Introduction aux systèmes informatiques

Codage et Systèmes d'exploitation

Pôle ASR – Module M1101 – Semestre 1

Bruno BEAUFILS

⟨bruno.beaufils@univ-lille.fr⟩

https://beaufils.u-lille.fr

Université de Lille, IUT « A », Département informatique

Année 2020/2021

http://m1101.iutinfo.fr

Codage

Coder des nombres

- Besoins en informatique concernant les nombres
  - **représenter** ..... stockage en binaire
  - pour **manipuler** ..... arithmétique ($+$, $-$, $*$, $/$)
  - en **automatisant** au maximum
- Contraintes en informatique : **espace limité (fini)**
  - largeur mémoire fixe ..... **taille de chacune des cases**
  - longueur mémoire fixe ..... nombre de cases disponibles
- Choix
  - **numération de position**
    - mode de représentation des nombres
    - suite de symboles (*chiffres*)
    - importance de la position précise dans la suite
    - ensemble de chiffres fixé
  - arithmétique **adaptée** à chaque cas
    - calcul doivent être facile à faire avec des opérations simples (portes logiques)

Type de nombres étudiés

- entier naturel ($\approx \mathbb{N}$) ..... codage binaire **non signé**
- entier relatif ($\approx \mathbb{Z}$) ..... codage binaire **signé**
- réel ($\approx \mathbb{R}$) ..... codage binaire **flottant**

Coder des nombres

- Besoins en informatique concernant les nombres
  - **représenter** ..... stockage en binaire
  - pour **manipuler** ..... arithmétique ($+$, $-$, $*$, $/$)
  - en **automatisant** au maximum
- Contraintes en informatique : **espace limité (fini)**
  - largeur mémoire fixe ..... **taille de chacune des cases**
  - longueur mémoire fixe ..... nombre de cases disponibles
- Choix
  - **numération de position**
    - mode de représentation des nombres
    - suite de symboles (*chiffres*)
    - importance de la position précise dans la suite
    - ensemble de chiffres fixé
  - arithmétique **adaptée** à chaque cas
    - calcul doivent être facile à faire avec des opérations simples (portes logiques)

Type de nombres étudiés

- entier naturel ($\approx \mathbb{N}$) ..... codage binaire **non signé**
- entier relatif ($\approx \mathbb{Z}$) ..... codage binaire **signé**
- réel ($\approx \mathbb{R}$) ..... codage binaire **flottant**

Coder des nombres

- Besoins en informatique concernant les nombres
  - **représenter** ..... stockage en binaire
  - pour **manipuler** ..... arithmétique ($+$, $-$, $*$, $/$)
  - en **automatisant** au maximum
- Contraintes en informatique : **espace limité (fini)**
  - largeur mémoire fixe ..... **taille de chacune des cases**
  - longueur mémoire fixe ..... nombre de cases disponibles
- Choix
  - **numération de position**
    - mode de représentation des nombres
    - suite de symboles (*chiffres*)
    - importance de la position précise dans la suite
    - ensemble de chiffres fixé
  - arithmétique **adaptée** à chaque cas
    - calcul doivent être facile à faire avec des opérations simples (portes logiques)

Type de nombres étudiés

- entier naturel ($\approx \mathbb{N}$) ..... codage binaire **non signé**
- entier relatif ($\approx \mathbb{Z}$) ..... codage binaire **signé**
- réel ($\approx \mathbb{R}$) ..... codage binaire **flottant**

Coder des nombres

- Besoins en informatique concernant les nombres
  - **représenter** ..... stockage en binaire
  - pour **manipuler** ..... arithmétique ($+$, $-$, $*$, $/$)
  - en **automatisant** au maximum
- Contraintes en informatique : **espace limité (fini)**
  - largeur mémoire fixe ..... **taille de chacune des cases**
  - longueur mémoire fixe ..... nombre de cases disponibles
- Choix
  - **numération de position**
    - mode de représentation des nombres
    - suite de symboles (*chiffres*)
    - importance de la position précise dans la suite
    - ensemble de chiffres fixé
  - arithmétique **adaptée** à chaque cas
    - calcul doivent être facile à faire avec des opérations simples (portes logiques)

Type de nombres étudiés

- entier naturel ($\approx \mathbb{N}$) ..... codage binaire **non signé**
- entier relatif ($\approx \mathbb{Z}$) ..... codage binaire **signé**
- réel ($\approx \mathbb{R}$) ..... codage binaire **flottant**

unsigned-*k* : entier non signé (codage)

codage naturel — unsigned-*k*

- on code un entier positif par sa représentation en base 2
- le nombre à coder doit être dans l'espace codable
- le code utilise toute la largeur fixée

unsigned-k : entier non signé (codage)

codage naturel — **unsigned-k**

- on code un entier positif par sa représentation en base 2
- le nombre à coder doit être dans l'espace codable
- le code utilise toute la largeur fixée

Coder dans le cas général

- nombre de chiffres limités — **nombre de chiffres** $base = b$
- contrainte sur la taille des mots — **largeur des mots** $= k$
  - taille du mot affecte la quantité de nombres représentables
  - mot de k bits
    - entiers de 0 à $2^k - 1$
    - 2^k nombres
- représentation générale : $(\mathbf{c_{k-1}} \cdots \mathbf{c_3 c_2 c_1 c_0})_\mathbf{b}$

En pratique

- $b = 2$ (0 et 1)
- $k = 8$, $k = 16$, $k = 32$ ou $k = 64$

unsigned-k : entier non signé (codage)

codage naturel — unsigned-k

- on code un entier positif par sa représentation en base 2
- le nombre à coder doit être dans l'espace codable
- le code utilise toute la largeur fixée

Exemple

Avec un **octet** ($k = 8$)

- on code les entiers de **0** à **255**
- c'est-à-dire **256** nombres
- `00000000` code 0
- `11111111` code 255

Codage de sous-ensembles de $\mathbb{N}$

unsigned-k : entier non signé (arithmétique)

Addition

- Arithmétique identique en base 2 qu'en base 10
- Table d'addition
  - unités → `XOR`
  - retenue → `AND`
  - facilement automatisable

positionnelle

ADD	0	1
0	0 0	0 1
1	0 1	1 0

- Algorithme
  1. codage binaire des 2 nombres
  2. addition chiffre à chiffre de la droite vers la gauche avec gestion de la retenue
  3. retenue initiale à 0
- taille de mot fixe (k) ⇒ résultat de l'opération pas toujours représentable
  - si $x + y > 2^k - 1$ alors **débordement** (*overflow*)
  - repérable facilement : addition sur le bit de poids fort provoque une retenue à 1
  - les opérations avec débordement sont invalides

Exemple

unsigned-k : entier non signé (arithmétique)

Addition

- Arithmétique identique en base 2 qu'en base 10
- Table d'addition
  - unités → `XOR`
  - retenue → `AND`
  - facilement automatisable
- Algorithme
  1. codage binaire des 2 nombres
  2. addition chiffre à chiffre de la droite vers la gauche avec gestion de la retenue
  3. retenue initiale à 0
- taille de mot fixe (k) ⇒ résultat de l'opération pas toujours représentable
  - si $x + y > 2^k - 1$ alors **débordement** (*overflow*)
  - repérable facilement : addition sur le bit de poids fort provoque une retenue à 1
  - les opérations avec débordement sont invalides

positionnelle

ADD	0	1
0	**0 0**	0 1
1	0 1	1 0

Exemple

unsigned-k : entier non signé (arithmétique)

