ARC1 - Cours n°4

CODAGE DE L'INFORMATION ET OPERATIONS ARITHMETIQUES

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

A. CODAGE DE L'INFORMATION

1. Nombres entiers

2. Changement de base de représentation

3. Représentation des caractères

4. Représentation des nombres négatifs

5. Représentation des réels

B. OPERATIONS ARITHMETIQUES

1. Addition et soustraction de deux nombres binaires non signés

2. Addition et soustraction de deux binaires entiers en complément à 2

3. Addition et soustraction en DCB

A. CODAGE DE L'INFORMATION

1. Nombres entiers

- Plusieurs façons de représenter les nombres entiers :

✓ **Base 2** (binaire) avec deux symboles (bits) : 0 et 1
✓ **Base8** (octal) avec huit symboles : 0,1,2,3,4,5,6,7
✓ **Base 10** (décimal) avec dix symboles : 0,1,2,3,4,5,6,7,8,9
✓ **Base 16** (héxadécimal) avec seize symboles : 0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F les symboles lettres valant 10, 11, 12, 13, 14 et 15 en décimal.
✓ **Décimal Codé Binaire** (DCB) où chaque chiffre décimal de 0 à 9 est codé en binaire sur 4 bits.

- Dans les quatre premières représentations (pondérées) :
  ✓ Nombre N s'exprime en fonction de la base (B) et des symboles a_i :

$$\boldsymbol{N = \sum_{i=0}^{n-1} a_i B^i}$$

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

- En pratique : symboles juxtaposés (gauche : **poids forts**, droite : **poids faibles**)

- Exemples :

$$N_2 = 10110_2 = 1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 0.2^0 = 16 + 4 + 2 = 22_{10}$$

$$N_8 = 572_8 = 5.8^2 + 7.8^1 + 2.8^0 = 5 * 64 + 7 * 8 + 2 * 1 = 378_{10}$$

$$N_{10} = 782_{10} = 7.10^2 + 8.10^1 + 2.10^0 = 700 + 80 + 2 = 782_{10}$$

$$N_{16} = 2A9F_{16} = 2.16^3 + 10.16^2 + 9.16^1 + 15.16^0$$
$$= 2 * 4096 + 10 * 256 + 9 * 16 + 15 * 1 = 10911_{10}$$

$$N_{DCB} = 1001\ 0010\ 1000_{DCB}$$
$$= \quad 9 \qquad 2 \qquad 8 = 928_{10}$$

Comptage (succession de nombres) dans les différents systèmes de représentation

Décimal	*Binaire*	*Octal*	*Hexadécimal*	*Décimal codé binaire*
00	000 000	00	00	0000 0000
01	000 001	01	01	0000 0001
02	000 010	02	02	0000 0010
03	000 011	03	03	0000 0011
04	000 100	04	04	0000 0100
05	000 101	05	05	0000 0101
06	000 110	06	06	0000 0110
07	000 111	07	07	0000 0111
08	001 000	10	08	0000 1000
09	001 001	11	09	0000 1001
10	001 010	12	0A	0001 0000
11	001 011	13	0B	0001 0001
12	001 100	14	0C	0001 0010
13	001 101	15	0D	0001 0011
14	001 110	16	0E	0001 0100
15	001 111	17	0F	0001 0101
16	010 000	20	10	0001 0110
17	010 001	21	11	0001 0111
18	010 010	22	12	0001 1000
19	010 011	23	13	0001 1001

..etc

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

2. Changement de bases de représentation

- Soit un nombre N, exprimé dans une base B_1, à **convertir** dans une base B_2
- Méthode dépend des bases de départ et d'arrivée

2.1. Binaire ↔ Octal / Hexadécimal

- Méthode **Binaire → Octal** :

  ✓ diviser le nombre en groupes de 3 bits en partant de la virgule (parfois nécessaire d'ajouter un 0 au début ou à la fin d'un groupe de 3 bits)
  ✓ remplacer chaque groupe de 3 bits par un nombre de 0 à 7 :
  000 → 0, 001 → 1, …, 110 → 6, 111 → 7

- Méthode **Octal → Binaire** :

  ✓ remplacer chaque chiffre octal (0……7) par un groupe de 3 bits :
  0 → 000, 1 → 001, …, 7 → 111

- Méthode **Binaire ↔ Hexadécimal** :
  ✓ Même principe avec des groupes de 4 bits.

Exercice d'application

Donner les représentations en base 8, puis en base 16 des deux nombres suivants représentés en base 2.

