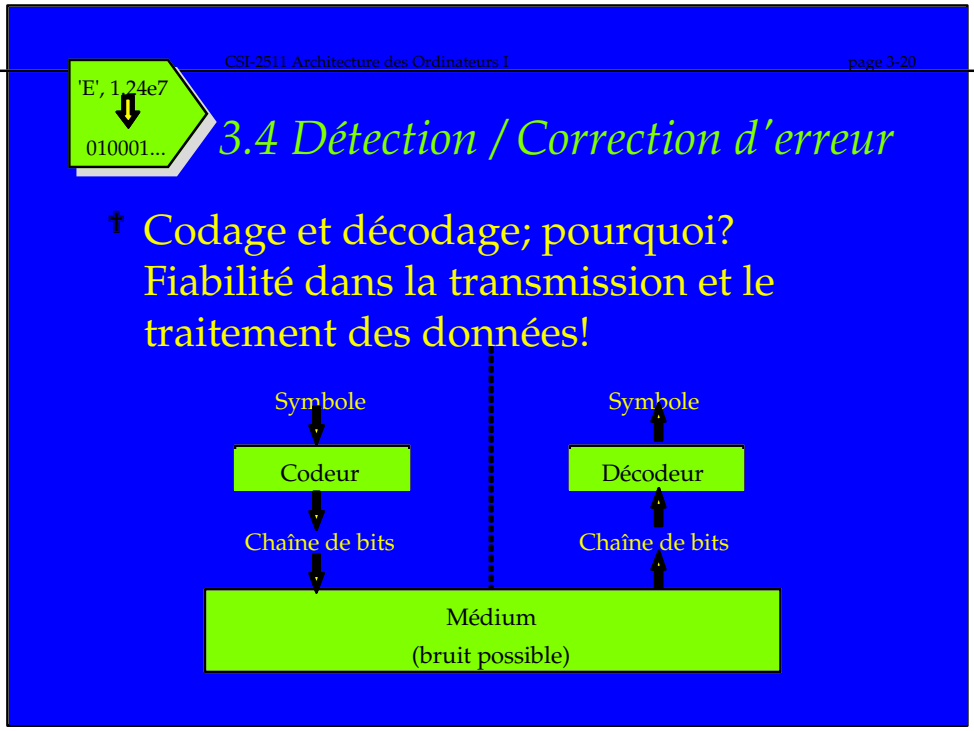
ISO Latin 1 sur 8 bits (sur Windows, sur courrier électronique...),

et de nouveaux code 32 bits permettant de coder tous les caractères de tous les alphabets de la planète (plus de problèmes d'accents, enfin!).



*Codes BCD et Excess-3 **

Décimal	BCD	Excess-3
0	0000	0011
1	0001	0100
2	0010	0101
...		
7	0111	1010
8	1000	1011
9	1001	1100



Code:

Ensemble de mots codés (C_0, C_1, ..., C_n) dans lequel il y a un mot codé pour chaque symbole (S_0, S_1, ..., S_n) de l'alphabet source.

Le nombre de bits nécessaires pour représenter (interpréter) chaque symbole peut dépendre de plusieurs facteurs:

La fréquence moyenne des occurences d'un symbole dans l'information type

Si S_0, S_1, ..., S_n ont peu de chance de survenir avec des probabilités semblables, alors on utilise un code à longueur variable (compression).

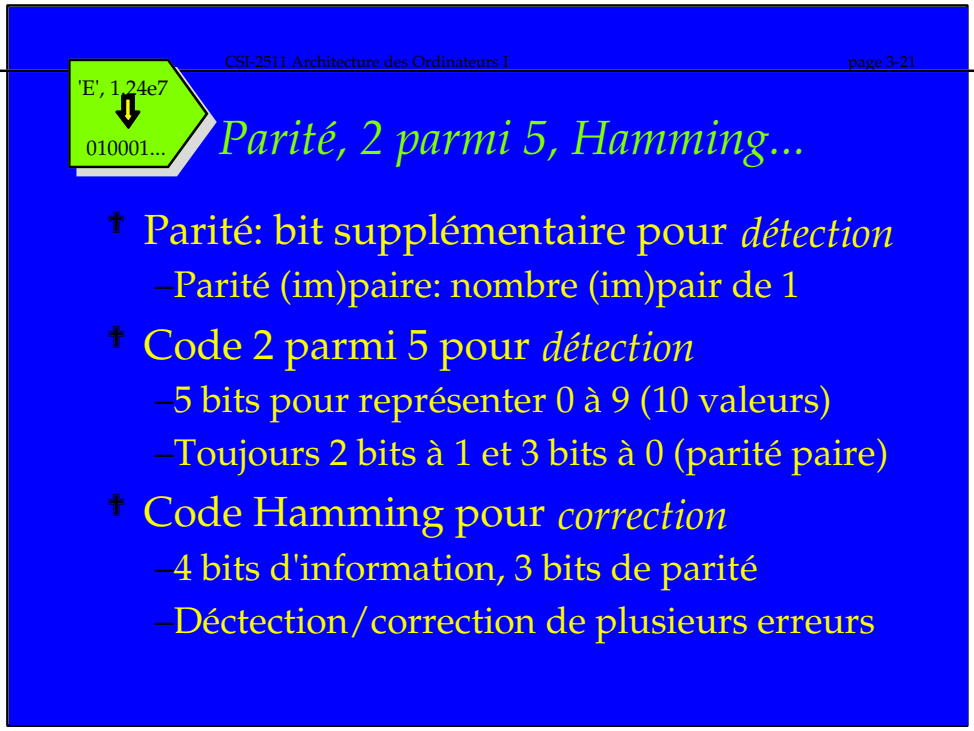
Si S_0, S_1, ..., S_n surviennent avec des probabilités semblables, alors on utilise un code à longueur fixe (ASCII...).

La probabilité d'erreurs simples, doubles ou triples... survenant dans un médium particulier par lequel des mots codés seront transmis ou emmagasinés.

Si les erreurs dans une séquence reçue ou récupérée doivent être

Détectées seulement → **~~code détecteur d'erreurs~~** : Parité, 2 parmi 5...

Corigées → **~~code correcteur d'erreurs~~** : Hamming...



Parité:

- Bit de contrôle supplémentaire ajouté à un mot de *n* bits.
- Parité (im)paire: le nombre total de 1 dans le mot (incluant le bit de parité) doit être (im)pair. Si ce n'est pas le cas, une erreur est détectée.

Code 2 parmi 5:

- Les valeurs de 0 à 9 sont représentées sur 5 bits dont exactement 2 sont des 1
- Code de détection plus fiable mais plus coutêux. La **distance** est de deux (deux combinaisons du code diffèrent par la valeur des bits en au moins deux positions).

Code Hamming:

- Code détecteur/correcteur sur 7 bits (4 d'information (**B**), 3 de parité (**C**)).
- Distance de 3, donc très fiable et puissant, mais aussi redondant et coûteux.

C1	**C2**	**B3**	**C4**	**B5**	**B6**	**B7**		
XX	XX							
	XX					XX		
			XX	XX				

Plusieurs autres types de code (tels les CRC en télécommunication) sont utilisés.