Addition

- Arithmétique identique en base 2 qu'en base 10
- Table d'addition
  - unités → `XOR`
  - retenue → `AND`
  - facilement automatisable
- Algorithme
  1. codage binaire des 2 nombres
  2. addition chiffre à chiffre de la droite vers la gauche avec gestion de la retenue
  3. retenue initiale à 0
- taille de mot fixe (k) ⇒ résultat de l'opération pas toujours représentable
  - si $x + y > 2^k - 1$ alors **débordement** (*overflow*)
  - repérable facilement : addition sur le bit de poids fort provoque une retenue à 1
  - les opérations avec débordement sont invalides

positionnelle

ADD	0	1
0	0 **0**	0 **1**
1	0 1	1 0

Exemple

unsigned-k : entier non signé (arithmétique)

Addition

- Arithmétique identique en base 2 qu'en base 10
- Table d'addition
  - unités → `XOR`
  - retenue → `AND`
  - facilement automatisable
- Algorithme
  1. codage binaire des 2 nombres
  2. addition chiffre à chiffre de la droite vers la gauche avec gestion de la retenue
  3. retenue initiale à 0
- taille de mot fixe (k) ⇒ résultat de l'opération pas toujours représentable
  - si $x + y > 2^k - 1$ alors **débordement** (*overflow*)
  - repérable facilement : addition sur le bit de poids fort provoque une retenue à 1
  - les opérations avec débordement sont invalides

positionnelle

ADD	0	1
0	0 **0**	0 **1**
1	0 **1**	1 0

Exemple

unsigned-k : entier non signé (arithmétique)

Addition

- Arithmétique identique en base 2 qu'en base 10
- Table d'addition
  - unités → `XOR`
  - retenue → `AND`
  - facilement automatisable
- Algorithme
  1. codage binaire des 2 nombres
  2. addition chiffre à chiffre de la droite vers la gauche avec gestion de la retenue
  3. retenue initiale à 0
- taille de mot fixe (k) ⇒ résultat de l'opération pas toujours représentable
  - si $x + y > 2^k - 1$ alors **débordement** (*overflow*)
  - repérable facilement : addition sur le bit de poids fort provoque une retenue à 1
  - les opérations avec débordement sont invalides

positionnelle

ADD	0	1
0	0 **0**	0 **1**
1	0 **1**	1 **0**

Exemple

unsigned-k : entier non signé (arithmétique)

Addition

- Arithmétique identique en base 2 qu'en base 10 *positionnelle*
- Table d'addition
  - unités → `XOR`
  - retenue → `AND`
  - facilement automatisable

ADD	0	1
0	0 **0**	0 **1**
1	0 **1**	1 **0**

- Algorithme
  1. codage binaire des 2 nombres
  2. addition chiffre à chiffre de la droite vers la gauche avec gestion de la retenue
  3. retenue initiale à 0
- taille de mot fixe (k) ⇒ résultat de l'opération pas toujours représentable
  - si $x + y > 2^k - 1$ alors **débordement** (*overflow*)
  - repérable facilement : addition sur le bit de poids fort provoque une retenue à 1
  - les opérations avec débordement sont invalides

Exemple

$10 + 3 = 13$

		0	0	0	0	1	0	1	0	(← 10)
+		0	0	0	0	0	0	1	1	(← 3)
+	0	0	0	0	0	0	1	0	0	Retenue
=		0	0	0	0	1	1	0	1	(→ 13)

unsigned-k : entier non signé (arithmétique)

Addition

- Arithmétique identique en base 2 qu'en base 10
- Table d'addition
  - unités → `XOR`
  - retenue → `AND`
  - facilement automatisable
- Algorithme
  1. codage binaire des 2 nombres
  2. addition chiffre à chiffre de la droite vers la gauche avec gestion de la retenue
  3. retenue initiale à 0
- taille de mot fixe (k) ⇒ résultat de l'opération pas toujours représentable
  - si $x + y > 2^k - 1$ alors **débordement** (*overflow*)
  - repérable facilement : addition sur le bit de poids fort provoque une retenue à 1
  - les opérations avec débordement sont invalides

positionnelle

ADD	0	1
0	0 **0**	0 **1**
1	0 **1**	1 **0**

Exemple

$10 + 3 = 13$

		0	0	0	0	1	0	1	0	(← 10)
+		0	0	0	0	0	0	1	1	(← 3)
+	0	0	0	0	0	0	1	0	0	Retenue
=		0	0	0	0	1	1	0	1	(→ 13)

unsigned-k : entier non signé (arithmétique)

Addition

- Arithmétique identique en base 2 qu'en base 10 *positionnelle*
- Table d'addition
  - unités → `XOR`
  - retenue → `AND`
  - facilement automatisable

ADD	0	1
0	0 **0**	0 **1**
1	0 **1**	1 **0**

- Algorithme
  1. codage binaire des 2 nombres
  2. addition chiffre à chiffre de la droite vers la gauche avec gestion de la retenue
  3. retenue initiale à 0
- taille de mot fixe (k) ⇒ résultat de l'opération pas toujours représentable
  - si $x + y > 2^k - 1$ alors **débordement** (*overflow*)
  - repérable facilement : addition sur le bit de poids fort provoque une retenue à 1
  - les opérations avec débordement sont invalides

Exemple

$128 + 128 = 256$

		1	0	0	0	0	0	0	0	(← 128)
+		1	0	0	0	0	0	0	0	(← 128)
+	1	0	0	0	0	0	0	0	0	Retenue
=		0	0	0	0	0	0	0	0	(→ 0)

unsigned-k : entier non signé (arithmétique)

Multiplication/Division

Somme de décalages

- multiplication = somme de décalage à gauche
  - décalage à gauche = multiplication par 2
  - uniquement pour les bits à 1 du multiplicateur
- division = somme de décalage à droite
  - décalage à droite = division par 2
- problème de débordement possible

Exemple

$13 \times 6 = 78$

			0	0	0	0	1	1	0	1	(← 13)
		×	0	0	0	0	0	1	1	0	(← 6)
			0	0	0	0	0	0	0	0	
+		0	0	0	0	1	1	0	1		
+	0	0	0	0	1	1	0	1			
+	0	0	0	1	1	0	0	0	0	0	Retenue
	0	0	0	1	0	0	1	1	1	0	(→ 78)

Entier signé

Objectifs : coder des entiers relatifs (dans $\mathbb{Z}$)

- codage en binaire
- représenter le signe et la valeur absolue
- préserver des opérations arithmétiques facile à faire

Codage naïf

- réserver un bit pour le signe
  - bit de poids fort
  - bit à 0 → signe positif
  - bit à 1 → signe négatif
- sur 1 octet on code de -127 à $+127$

Problèmes

- 0 est codé 2 fois : 10000000 et 00000000
- opérations arithmétiques difficiles à exécuter

Entier signé

Objectifs : coder des entiers relatifs (dans $\mathbb{Z}$)

- codage en binaire
- représenter le signe et la valeur absolue
- préserver des opérations arithmétiques facile à faire

Codage naïf

- réserver un bit pour le signe
  - bit de poids fort
  - bit à 0 → signe positif
  - bit à 1 → signe négatif
- sur 1 octet on code de -127 à $+127$