- 0010101110011010_2
- $1101011101010001{,}101_2$

Correction

- **1er nombre :**
  - ✓ *En octal*

    000 010 101 110 011 010 (2)

    0 2 5 6 3 2 (8)

  - ✓ *En hexadécimal*

    0010 1011 1001 1010 (2)

    2 B 9 A (16)

- **2ème nombre :**
  - ✓ *En octal*

    001 101 011 101 010 001,101 2

    3 5 3 5 2 1 , 5 (8)

  - ✓ *En hexadécimal*

    1101 0111 0101 0001 , 101**0** 2

    D 7 5 1 , A (16)

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

2.2. Binaire ↔ Décimal

- Deux méthodes pour convertir un **nombre décimal entier** vers son équivalent binaire

- **Méthode 1** (pour les petits nombres) :
  - ✓ Exprimer le nombre comme une somme de puissances de 2
  - ✓ Inscrire des 1 et des 0 en face des positions binaires
  - ✓ Exemple : $39_{10}=32+4+2+1 \quad =1x32+0x16+0x8+1x4+1x2+1x1$

    $=1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1_2$

- **Méthode 2** : divisions successives
  - ✓ Convient mieux aux grands nombres
  - ✓ Division du quotient par 2, **jusqu'à ce qu'il soit nul**
  - ✓ Nombre binaire formé par les restes des divisions (le premier reste obtenu en position poids faible, et le dernier reste obtenu en poids fort)

- **Exemple Méthode 2 : (on prend les restes de bas en haut mais on les écrit de gauche à droite.**

$\frac{34}{2} = 17 + reste\ de\ 0$ $\frac{17}{2} = 8 + reste\ de\ 1$ $\frac{8}{2} = 4 + reste\ de\ 0$ $\frac{4}{2} = 2 + reste\ de\ 0$ $\frac{2}{2} = 1 + reste\ de\ 0$ $\frac{1}{2} = 0 + reste\ de\ 1$	**Résultat :** $\mathbf{34_{10}=100010_2}$

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

2.3. Décimal ↔ Octal

- **Octal → Décimal :**
  - ✓ Multiplier chaque chiffre octal par son **poids** positionnel (**puissance de 8**)
  - ✓ Additionner les nombres ainsi obtenus
  - ✓ Exemple : $234_8=2x8^2+1x8^1+4x8^0=2x64+8+4=140_{10}$
  - ✓ Exemple : $13,6_8=1x8^1+3x8^0+6x8^{-1}=8+3+0,75=11,75_{10}$

- **Décimal → Octal :**
  - ✓ Passage par le binaire (divisions par 2 puis conversion binaire→Octal)
  - ✓ Divisions successives par 8

$\frac{150}{8} = 18 + \boldsymbol{reste\ de\ 6}$ $\frac{18}{8} = 2 + \boldsymbol{reste\ de\ 2}$ $\frac{2}{8} = 0 + \boldsymbol{reste\ de\ 2}$	**Résultat :** $\mathbf{150_{10}=226_8}$

2.4. Décimal ↔ Hexadécimal

- **Hexadécimal → Décimal :**
  - ✓ Multiplier chaque chiffre hexadécimal par son **poids** positionnel (**puissance de 16**)
  - ✓ Additionner les nombres ainsi obtenus
  - ✓ Exemple : $A3C_{16}=10x16^{2}+3x16^{1}+12x16^{0}=2560+48+12=2620_{10}$
  - ✓ Exemple : $2B,8_{16}=2x16^{1}+11x16^{0}+8x16^{-1}=32+11+0,5=43,5_{10}$

- **Décimal →Hexadécimal:**
  - ✓ Passage par le binaire (divisions par 2 puis conversion binaire→Hexa)
  - ✓ Divisions successives par 16

$$\frac{287}{16} = 17 + \boldsymbol{reste\ de}\ 15$$ $$\frac{17}{16} = 1 + \boldsymbol{reste\ de}\ 1$$ $$\frac{1}{16} = 0 + \boldsymbol{reste\ de}\ 1$$	**Résultat :** **$287_{10}=11F_{16}$**

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

2.5. Décimal ↔ Décimal Codé Binaire (DCB)

- Le DCB est un code de représentation des nombres qui combine certaines caractéristiques du système binaire et du système décimal (ex : ordinateurs gestion plus entrées/sorties qu'opérations)

- **Décimal → DCB :**

✓ Remplacer chaque chiffre du nombre décimal par son équivalent binaire

✓ Le chiffre le plus élevé est 9, donc il on utilise 4 bits pour chaque chiffre

✓ Exemple :

$$\begin{array}{ccc} 6 & 7 & 2_{10} \\ \downarrow & \downarrow & \downarrow \\ 0110 & 0111 & 0010_{DCB} \end{array}$$

✓ Seuls les 10 groupes 0000 à 1001 existent (sont valides) en DCB

✓ Les autres combinaisons (1010…1111) sont invalides.