Problèmes

- 0 est codé 2 fois : 10000000 et 00000000
- opérations arithmétiques difficiles à exécuter

Entier signé

Objectifs : coder des entiers relatifs (dans $\mathbb{Z}$)

- codage en binaire
- représenter le signe et la valeur absolue
- préserver des opérations arithmétiques facile à faire

Codage naïf

- réserver un bit pour le signe
  - bit de poids fort
  - bit à 0 → signe positif
  - bit à 1 → signe négatif
- sur 1 octet on code de -127 à $+127$

Problèmes

- 0 est codé 2 fois : `10000000` et `00000000`
- opérations arithmétiques difficiles à exécuter

Des outils : les compléments

Pour un nombre n, en base b sur des nombres de k chiffres :

- le **complément à** $b-1$

  écart entre le **plus grand** nombre exprimable (b^k-1) et n

- le **complément à** b

  écart entre le **nombre** de nombres exprimables (b^k) et n

Appliqué à $b=2$

- Complément à 1 (CA1) : `NOT` bit à bit
- Complément à 2 (CA2) : `NOT` bit à bit puis ajout de 1

Des outils : les compléments

Pour un nombre n, en base b sur des nombres de k chiffres :

- le **complément à** $b-1$

  écart entre le **plus grand** nombre exprimable (b^k-1) et n

- le **complément à** b

  écart entre le **nombre** de nombres exprimables (b^k) et n

Appliqué à $b=2$

- **Complément à 1** (`CA1`) ..... `NOT` bit à bit
- **Complément à 2** (`CA2`) ..... `NOT` bit à bit puis ajout de 1

Exemple

	0	0	1	0	1	0	1	0	$\leftarrow 42 = n$
+	1	1	0	1	0	1	0	1	$\leftarrow 213 =$ `CA1`(42)
=	1	1	1	1	1	1	1	1	$\rightarrow 255 = 2^8-1$

Des outils : les compléments

Pour un nombre n, en base b sur des nombres de k chiffres :

- le **complément à** $b-1$

  écart entre le **plus grand** nombre exprimable (b^k-1) et n

- le **complément à** b

  écart entre le **nombre** de nombres exprimables (b^k) et n

Appliqué à $b=2$

- **Complément à 1** (`CA1`) ..... `NOT` bit à bit
- **Complément à 2** (`CA2`) ..... `NOT` bit à bit puis ajout de 1

Exemple

	0	0	1	0	1	0	1	0	$\leftarrow 42 = n$
+	1	1	0	1	0	1	1	0	$\leftarrow 214 =$ `CA2(42)`
$=1$	0	0	0	0	0	0	0	0	$\rightarrow 256 = 2^8$

signed-k : Entier signé (codage)

codage par complément à 2 sur k bits — signed-k

- un entier positif est codé en binaire naturel non signé sur k bits (`unsigned-k`)
- un entier négatif n est codé par son complément à deux sur k bits (`CA2`) :
  1. on convertit la valeur absolue $|n|$ en binaire
  2. on calcule son complément à deux
  3. on ne conserve que les k bits de poids faibles

Exemple

Par exemple pour $k = 4$ on représente -6 en complément à deux par 1010

on code la valeur absolue 6	$\rightarrow$	0110
on inverse bit à bit	$\rightarrow$	1001
on ajoute 1	$\rightarrow$	1010

signed-k : Entier signé (codage)

codage par complément à 2 sur k bits — signed-k

- un entier positif est codé en binaire naturel non signé sur k bits (`unsigned-k`)
- un entier négatif n est codé par son complément à deux sur k bits (`CA2`) :
  1. on convertit la valeur absolue $|n|$ en binaire
  2. on calcule son complément à deux
  3. on ne conserve que les k bits de poids faibles

Exemple

Par exemple pour $k = 4$ on représente -6 en complément à deux par 1010

on code la valeur absolue 6	$\rightarrow$	0110
on inverse bit à bit	$\rightarrow$	1001
on ajoute 1	$\rightarrow$	1010

signed-k : Entier signé (remarques)

Idée : *calcul modulo*

Représentation limitée à k bits implique arithmétique **modulo** 2^k

Exemple

Si $k = 4$ on calcule modulo 2^4, c'est-à-dire modulo 16

- sur 4 bits ajouter 15 revient à enlever 1
  - $(1 + 15)\%16 = 0$
  - $(2 + 15)\%16 = 1$
  - $(3 + 15)\%16 = 2$
  - etc.
- sur 4 bits ajouter 14 revient à enlever 2
  - $(1 + 14)\%16 = 15$
  - $(2 + 14)\%16 = 0$
  - $(3 + 14)\%16 = 1$
  - etc.
- le codage fait donc en sorte que
  - $15 \rightarrow -1$
  - $14 \rightarrow -2$,
  - etc.

signed-k : Entier signé (remarques)

Idée : *calcul modulo*

Représentation limitée à k bits implique arithmétique **modulo** 2^k

Exemple

Si $k = 4$ on calcule modulo 2^4, c'est-à-dire modulo 16

- sur 4 bits ajouter 15 revient à enlever 1
  - $(1 + 15)\%16 = 0$
  - $(2 + 15)\%16 = 1$
  - $(3 + 15)\%16 = 2$
  - etc.
- sur 4 bits ajouter 14 revient à enlever 2
  - $(1 + 14)\%16 = 15$
  - $(2 + 14)\%16 = 0$
  - $(3 + 14)\%16 = 1$
  - etc.
- le codage fait donc en sorte que
  - $15 \rightarrow -1$
  - $14 \rightarrow -2$,
  - etc.

signed-k : Entier signé (remarques)

Idée : *calcul modulo*

Représentation limitée à k bits implique arithmétique **modulo** 2^k

Exemple

Si $k = 4$ on calcule modulo 2^4, c'est-à-dire modulo 16

- sur 4 bits ajouter 15 revient à enlever 1
  - $(1 + 15)\%16 = 0$
  - $(2 + 15)\%16 = 1$
  - $(3 + 15)\%16 = 2$
  - etc.
- sur 4 bits ajouter 14 revient à enlever 2
  - $(1 + 14)\%16 = 15$
  - $(2 + 14)\%16 = 0$
  - $(3 + 14)\%16 = 1$
  - etc.
- le codage fait donc en sorte que
  - $15 \rightarrow -1$
  - $14 \rightarrow -2$,
  - etc.

signed-*k* : Entier signé (remarques)

avec unsigned-*4*	←	code	→	**avec signed-*4***
0	←	0000	→	0
1	←	0001	→	1
2	←	0010	→	2
3	←	0011	→	3
4	←	0100	→	3
5	←	0101	→	5
6	←	0110	→	6
7	←	0111	→	7
8	←	1000	→	-8
9	←	1001	→	-7
10	←	1010	→	-6
11	←	1011	→	-5
12	←	1100	→	-4
13	←	1101	→	-3
14	←	1110	→	-2
15	←	1111	→	-1

signed-k : Entier signé (remarques)

Avantages

- le bit de poids fort représente le signe
- on peut toujours coder 2^k nombres différents
  - tous les entiers entre -2^{k-1} et $2^{k-1}-1$
  - avec des octets on code de -128 à $+127$
- l'arithmétique est préservée

signed-k : Entier signé (remarques)

Attention

coder n en complément à 2 $\neq$ complémenter n à 2

`signed-`k $\neq$ `CA2`

Sur 4 bits

- 7 est codé en `signed-4` par `0111` — `signed-4(7) = 0111`
- le complément à 2 de 7 est 9 (1001) — `CA2(7) = 1001`

Entier signé (conversions)

$$n = (-c_{k-1} * 2^{k-1} + c_{k-2} * 2^{k-2} \cdots + c_2 * 2^2 + c_1 * 2^1 + c_0 * 2^0)_{10}$$

- on convertit le nombre sur les $k-1$ bits de poids faibles
- si le bit de poids fort est à 1 on retire 2^{k-1}

Exemple

Par exemple avec $k = 4$

`1011`

- `011` $\rightarrow 3$
- $3 - 8 = \mathbf{-5}$

`0011`

- `011` $\rightarrow \mathbf{+3}$

Entier signé (conversions)

$$n = (-c_{k-1} * 2^{k-1} + c_{k-2} * 2^{k-2} \cdots + c_2 * 2^2 + c_1 * 2^1 + c_0 * 2^0)_{10}$$

Exemple

$$
\begin{array}{lcll}
0000 & \rightarrow & -\mathbf{0} * 2^3 + \mathbf{0} * 2^2 + \mathbf{0} * 2^1 + \mathbf{0} * 2^0 & = 0 \\
0001 & \rightarrow & -\mathbf{0} * 2^3 + \mathbf{0} * 2^2 + \mathbf{0} * 2^1 + \mathbf{1} * 2^0 & = +1 \\
0010 & \rightarrow & -\mathbf{0} * 2^3 + \mathbf{0} * 2^2 + \mathbf{1} * 2^1 + \mathbf{0} * 2^0 & = +2 \\
0011 & \rightarrow & -\mathbf{0} * 2^3 + \mathbf{0} * 2^2 + \mathbf{1} * 2^1 + \mathbf{1} * 2^0 & = +3 \\
0100 & \rightarrow & -\mathbf{0} * 2^3 + \mathbf{1} * 2^2 + \mathbf{0} * 2^1 + \mathbf{0} * 2^0 & = +4 \\
0101 & \rightarrow & -\mathbf{0} * 2^3 + \mathbf{1} * 2^2 + \mathbf{0} * 2^1 + \mathbf{1} * 2^0 & = +5 \\
0110 & \rightarrow & -\mathbf{0} * 2^3 + \mathbf{1} * 2^2 + \mathbf{1} * 2^1 + \mathbf{0} * 2^0 & = +6 \\
0111 & \rightarrow & -\mathbf{0} * 2^3 + \mathbf{1} * 2^2 + \mathbf{1} * 2^1 + \mathbf{1} * 2^0 & = +7 \\
1000 & \rightarrow & -\mathbf{1} * 2^3 + \mathbf{0} * 2^2 + \mathbf{0} * 2^1 + \mathbf{0} * 2^0 & = -8 \\
1001 & \rightarrow & -\mathbf{1} * 2^3 + \mathbf{0} * 2^2 + \mathbf{0} * 2^1 + \mathbf{1} * 2^0 & = -7 \\
1010 & \rightarrow & -\mathbf{1} * 2^3 + \mathbf{0} * 2^2 + \mathbf{1} * 2^1 + \mathbf{0} * 2^0 & = -6 \\
1011 & \rightarrow & -\mathbf{1} * 2^3 + \mathbf{0} * 2^2 + \mathbf{1} * 2^1 + \mathbf{1} * 2^0 & = -5 \\
1100 & \rightarrow & -\mathbf{1} * 2^3 + \mathbf{1} * 2^2 + \mathbf{0} * 2^1 + \mathbf{0} * 2^0 & = -4 \\
1101 & \rightarrow & -\mathbf{1} * 2^3 + \mathbf{1} * 2^2 + \mathbf{0} * 2^1 + \mathbf{1} * 2^0 & = -3 \\
1110 & \rightarrow & -\mathbf{1} * 2^3 + \mathbf{1} * 2^2 + \mathbf{1} * 2^1 + \mathbf{0} * 2^0 & = -2 \\
1111 & \rightarrow & -\mathbf{1} * 2^3 + \mathbf{1} * 2^2 + \mathbf{1} * 2^1 + \mathbf{1} * 2^0 & = -1
\end{array}
$$

Entier signé (arithmétique)

Arithmétique simple

Définition

Pour ajouter deux nombres il suffit de faire la somme bit à bit comme on le fait dans le cas non signé.

La seule différence notable concerne la détection des débordements :

1. uniquement pour les additions de deux nombres aux signes identiques
2. détections
   - *humaine*
     signe du résultat ≠ signe (commun) des deux opérandes
   - automatique
     retenue finale ≠ retenue propagée sur le bit de poids fort

Entier signé (arithmétique)

		0	0	0	1	0	0	0	0	$\leftarrow 16$
+		0	0	0	0	1	0	1	1	$\leftarrow 11$
+	0	0	0	0	0	0	0	0	0	Retenue
=		0	0	0	1	1	0	1	1	$\rightarrow 27$

$\underbrace{16 + 11 = 27}$

OK

		0	1	1	1	1	1	1	1	$\leftarrow 127$
+		0	0	0	0	0	0	0	1	$\leftarrow 1$
+	0	1	1	1	1	1	1	1	0	Retenue
=		1	0	0	0	0	0	0	0	$\rightarrow -128$

$\underbrace{127 + 1 \neq -128}$

Débordement

Entier signé (arithmétique)

		0	0	0	1	0	0	0	0	$\leftarrow 16$
+		0	0	0	0	1	0	1	1	$\leftarrow 11$
+	0	0	0	0	0	0	0	0	0	Retenue
=		0	0	0	1	1	0	1	1	$\rightarrow 27$

$$\underbrace{16 + 11 = 27}$$

OK

		0	1	1	1	1	1	1	1	$\leftarrow 127$
+		0	0	0	0	0	0	0	1	$\leftarrow 1$
+	0	1	1	1	1	1	1	1	0	Retenue
=		1	0	0	0	0	0	0	0	$\rightarrow -128$

$$\underbrace{127 + 1 \neq -128}$$

Débordement

Entier signé (arithmétique)

		0	0	0	1	0	0	0	0	$\leftarrow 16$
+		0	0	0	0	1	0	1	1	$\leftarrow 11$
+	0	0	0	0	0	0	0	0	0	Retenue
=		0	0	0	1	1	0	1	1	$\rightarrow 27$

$\underbrace{16 + 11 = 27}$

OK

		0	1	1	1	1	1	1	1	$\leftarrow 127$
+		0	0	0	0	0	0	0	1	$\leftarrow 1$
+	0	1	1	1	1	1	1	1	0	Retenue
=		1	0	0	0	0	0	0	0	$\rightarrow -128$

$\underbrace{127 + 1 \neq -128}$

Débordement

Entier signé (arithmétique)