✓ Comparaison entre code DCB et nombre binaire :

$176_{10}=10110000_2$ (Binaire)

$176_{10}=0001\ 0111\ 0110_{DCB}$ (DCB)

- **DCB → Décimal :** remplacer chaque groupe de 4 bits par son équiv. Décimal.

3. Représentation des caractères

- Un ordinateur reconnait et manipule des codes qui correspondent à des nombres, des lettres, des signes de ponctuation et des caractères spéciaux

- Codes « alphanumériques » dont le principal est le code ASCII (7bits)

- Permettent de représenter :

  - ✓ Les 26 lettres minuscules
  - ✓ Les 26 lettres majuscules
  - ✓ Les dix chiffres
  - ✓ Les 7 signes de ponctuation
  - ✓ Les caractères spéciaux comme +, /, &, %....

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

- Exemples de représentation en ASCII

caractère	code ASCII	code hexadécimal
(	40	28
)	41	29
*	42	2A
+	43	2B
,	44	2C
-	45	2D
.	46	2E
/	47	2F
0	48	30
1	49	31
2	50	32
…………	…..	……
9	57	39
:	58	3A

>	62	3E
?	63	3F
@	64	40
A	65	41
B	66	42
C	67	43
D	68	44
…………………		
Z	90	5A
[	91	5B
\	92	5C
]	93	5D
^	94	5E
_	95	5F

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

;	59	3B
<	60	3C
=	61	3D

`	96	60
a	97	61
b	98	62
c	99	63
d	100	64
……		
y	121	79
z	122	7A

4. Représentation des nombres négatifs

Deux principales méthodes de représentation des nombres négatifs :

4.1. Signe / Amplitude

- C'est la notation que nous utilisons **implicitement** quand nous manipulons un entier signé, comme par exemple -5 ou +10.

- Une position est réservée au **signe**, les autres étant destinées à exprimer la valeur absolue (ou module, ou **amplitude**).

- La convention adoptée pour un nombre de n bits est la suivante :

  ✓ Le bit le plus à gauche (rang n-1) représente le **signe** :

  0 : signe + (valeur positive)
  1 : signe – (valeur négative)

  ✓ Les autres bits (rang 0 à n-2) représentent la **valeur absolue** du nombre

- Exemples : soit un nombre A utilisant le format n=4 bits

A	Signe/Module
+0	0000
+1	0001
+2	0010
….	….
+7	0111
-0	1000
-1	1001
…	…
-6	1110
-7	1111

Deux représentations du 0 (+0 et -0).

- Nécessité d'indiquer le **format** lorsque l'on code un nombre négatif (ce qui est le cas également pour les autres méthodes de représentation des nombres relatifs comme le complément à 2).

- Nombre 111 codé en Signe/Amplitude. Est-il positif ou négatif ?

  - ✓ A priori impossible à dire si l'on ne sait pas sur combien de bits il est représenté (format).
  - ✓ Si le format est n=3 bits, le nombre est négatif (bit le plus à gauche =1).
  - ✓ Si le format est n>3, 4 bits par exemple, le nombre est positif car sa représentation sur 4 bits est 0111 (on ajoute un 0 devant pour être dans le bon format) et alors le bit le plus à gauche est 0.

- Exemple 1 : **Interprétation** d'un nombre ($N=0101101_{SA}$) représenté en Signe/Amplitude sur 7 bits

  0 : bit de signe 101101 : grandeur exacte

  D'où la valeur du nombre représenté : $N=+45_{10}$

- Exemple 2 : **Représentation** d'un nombre ($N=-14_{10}$) en Signe/Amplitude avec un format de travail n=8 bits.

  Bit de signe : 1 (nombre négatif) Valeur absolue sur 7 bits : 000 1010

  Résultat : $N = 1000\ 1010_{SA}$

- **Addition** avec format Signe/Amplitude:
    - ✓ Connaître le signe de chaque opérande
    - ✓ Réaliser, selon signes (identiques ou non) addition ou soustraction des valeurs absolues
    - ✓ Traiter les signes à part

- **Avantages/inconvénients** de la représentation :

✓ Avantages :

- Nombre directement **compréhensible** par l'utilisateur
- Calcul direct de l'**opposé** arithmétique en changeant le signe.