		0	0	0	1	0	0	0	0	$\leftarrow 16$
+		0	0	0	0	1	0	1	1	$\leftarrow 11$
+	0	0	0	0	0	0	0	0	0	Retenue
=		0	0	0	1	1	0	1	1	$\rightarrow 27$

$$\underbrace{16 + 11 = 27}$$

OK

		0	1	1	1	1	1	1	1	$\leftarrow 127$
+		0	0	0	0	0	0	0	1	$\leftarrow 1$
+	0	1	1	1	1	1	1	1	0	Retenue
=		1	0	0	0	0	0	0	0	$\rightarrow -128$

$$\underbrace{127 + 1 \neq -128}$$

Débordement

Représentation des réels

Un nombre réel x $\qquad x = E(x) + F(x)$

- possède une partie entière ..... $E(x)$ dans $\mathbb{Z}$
- possède une partie fractionnaire ..... $F(x)$ dans $[0,1[$
- en notation positionnelle $\qquad$ en base b
  - se représente avec une virgule en position **fixe** entre les deux parties

$$x = (c_k \cdots c_3 c_2 c_1 c_0 \,,\, \mathbf{c_{-1}} \mathbf{c_{-2}} \cdots \mathbf{c_{-j}})_b$$

Exemple $(12,34)$

En base 10

$$\mathbf{1} \times 10^1 + \mathbf{2} \times 10^0 + \mathbf{3} \times 10^{-1} + \mathbf{4} \times 10^{-2} = \frac{1234}{100}$$

- En base b on ne peut représenter exactement que des nombres de la forme $\frac{x}{b^n}$

Conversions

Pour convertir x de la base b à la base 10

$$x = (c_k \times b^k + \cdots + c_1 \times b^1 + c_0 \times b^0 + \mathbf{c_{-1}} \times \mathbf{b^{-1}} + \mathbf{c_{-2}} \times \mathbf{b^{-2}} + \cdots + \mathbf{c_{-j}} \times \mathbf{b^{-j}})_{10}$$

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}
0,5	0,25	0,125	0,0625	0,03125	0,015625	0,0078125	0,00390625	0,001953125	0,0009765625

Conversions

Pour convertir x de la base b à la base 10

$$x = (c_k \times b^k + \cdots + c_1 \times b^1 + c_0 \times b^0 + \mathbf{c_{-1}} \times \mathbf{b^{-1}} + \mathbf{c_{-2}} \times \mathbf{b^{-2}} + \cdots + \mathbf{c_{-j}} \times \mathbf{b^{-j}})_{10}$$

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}
0,5	0,25	0,125	0,0625	0,03125	0,015625	0,0078125	0,00390625	0,001953125	0,0009765625

Exemple ($x = 101,101$ et $b = 2$)

$$\begin{aligned} x &= \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0 + \mathbf{1} \times 2^{-1} + \mathbf{0} \times 2^{-2} + \mathbf{1} \times 2^{-3} \\ &= 4 + 1 + 0,5 + 0,125 \\ &= 5,625 \end{aligned}$$

Conversions

Pour convertir x de la base 10 à la base b on construit :

- sa partie entière, **de la droite vers la gauche**,
  - reste des divisions euclidiennes successives par b
  - jusqu'à quotient nul
- sa partie fractionnaire, **de la gauche vers la droite**,
  - en utilisant la **partie entière des multiplications successives** par b de la partie fractionnaire
  - jusqu'à partie fractionnaire nulle.

Conversions

Pour convertir x de la base 10 à la base b on construit :

- sa partie entière, **de la droite vers la gauche**,
  - reste des divisions euclidiennes successives par b
  - jusqu'à quotient nul
- sa partie fractionnaire, **de la gauche vers la droite**,
  - en utilisant la **partie entière des multiplications successives** par b de la partie fractionnaire
  - jusqu'à partie fractionnaire nulle.

Exemple ($x = 6,375$ et $b = 2$)

- partie entière $6 \rightarrow 110$
- partie fractionnaire : $0,375 \rightarrow 0,011$
  - $0,375 \times 2 = \mathbf{0},75$
  - $0,75 \times 2 = \mathbf{1},5$
  - $0,5 \times 2 = \mathbf{1},0$
- $x = (110,011)_2$

Notation plus concise

- virgule fixe incommode pour des valeurs très différentes
  - masse de la terre 597360000000000000000000 kg
  - masse d'un proton 0,000000000000000000000000000091093822 kg
- notation scientifique plus pratique

$$a \times 10^e$$

  - $e \in \mathbb{Z}$ et $1 \leq |a| < 10$
  - masse de la terre $5,9736 \times 10^{24}$ kg
  - masse d'un proton $9,1093822 \times 10^{-31}$ kg
  - position de la virgule dépend de l'exposant e

Définition (généralisation de la virgule flottante)

x en base b est représenté comme le produit

$$x = S \times M \times b^E$$

- S est le **signe**, $S \in \{-1, +1\}$
- M est la **mantisse** et précise les chiffres significatifs
- E est l'**exposant** qui donne l'ordre de grandeur (la position de la virgule) $E \in \mathbb{Z}$

Notation plus concise

- virgule fixe incommode pour des valeurs très différentes
  - masse de la terre $5973600000000000000000000$ kg
  - masse d'un proton $0,000000000000000000000000000091093822$ kg
- notation scientifique plus pratique

$$a \times 10^e$$

  - $e \in \mathbb{Z}$ et $1 \leq |a| < 10$
  - masse de la terre $5,9736 \times 10^{24}$ kg
  - masse d'un proton $9,1093822 \times 10^{-31}$ kg
  - position de la virgule dépend de l'exposant e

Définition (généralisation de la virgule flottante)

x en base b est représenté comme le produit

$$x = S \times M \times b^E$$

- S est le **signe**, $S \in \{-1, +1\}$
- M est la **mantisse** et précise les chiffres significatifs
- E est l'**exposant** qui donne l'ordre de grandeur (la position de la virgule) $E \in \mathbb{Z}$

Notation plus concise

- virgule fixe incommode pour des valeurs très différentes
  - masse de la terre $5973600000000000000000000$ kg
  - masse d'un proton $0,000000000000000000000000000091093822$ kg
- notation scientifique plus pratique

$$a \times 10^e$$

  - $e \in \mathbb{Z}$ et $1 \leq |a| < 10$
  - masse de la terre $5,9736 \times 10^{24}$ kg
  - masse d'un proton $9,1093822 \times 10^{-31}$ kg
  - position de la virgule dépend de l'exposant e

Définition (généralisation de la virgule flottante)

x en base b est représenté comme le produit

$$x = S \times M \times b^E$$

- S est le **signe**, $S \in \{-1, +1\}$
- M est la **mantisse** et précise les chiffres significatifs
- E est l'**exposant** qui donne l'ordre de grandeur (la position de la virgule) $E \in \mathbb{Z}$

Codage IEEE 754

codage IEEE-754

- on représente un réel x via une notation en virgule flottante **binaire** *adaptée*

$$\begin{aligned} x = & \quad (-1)^{s} \times \quad (1 + \sum_{i=1}^{\mu} m_i \times 2^{-i})_2 \quad \times 2^{e - e_{max}} \\ = & \quad (-1)^{s} \times \quad (1, m_1 m_2 \cdots m_\mu)_2 \quad \times 2^{e - e_{max}} \end{aligned}$$

- on code en plaçant successivement (du poids fort au poids faible) :
  1. **s** sur 1 bit pour le signe — 0 pour + et 1 pour −
  2. **e** sur η bits
  3. **m** sur μ bits — $m = m_1 m_2 \cdots m_\mu$ avec $m_i \in \{0, 1\}$