✓ Inconvénients :

- **Deux représentations pour le 0**
  ⇒Peut poser des problèmes, par exemple lors de calculs, car les deux cas sont à prendre en compte.
- Opérations arithmétiques complexes (addition et soustraction en deux parties, calcul de la valeur, puis traitement du signe).

⇒ Représentation peu utilisée

4.2. Complément à 2

- **Complément à 1** d'un nombre

Soit A un nombre de n bits :

$$A=a_{n-1}\ a_{n-2}\dots a_1a_0$$

Le complément à 1 d'un nombre s'obtient en inversant tous les bits du nombre

- **Complément à 2** d'un nombre

Obtenu en ajoutant 1 (addition binaire) au complément à 1 :

$$Cpt2(A)=Cpt1(A)+1$$

Rappel des règles d'addition en binaire :

	0	*0*	*1*	*1*
	+0	*+1*	*+0*	*+1*
Somme	*0*	*1*	*1*	*0*
Retenue	*0*	*0*	*0*	*1*

En complément à 2, la retenue engendrée par l'addition des bits les plus à gauche est rejetée.

- **Représentation** des nombres en complément à 2

✓ **Nombre positif**

Pour obtenir la représentation en Complément à 2 d'un nombre positif sur n bits :

- Représenter le nombre en binaire sur n-1 bits
- Ajouter un 0 devant (bit de signe)
- Exemple : Représentation de +6 en Cpt2, sur 4 bits :

$+6_{10}=110_2$

Sur 4 bits : $+6_{10}=0110_{Cpt2}$

✓ **Nombre négatif**

Pour obtenir la représentation en Complément à 2 d'un nombre négatif sur n bits :

- Prendre le complément à 2 du nombre positif équivalent, bit de signe compris. On obtient alors un nombre dont le bit de signe est 1.
- Exemple : Représentation de -5_{10} en Cpt2, sur 4 bits ?
- $+5_{10}=101_2$
- Sur 4 bits : $+5_{10}=0101$
- Complément à 1 : 1010_{Cpt1}
- Complément à 2 : 1011_{Cpt2}
- Finalement $-5_{10}=1010_{Cpt2}$

On obtient bien un bit de signe = 1

- **Interprétation** d'un nombre représenté en complément à 2

✓ **Nombre positif** (bit de signe = 0) :

- Pour obtenir l'équivalent décimal on fait la somme des bits (multipliés par leur poids)
- Ex : $N=01100_{Cpt2}$ est représenté en Cpt2 sur 5 bits :

Bit de signe=0 $\Rightarrow$ nombre positif

$\Rightarrow$ quatre derniers chiffres représentent en binaire la grandeur exacte : $N=+12_{10}$.

✓ **Nombre négatif** (bit de signe = 1) :

- Prendre le Cpt2 pour obtenir le nombre positif correspondant (c'est-à-dire la valeur absolue)
- Calculer l'équivalent décimal
- Ajouter un signe moins.
- Ex : 101001_{Cpt2} est un nombre représenté en Cpt2 sur 6 bits.

Bit de signe=1 $\Rightarrow$ nombre négatif

$\Rightarrow$ calculer le complément à 2 du nombre (avec ou sans bit de signe) pour connaître la valeur absolue du nombre

Cpt1(101001)=010110

$Cpt2(101001)=010110+1=010111_2$

$N=-(010111)_2=-23_{10}$

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

5. Représentation des réels

- Langages de programmation doivent pouvoir manipuler, en plus des entiers signés et non signés, des **nombres réels**
- Exemples :

  $3.14159265\ldots_{10}$ (π)

  $2.71828\ldots_{10}$ (e) *(Nombre d'Euler : ln(e)=1 ; exp(1)=e)*

  0.000000001_{10} ou $1.0_{10}\text{x}10^{-9}$ (nb de secondes dans une nanosecondes)

  $3.155\ 760\ 000_{10}$ ou $3.15576_{10}\text{x}10^{9}$ (secondes dans un siècle)

- Le dernier nombre est plus grand que ce qui peut être représenté avec un entier signé de 32 bits
- Autre notation plus pratique (pour les deux derniers nombres) :
  - ✓ « ***notation scientifique*** », avec un seul chiffre à gauche de la virgule
- Quand il n'y a qu'un seul chiffre (différent de 0) avant la virgule, le nombre est dit « **normalisé** », exemple :
  - ✓ $1{,}5_{10}\text{x}10^{-8}$ est en ***notation scientifique normalisée***
  - ✓ $15{,}0_{10}\text{x}10^{-9}$ non $0{,}15_{10}\text{x}10^{-7}$ non plus

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

- Nombres binaires en notation scientifique normalisée, exemple :
  - ✓ $1{,}0_2 x 2^{-1}$ (format virgule flottante)
  - ✓ En langage C : ces nombres sont appelés des « flottants » (**float**)
  - ✓ Format général des nombres binaires représentés en virgule flottante :

$$\mathbf{1.a_2 \; x \; 2^b}$$

- Avantages :
  - ✓ Simplifie les échanges de données
  - ✓ Simplifie les opérations arithmétiques
  - ✓ **Augmente la précision** d'un nombre stocké dans un mot car les 0 inutiles du début sont remplacés par des chiffres à droite de la virgule (stockés dans la mantisse)
  - ✓ **Augmente aussi l'intervalle des valeurs possibles** (un mot de k bits peut représenter des nombres bien plus grands que 2k)

- Représentation :
  - ✓ Nécessité de trouver un **compromis** entre la place prise par les **chiffres significatifs (mantisse)** et la **taille de l'exposant** pour représenter au mieux sur un nombre de bits fixe
  - ✓ Compromis entre **précision** et **plage** des nombres qui peuvent être représentés :
    - Augmenter la taille (nb de bits) de la mantisse permet d'**augmenter sa précision**
    - Augmenter la taille (nb de bits) de l'exposant permet d'**augmenter la plage des nombres représentables**

31	30 29…………23	22……………………………………………..0
S	E=Exposant	M=Mantisse
1bit	8 bits	23 bits

Représentation IEEE 32 bits

- Interprétation du nombre représenté par S, E, M en format virgule flottante :

$$\mathbf{[S,E,M]_{flottant}} = (-\mathbf{1})^{\boldsymbol{S}}.\boldsymbol{m}.\mathbf{2}^{\boldsymbol{e}}$$

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

m est *à l'origine* de la valeur stockée dans la partie **Mantisse** du tableau de 32 bits :

- $m=1,M_{22}M_{21}\ldots M_0$ (1 implicite, non stocké dans les bits deM)

e est *à l'origine* de la valeur stockée dans la partie **Exposant** du tableau de 32 bits

- e appartient à l'intervalle [-127,+128]
- **e** = [**E**]$_2$-127

Plage des nombres représentables avec cette notation, très grande :

Le plus petit : $2_{10}x10^{-38}$……………………………………Le plus grand : $2_{10}x10^{38}$

- Overflow / underflow

  - ✓ Overflow : si le **nombre est tellement grand** que l'exposant du nombre est trop grand pour être représenté par les 8 bits réservés à l'exposant
  - ✓ Underflow : si le **nombre est tellement petit** que l'exposant négatif est trop grand pour être représenté par les 8 bits réservés à l'exposant

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

- ✓ Comment empêcher l'overflow et l'underflow ? En **augmentant le nombre de bits réservés à l'exposant** (ex : 11 bits exposant / 20 bits mantisse)
- ✓ En C : le **type double**, et les opérations « double précision » contrairement aux précédentes (simple précision)
- ✓ Plage des nombres dans le type double : de $2.0_{10}x10^{-308}$ à $2.0_{10}x10^{308}$

- Pour pouvoir stocker encore plus de bits significatifs, le format IEEE rend **implicite le 1 placé devant la virgule (on ne le stocke pas, on sait qu'il est là)**

  $\Rightarrow$ 24 bits significatifs (le 1 et les 23 bits) en simple précision
- Forme générale :

$$\boldsymbol{(-1)^{S}.(1+Mantisse).2^{e}}$$

La valeur du nombre stocké est donc :

$$(-1)^{S}.[1+(M1.2^{-1})+(M2.2^{-2})+(M3.2^{-3})\ldots].2^{e}$$

- **Exposant codé / vrai exposant** :
  - ✓ L'exposant codé (E) n'est pas le vrai exposant (e)
  - ✓ L'exposant codé est le vrai exposant auquel on a jouté un biais

- ✓ Pourquoi ? Pour qu'une fois codé, les exposants négatifs paraissent (dans un tri simple) plus petits que les exposants positifs (ce qui n'est pas le cas si on représente tel quel, en Cpt à 2 par exemple l'exposant)
- ✓ Dans la norme IEEE 32 bits, le **biais est égal à 127**

- ✓ Ainsi la valeur du nombre codé avec <u>Exposant</u> sur 7 bits et <u>mantisse</u> sur 23 bits est en fait :

$$(-\mathbf{1})^{\boldsymbol{S}}.\,(\mathbf{1}+\boldsymbol{Mantisse}).\,\mathbf{2}^{\boldsymbol{Exposant}-\boldsymbol{Biais}}$$

(Exposant stocké=vrai exposant+127)

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

Exercice d'application n°1

1. Montrez pourquoi les exposants négatifs non biaisés paraissent plus grands (dans une opération de comparaison des exposants – en regardant le premier bit par exemple - dans un but de tri des nombres) que les exposants positifs, lorsque les exposants sont représentés en complément à 2.