Codage IEEE 754

codage IEEE-754

- on représente un réel x via une notation en virgule flottante **binaire** *adaptée*

$$\begin{aligned} x = & \quad (-1)^{s} \times & (1 + \textstyle\sum_{i=1}^{\mu} m_i \times 2^{-i})_2 & \quad \times 2^{e - e_{max}} \\ = & \quad (-1)^{s} \times & (1, m_1 m_2 \cdots m_\mu)_2 & \quad \times 2^{e - e_{max}} \end{aligned}$$

- on code en plaçant successivement (du poids fort au poids faible) :
  1. **s** sur 1 bit pour le signe — 0 pour + et 1 pour −
  2. **e** sur η bits
  3. **m** sur μ bits — $m = m_1 m_2 \cdots m_\mu$ avec $m_i \in \{0, 1\}$

Codage IEEE 754

codage IEEE-754

- on représente un réel x via une notation en virgule flottante **binaire** *adaptée*

$$\begin{aligned} x = & \quad (-1)^{s} \times & (1 + \textstyle\sum_{i=1}^{\mu} m_i \times 2^{-i})_2 & \times 2^{e - e_{max}} \\ = & \quad (-1)^{s} \times & (1, m_1 m_2 \cdots m_\mu)_2 & \times 2^{e - e_{max}} \end{aligned}$$

- on code en plaçant successivement (du poids fort au poids faible) :
  1. **s** sur 1 bit pour le signe — 0 pour + et 1 pour −
  2. **e** sur η bits
  3. **m** sur μ bits — $m = m_1 m_2 \cdots m_\mu$ avec $m_i \in \{0, 1\}$

Codage IEEE 754

codage IEEE-754

- on représente un réel x via une notation en virgule flottante **binaire** *adaptée*

$$\begin{aligned} x = & \quad (-1)^{s} \times & (1 + \textstyle\sum_{i=1}^{\mu} m_i \times 2^{-i})_2 & \times 2^{e - e_{max}} \\ = & \quad (-1)^{s} \times & (1, m_1 m_2 \cdots m_\mu)_2 & \times 2^{e - e_{max}} \end{aligned}$$

- on code en plaçant successivement (du poids fort au poids faible) :
  1. **s** sur 1 bit pour le signe — 0 pour + et 1 pour −
  2. **e** sur η bits
  3. **m** sur μ bits — $m = m_1 m_2 \cdots m_\mu$ avec $m_i \in \{0, 1\}$

Codage IEEE 754

codage IEEE-754

- on représente un réel x via une notation en virgule flottante **binaire** *adaptée*

$$\begin{aligned} x = & \quad (-1)^{s} \times & (1 + \textstyle\sum_{i=1}^{\mu} m_i \times 2^{-i})_2 & \times 2^{e - e_{max}} \\ = & \quad (-1)^{s} \times & (1, m_1 m_2 \cdots m_\mu)_2 & \times 2^{e - e_{max}} \end{aligned}$$

- on code en plaçant successivement (du poids fort au poids faible) :
  1. **s** sur 1 bit pour le signe — 0 pour + et 1 pour −
  2. **e** sur η bits
  3. **m** sur μ bits — $m = m_1 m_2 \cdots m_\mu$ avec $m_i \in \{0, 1\}$

Codage IEEE 754

codage IEEE-754

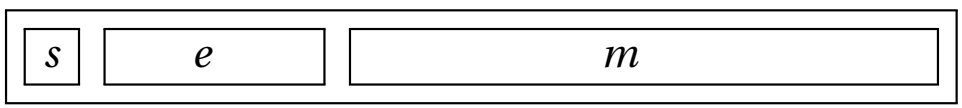
- on représente un réel x via une notation en virgule flottante **binaire** *adaptée*

$$\begin{aligned} x = & \quad (-1)^{s} \times & (1 + \textstyle\sum_{i=1}^{\mu} m_i \times 2^{-i})_2 & \times 2^{e - e_{max}} \\ = & \quad (-1)^{s} \times & (1, m_1 m_2 \cdots m_\mu)_2 & \times 2^{e - e_{max}} \end{aligned}$$

- on code en plaçant successivement (du poids fort au poids faible) :
  1. **s** sur 1 bit pour le signe — 0 pour + et 1 pour −
  2. **e** sur η bits
  3. **m** sur μ bits — $m = m_1 m_2 \cdots m_\mu$ avec $m_i \in \{0, 1\}$

simple précision (FLOAT) ..... 4 octets

1. s — 1 bit
2. e — $(e_{max} = 127)$ — 8 bits
3. m — 23 bits



Codage IEEE 754

codage IEEE-754

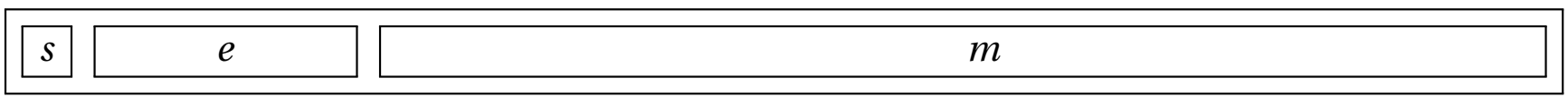
- on représente un réel x via une notation en virgule flottante **binaire** *adaptée*

$$\begin{aligned} x = & \quad (-1)^{s} \times & (1 + \textstyle\sum_{i=1}^{\mu} m_i \times 2^{-i})_2 & \times 2^{e - e_{max}} \\ = & \quad (-1)^{s} \times & (1, m_1 m_2 \cdots m_\mu)_2 & \times 2^{e - e_{max}} \end{aligned}$$

- on code en plaçant successivement (du poids fort au poids faible) :
  1. **s** sur 1 bit pour le signe — 0 pour + et 1 pour −
  2. **e** sur η bits
  3. **m** sur μ bits — $m = m_1 m_2 \cdots m_\mu$ avec $m_i \in \{0, 1\}$

double précision (DOUBLE) ..... 8 octets

1. s — 1 bit
2. e — $(e_{max} = 1023)$ — 11 bits
3. m — 52 bits



IEEE 754 : nombre réels (conversions)

Codage

Exemple (simple précision)

Codons $x = 6,625$

1. Conversion binaire : $6,625 \rightarrow 110,101$
2. Représentation en virgule flottante : $+1,10101 \times 2^2$
3. Représentation IEEE-754 :
   - $s = 0$ $\qquad -1^0 = +1$
   - $1,m = 1,10101$
   - $e - 127 = 2$, donc $e = 129$ $\qquad 2^{8-1} - 1 = 127$
4. Codage IEEE-754
   - `0 10000001 10101000000000000000000`
   - `01000000 11010100 0000000 000000000`
   - $6,625 \rightarrow$ `0x40 d4 00 00`

IEEE 754 : nombre réels (conversions)

Codage

Exemple (simple précision)

Codons $x = 6,625$

1. Conversion binaire : $6,625 \rightarrow 110,101$
2. Représentation en virgule flottante : $+1,10101 \times 2^2$
3. Représentation IEEE-754 :
   - $s = 0$ $\qquad -1^0 = +1$
   - $1,m = 1,10101$
   - $e - 127 = 2$, donc $e = 129$ $\qquad 2^{8-1} - 1 = 127$
4. Codage IEEE-754
   - 0 10000001 10101000000000000000000
   - 01000000 11010100 0000000 000000000
   - $6,625 \rightarrow$ 0x40 d4 00 00

IEEE 754 : nombre réels (conversions)

Codage

Exemple (simple précision)

Codons $x = 6,625$

1. Conversion binaire : $6,625 \rightarrow 110,101$
2. Représentation en virgule flottante : $+1,10101 \times 2^2$
3. Représentation IEEE-754 :
   - $s = 0$ $\qquad -1^0 = +1$
   - $1,m = 1,10101$
   - $e - 127 = 2$, donc $e = 129$ $\qquad 2^{8-1} - 1 = 127$
4. Codage IEEE-754
   - `0 10000001 10101000000000000000000`
   - `01000000 11010100 0000000 000000000`
   - $6,625 \rightarrow$ `0x40 d4 00 00`

IEEE 754 : nombre réels (conversions)

Codage

Exemple (simple précision)

Codons $x = 6,625$

1. Conversion binaire : $6,625 \rightarrow \texttt{110,101}$
2. Représentation en virgule flottante : $+1,10101 \times 2^2$
3. Représentation `IEEE-754` :
   - $s = 0$ $\qquad -1^0 = +1$
   - $1,m = 1,10101$
   - $e - 127 = 2$, donc $e = 129$ $\qquad 2^{8-1} - 1 = 127$
4. Codage `IEEE-754`
   - `0 10000001 10101000000000000000000`
   - `01000000 11010100 0000000 000000000`
   - $6,625 \rightarrow$ `0x40 d4 00 00`

IEEE 754 : nombre réels (conversions)

Codage

Exemple (simple précision)

Codons $x = 6,625$

1. Conversion binaire : $6,625 \rightarrow 110,101$
2. Représentation en virgule flottante : $+1,10101 \times 2^2$
3. Représentation `IEEE-754` :
   - $s = 0$ — $-1^0 = +1$
   - $1,m = 1,10101$
   - $e - 127 = 2$, donc $e = 129$ — $2^{8-1} - 1 = 127$
4. Codage `IEEE-754`
   - `0 10000001 10101000000000000000000`
   - `01000000 11010100 0000000 000000000`
   - $6,625 \rightarrow$ `0x40 d4 00 00`

IEEE 754 : nombre réels (conversions)

Décodage

Exemple (simple précision)

Décodons $x = \texttt{0xc1180000}$

La valeur d'un nombre est

$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i \times 2^i) \times 2^{e-e_{max}}$$

1. Conversion binaire
   - 0xc1180000
   - 11000001000110000000000000000000
2. Découpage binaire
   - 1 10000010 00110000000000000000000
   - $s = (-1)^1 = -1$
   - $e - e_{max} = 130 - 127 = 3$ $\quad (2^7 + 2^1) = 130$
   - $m = 2^{-3} + 2^{-4} = 0,125 + 0,0625 = 0,1875$
3. Calcul de la valeur : $x = -1,1875 \times 2^3 = \mathbf{-9,5}$

IEEE 754 : nombre réels (conversions)

Décodage

Exemple (simple précision)

Décodons $x = \texttt{0xc1180000}$
La valeur d'un nombre est

$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i \times 2^i) \times 2^{e-e_{max}}$$

1. Conversion binaire
   - 0xc1180000
   - 11000001000110000000000000000000
2. Découpage binaire
   - 1 10000010 00110000000000000000000
   - $s = (-1)^1 = -1$
   - $e - e_{max} = 130 - 127 = 3$ $\quad (2^7 + 2^1) = 130$
   - $m = 2^{-3} + 2^{-4} = 0,125 + 0,0625 = 0,1875$
3. Calcul de la valeur : $x = -1,1875 \times 2^3 = -9,5$

IEEE 754 : nombre réels (conversions)

Décodage

Exemple (simple précision)

Décodons $x =$ `0xc1180000`
La valeur d'un nombre est

$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i \times 2^i) \times 2^{e-e_{max}}$$

1. Conversion binaire
   - `0xc1180000`
   - `11000001000110000000000000000000`
2. Découpage binaire
   - `1 10000010 00110000000000000000000`
   - $s = (-1)^1 = -1$
   - $e - e_{max} = 130 - 127 = 3$ $\quad (2^7 + 2^1) = 130$
   - $m = 2^{-3} + 2^{-4} = 0,125 + 0,0625 = 0,1875$
3. Calcul de la valeur : $x = -1,1875 \times 2^3 = -9,5$

IEEE 754 : nombre réels (conversions)

Décodage

Exemple (simple précision)

Décodons $x =$ `0xc1180000`
La valeur d'un nombre est

$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i \times 2^i) \times 2^{e-e_{max}}$$

1. Conversion binaire
   - `0xc1180000`
   - `11000001000110000000000000000000`
2. Découpage binaire
   - `1 10000010 00110000000000000000000`
   - $s = (-1)^1 = -1$
   - $e - e_{max} = 130 - 127 = 3$ $\quad (2^7 + 2^1) = 130$
   - $m = 2^{-3} + 2^{-4} = 0,125 + 0,0625 = 0,1875$
3. Calcul de la valeur : $x = -1,1875 \times 2^3 = -9,5$

IEEE 754 : nombre réels (conversions)

Décodage

Exemple (simple précision)

Décodons $x = \texttt{0xc1180000}$
La valeur d'un nombre est

$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i \times 2^i) \times 2^{e-e_{max}}$$

1. Conversion binaire
   - `0xc1180000`
   - `11000001000110000000000000000000`
2. Découpage binaire
   - `1 10000010 00110000000000000000000`
   - $s = (-1)^1 = -1$
   - $e - e_{max} = 130 - 127 = 3$ $\quad (2^7 + 2^1) = 130$
   - $m = 2^{-3} + 2^{-4} = 0,125 + 0,0625 = 0,1875$
3. Calcul de la valeur : $x = -1,1875 \times 2^3 = -9,5$

IEEE 754 : nombre réels (conversions)

Décodage

Exemple (simple précision)

Décodons $x =$ `0xc1180000`
La valeur d'un nombre est

$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i \times 2^i) \times 2^{e-e_{max}}$$

1. Conversion binaire
   - `0xc1180000`
   - `11000001000110000000000000000000`
2. Découpage binaire
   - `1 10000010 00110000000000000000000`
   - $s = (-1)^1 = -1$
   - $e - e_{max} = 130 - 127 = 3$ $\qquad (2^7 + 2^1) = 130$
   - $m = 2^{-3} + 2^{-4} = 0,125 + 0,0625 = 0,1875$
3. Calcul de la valeur : $x = -1,1875 \times 2^3 = -9,5$

IEEE 754 : nombre réels (conversions)

Décodage

Exemple (simple précision)

Décodons $x =$ `0xc1180000`
La valeur d'un nombre est

$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i \times 2^i) \times 2^{e-e_{max}}$$

1. Conversion binaire
   - `0xc1180000`
   - `11000001000110000000000000000000`
2. Découpage binaire
   - `1 10000010 00110000000000000000000`
   - $s = (-1)^1 = -1$
   - $e - e_{max} = 130 - 127 = 3$ $\qquad (2^7 + 2^1) = 130$
   - $m = 2^{-3} + 2^{-4} = 0,125 + 0,0625 = 0,1875$
3. Calcul de la valeur : $x = -1,1875 \times 2^3 = -9,5$