Pour cela, prenez l'exemple des deux nombres suivants :
0.5_{10} et 2_{10}

2. Reprenez le même exemple, mais en ajoutant le biais aux exposants et vérifiez qu'une fois le biais ajouté, un exposant négatif stocké parait effectivement plus petit qu'un exposant positif.

Correction

1. Montrez pourquoi les exposants négatifs non biaisés paraissent plus grands (dans une opération de comparaison des exposants – en regardant le premier bit par exemple - dans un but de tri des nombres) que les exposants positifs, lorsque les exposants sont représentés en complément à 2.

- On met tout d'abord les nombres sous forme normalisée :

$$\mathbf{0.5_{10} = 1.0_{2}x2^{-1} \text{ et } 2_{10}=1.0_{2}x2^{+1}}$$

L'exposant du premier est donc -1 et l'exposant du second est donc 1.

La mantisse est la même pour les deux nombres : 0000…..00000 (le 1 devant la vigule n'est pas stocké)

- (-1) représenté en complément à 2 sur 8 bits :

On représente 1 sur 8 bit : 0000 00012

On complémente à 1 : 1111 1110Cpt1

Puis on ajoute 1 : 1111 1111Cpt2

- D'où la représentation de 0.5 en format Flottant normalisé IEEE 32 bits (sans le biais) :

31	30 29 28 27 26 25 24 23	22 21 ……………………………………………………………………………0
0	1 1 1 1 1 1 1 1	0 0……………………………………………………………………………..0

Correction (suite)

- (1) représenté en complément à 2 sur 8 bits : 0000 0001

- D'où la représentation de 2 en format Flottant normalisé IEEE 32 bits (sans le biais) :

31	30 29 28 27 26 25 24 23	22 21 ……………………………………………………………………………0
0	0 0 0 0 0 0 0 1	0 0……………………………………………………………………………..0

2. Reprenez le même exemple, mais en ajoutant le biais aux exposants et vérifiez qu'une fois le biais ajouté, un exposant négatif stocké parait effectivement plus petit qu'un exposant positif.

Si on ajoute un biais de 127 :

(-1) devient -1+127=126

(1) devient 1+127=128

126 représenté en Complément à 2 : 0111 1110

128 représenté en Complément à 2 : 1000 0000

Si l'on veut comparer efficacement les deux nombres en comparant leurs exposants, et en particulier le 1er bit de l'exposant stocké, alors on trouve bien que le nombre 2_{10} est plus grand que le nombre 0.5_{10}.

Exercice d'application n°2

1) Donner la représentation IEEE 32 bits du nombre -0.75_{10} en simple précision

Correction

- Représenter -0,75 en binaire :

$$-0,75_{10} = -(0,5+0,25)_{10} = -(1.2^{-1}+1.2^{-2})10 = -0.11_2$$

- Notation scientifique :

$$-0.11_2 \text{ x } 2^0$$

- Normaliser (décaler les bits de façon à ne plus avoir 0 devant la virgule) :

$$-1.1_2 \text{ x } 2^{-1}$$

- Représentation générale :

$$(-1)^S.(1 + \text{mantisse}).2^{\text{Exposant}-127}$$

D'où :

mantisse=$0.1000000\ldots_2$ et

ExposantStocké=vrai exposant+127=-1+127=126=01111110_2

- D'où la représentation de 2 en format Flottant normalisé IEEE 32 bits (sans le biais) :

31	30 29 28 27 26 25 24 23	22 21 …………………………………………………………………………0
1	0 1 1 1 1 1 1 0	1 0…………………………………………………………………………..0

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

Exercice d'application n°3

Quel est le nombre décimal représenté par le mot binaire suivant :

31	30 29 28 27 26 25 24 23	22 21 ………………………………………………………………………… 0
1	1 0 0 0 0 0 0 1	0 1 0 0 ………………………………………………………………………..0

Réponse :

- Bit de signe = 1, donc nombre négatif
- ExposantCodé=10000001_2=129_{10} donc exposant = 129-127_{10}=2_{10}
- $mantisse_2=0.01_2=1*2^{-2}=0.25_{10}$

Donc le nombre vaut :

$N=(-1)^S . (1+mantisse) . 2^{exposant}=-1.25 \times 2^2= -5.0_{10}$

B. OPERATIONS ARITHMETIQUES

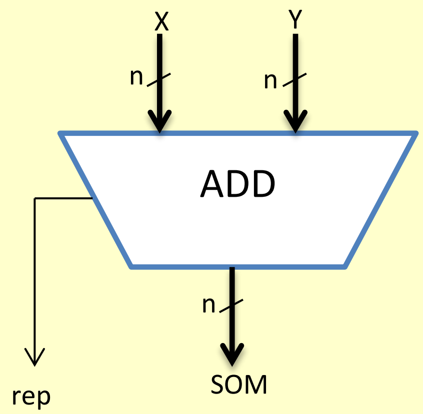
1. Addition et soustraction de deux nombres binaires <u>non signés</u>

1.1. Principe de l'addition

- Soit deux nombres A et B représentés sur quatre bits
- Somme de A et B : addition bit par bit, de droite vers la gauche en propageant la retenue au rang immédiatement supérieur.

Nombre A	a_4	a_3	a_2	a_1
Nombre B	b_4	b_3	b_2	b_1
Somme A+B	S_4	S_3	S_2	S_1
Retenues	$\leftarrow r_4$	r_3	r_2	r_1

- Opérateur ADD sur mots de n bits :



- Fonctionnement défini par un algorithme : algorithme d'addition binaire

x_i y_i r_i	r_{i+1}	s_i
0 0 0	0	0
0 0 1	0	1
0 1 0	0	1
1 0 0	0	1
0 1 1	1	0
1 1 0	1	0
1 0 1	1	0
1 1 1	1	1

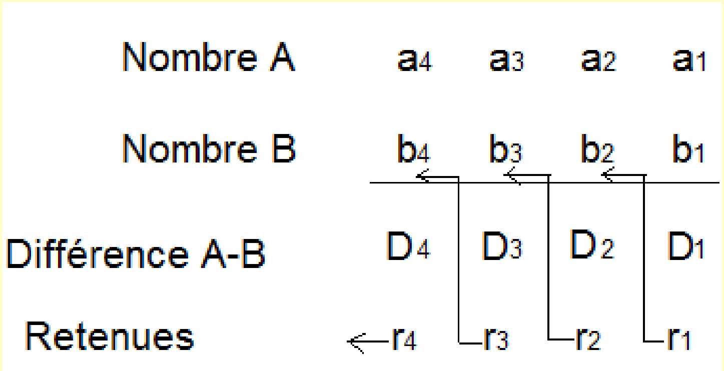
Appliqué itérativement avec $r_0=0$
SOM=s_{n-1} $s_{n-2}\ldots s_0$
rep=r_n ("report ou retenue")

Exemple :
X 1011
Y 0011
0 1110
rep SOM

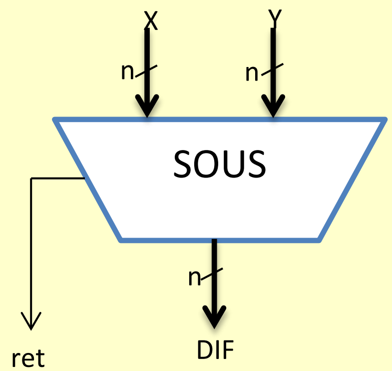
ECOLE SUPERIEURE D'INGENIEURS DE RENNES

1.2. Principe de la soustraction

- Soit deux nombres A et B représentés sur quatre bits
- Différence A moins B : la différence a_1 moins b_1 donne un résultat D_1 et une retenue r_1. La somme b_2+r_1 est soustraite de a_2 pour donner D_2 et r_2 et ainsi de suite.



- **Opérateur SOUS sur mots de n bits**



- Fonctionnement défini par un algorithme : algorithme de soustraction binaire

x_i y_i r_i	r_{i+1} s_i
0 0 0	0 0
0 0 1	1 1
0 1 0	1 1
1 0 0	1 0
0 1 1	0 1
1 1 0	0 0
1 0 1	0 0
1 1 1	1 1

Appliqué itérativement avec $r_0=0$
DIF=d_{n-1} d_{n-2}….d_0
ret=rn

Exemple :

$$\begin{array}{lrr} X & & 1101 \\ Y & & \underline{0011} \\ & 0 & 1010 \\ & \text{ret} & \text{DIF} \end{array}$$

- Explication de l'algorithme de soustraction binaire : si le facteur soustrait est supérieur au facteur soumis à la soustraction, il faut emprunter la valeur de la base à la colonne suivante pour que la soustraction soit possible (vrai ∀ la base).

- Exemples en base 10, 2, 8, 16 :

Base 10	Base 2	Base 8	Base 16
$9\ 5\ 3_{10}$ $-_{1}3_{1}7\ \ 5_{10}$ $5\ 7\ 8_{10}$	$1\ \ 0\ \ 1\ \ 0_{2}$ $-_{1}0\ _{1}1\ _{1}1\ \ 1_{2}$ $0\ \ 0\ \ 1\ \ 1_{2}$	$5\ 3\ \ 7\ 2_{8}$ $-_{1}2\ \ 5\ _{1}4\ \ 5_{8}$ $2\ 6\ 2\ 5_{8}$	$A\ \ 6\ \ 9\ C_{16}$ $-_{1}2\ _{1}B\ \ C\ 8_{16}$ $7\ \ A\ D\ 4_{16}$

Exercice d'application

Réaliser les soustractions suivantes :

$101100_{2}-10011_{2}$	$11000101_{2}-10110110_{2}$	$43765_{8}-37472_{8}$	$AE63B_{16}-8F9A2_{16}$

Correction

11001_2	1111_2	4273_8	$1DC99_{16}$

1.3. Problème d'overflow (« dépassement », « débordement »)

- Overflow : quand le résultat d'une opération ne peut pas être représenté sous forme qui peut être stockée dans les registres disponibles (**nombre de bits nécessaires supérieur à la taille du registre**)

- Dans un registre de k positions, pour une base b, seuls les nombres entiers positifs N tels que : $0 \leq N \leq b^k - 1$ peuvent être représentés. $b^k - 1$ est la capacité du registre.
  Ex : en base 2, un registre à 4 positions permet de représenter les nombres jusqu'à 2^4-1=15 (N_{max}=1111)

- **Quand** peut-il y avoir overflow ?
  - ✓ Pas dans l'addition de deux nombres de signes différents car la somme est toujours plus petite que l'un des opérandes (Ex : -10+4=-6)
  - ✓ Pas dans la soustraction de deux nombres de même signes, pour la même raison

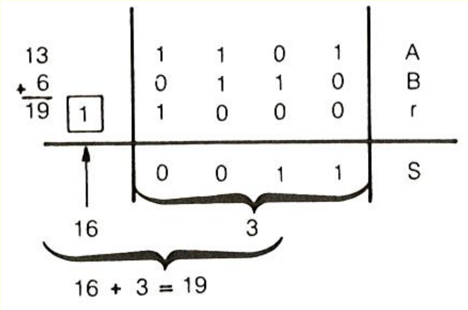
- Détection de l’overflow : examen du report sortant dans différents cas pour mettre en évidence les cas problématiques :
  - ✓ Cas1 : on **additionne** deux nombres **sans dépasser 2^k-1**, le résultat obtenu est correct, et la retenue de sortie vaut 0

	0	0	1	1	A	3
	0	0	0	1	B	+ 1
$r_n \rightarrow \boxed{0}$	0	1	1	0	r	4
	0	1	0	0	S	

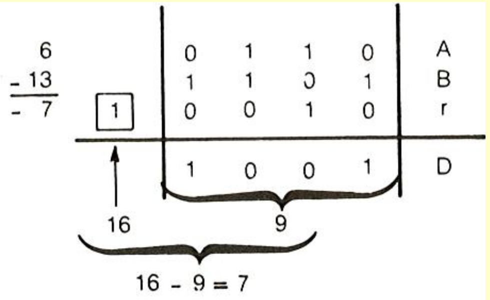
  - ✓ Cas2 : on **soustrait** un nombre d’un nombre plus grand le résultat obtenu est correct, et la retenue de sortie vaut 0

	1	0	1	0	A	10
	0	0	1	1	B	- 3
$r_n \rightarrow \boxed{0}$	1	1	1	0	r	7
	0	1	1	1	S	

✓ Cas3 : on **additionne** deux nombres tels que **le résultat dépasse 2^k-1**, le résultat obtenu sur 4 bits est incorrect, il ne représente que la partie « poids faible » de la somme, et la retenue de sortie vaut 1.



✓ Cas4 : on **soustrait** un nombre d'un autre nombre plus petit, le résultat obtenu sur 4 bits est incorrect, et le report sortant vaut 1.



ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

Conclusion :

- **En binaire, pour les additions de nombres non signés** (on verra que ce n'est pas le cas pour les additions de nombre signés en Cpt2), **la retenue de sortie suffit à détecter l'overflow.**

- Si après addition ou soustraction de deux nombres de k bits :
  - ✓ **retenue de sortie = 0, résultat correct**
  - ✓ **retenue de sortie = 1, overflow** (dépassement) et résultat incorrect sur k bits