IEEE 754 : nombre réels (conversions)

Décodage

Exemple (simple précision)

Décodons $x = \texttt{0xc1180000}$
La valeur d'un nombre est

$$(-1)^s \times (1 + \sum_{i=1}^{23} m_i \times 2^i) \times 2^{e-e_{max}}$$

1. Conversion binaire
   - `0xc1180000`
   - `11000001000110000000000000000000`
2. Découpage binaire
   - `1 10000010 00110000000000000000000`
   - $s = (-1)^1 = -1$
   - $e - e_{max} = 130 - 127 = 3$ $\qquad (2^7 + 2^1) = 130$
   - $m = 2^{-3} + 2^{-4} = 0,125 + 0,0625 = 0,1875$
3. Calcul de la valeur : $x = -1,1875 \times 2^3 = \mathbf{-9,5}$

IEEE 754 : nombres réels (remarques)

Avantage du standard

- on peut facilement comparer deux nombres
- on obtient le successeur d'un nombre en ajoutant 1 à la représentation binaire
- le codage de 0 ne contient que des 0
- on a un code pour
  - $\pm\infty$
  - `NaN` (Not A Number) = *résultat d'opérations impossible*

Optimisations par rapport à un représentation flottante *classique*

- **normalisation de la mantisse** et **exposant biaisé** permettent de simplifier
  - la comparaison
  - l'incrémentation
  - codage simple de 0
- **bit caché**
  - économie de bits
  - codage de ∞ et `NaN`
  - codage simple de 0

IEEE 754 : nombres réels (remarques)

Avantage du standard

- on peut facilement comparer deux nombres
- on obtient le successeur d'un nombre en ajoutant 1 à la représentation binaire
- le codage de 0 ne contient que des 0
- on a un code pour
  - $\pm\infty$
  - `NaN` (Not A Number) = *résultat d'opérations impossible*

Optimisations par rapport à un représentation flottante *classique*

- **normalisation de la mantisse** et **exposant biaisé** permettent de simplifier
  - la comparaison
  - l'incrémentation
  - codage simple de 0
- **bit caché**
  - économie de bits
  - codage de ∞ et `NaN`
  - codage simple de 0

Normalisation

Problème avec la mantisse *classique* sans contraintes

plusieurs représentations possibles d'un même nombre

- $+1 \times \mathbf{0,110} \times 2^5$
- $+1 \times \mathbf{110} \times 2^2$
- $+1 \times \mathbf{0,0110} \times 2^6$
- etc.

Solution : une convention $\frac{1}{b} \leq m < 1$

- $\frac{1}{b} \leq m < 1$, *i.e.* tous les chiffres significatifs sont à droite de la virgule
- le premier chiffre (*i.e.* poids fort) est différent de 0
- la mantisse normalisée ne peut pas représenter le nombre 0
  - son premier chiffre est différent de 0
  - par conséquent la représentation de 0 nécessite un traitement spécifique

Exemple

- masse de la terre : $+0,59736 \times 10^{25}$ kg
- masse d'un proton : $+0,91093822 \times 10^{-30}$ kg

Normalisation

Problème avec la mantisse *classique* sans contraintes

plusieurs représentations possibles d'un même nombre

- $+1 \times \mathbf{0,110} \times 2^5$
- $+1 \times \mathbf{110} \times 2^2$
- $+1 \times \mathbf{0,0110} \times 2^6$
- etc.

Solution : une convention $\frac{1}{b} \leq m < 1$

- $\frac{1}{b} \leq m < 1$, *i.e.* tous les chiffres significatifs sont à droite de la virgule
- le premier chiffre (*i.e.* poids fort) est différent de 0
- la mantisse normalisée ne peut pas représenter le nombre 0
  - son premier chiffre est différent de 0
  - par convention la représentation de 0 ne contient que des 0.

Exemple

masse de la terre	$+0,59736 \times 10^{25}$ kg
masse d'un proton	$+0,91093822 \times 10^{-30}$ kg

Normalisation

Problème avec la mantisse *classique* sans contraintes

plusieurs représentations possibles d'un même nombre

- $+1 \times \mathbf{0,110} \times 2^5$
- $+1 \times \mathbf{110} \times 2^2$
- $+1 \times \mathbf{0,0110} \times 2^6$
- etc.

Solution : une convention $\frac{1}{b} \leq m < 1$

- $\frac{1}{b} \leq m < 1$, *i.e.* tous les chiffres significatifs sont à droite de la virgule
- le premier chiffre (*i.e.* poids fort) est différent de 0
- la mantisse normalisée ne peut pas représenter le nombre 0
  - son premier chiffre est différent de 0
  - par convention la représentation de 0 ne contient que des 0.

Exemple

masse de la terre	$+0,59736 \times 10^{25}$ kg
masse d'un proton	$+0,91093822 \times 10^{-30}$ kg

Normalisation

Problème avec la mantisse *classique* sans contraintes

plusieurs représentations possibles d'un même nombre

- $+1 \times \mathbf{0,110} \times 2^5$
- $+1 \times \mathbf{110} \times 2^2$
- $+1 \times \mathbf{0,0110} \times 2^6$
- etc.

Solution : une convention $\frac{1}{b} \leq m < 1$

- $\frac{1}{b} \leq m < 1$, *i.e.* tous les chiffres significatifs sont à droite de la virgule
- le premier chiffre (*i.e.* poids fort) est différent de 0
- la mantisse normalisée ne peut pas représenter le nombre 0
  - son premier chiffre est différent de 0
  - par convention la représentation de 0 ne contient que des 0.

Exemple

masse de la terre	$+0,59736 \times 10^{25}$ kg
masse d'un proton	$+0,91093822 \times 10^{-30}$ kg

Codage *biaisé*

Translation de l'intervalle de représentation

- comparer des nombres codés en complément à 2 sur k bits est difficile
- au lieu de représenter les entiers dans $[-2^{k-1}, 2^{k-1}-1]$
- on représente dans $[0, 2^k - 1]$

Pour coder un entier relatif de la base 10 à la base 2 en le **biaisant**

1. on ajoute un biais (2^{k-1})
2. on code le résultat avec `unsigned`

Pour décoder un entier relatif **biaisé** de la base 2 à la base 10

1. on décode l'entier par `unsigned` (sans complément)
2. on retranche le biais (2^{k-1})

Exemple

Codage *biaisé*

Translation de l'intervalle de représentation

- comparer des nombres codés en complément à 2 sur k bits est difficile
- au lieu de représenter les entiers dans $[-2^{k-1}, 2^{k-1}-1]$
- on représente dans $[0, 2^k - 1]$

Pour coder un entier relatif de la base 10 à la base 2 en le **biaisant**

1. on ajoute un biais (2^{k-1})
2. on code le résultat avec `unsigned`

Pour décoder un entier relatif **biaisé** de la base 2 à la base 10

1. on décode l'entier par `unsigned` (sans complément)
2. on retranche le biais (2^{k-1})

Exemple

Codage *biaisé*

Translation de l'intervalle de représentation

- comparer des nombres codés en complément à 2 sur k bits est difficile
- au lieu de représenter les entiers dans $[-2^{k-1}, 2^{k-1}-1]$
- on représente dans $[0, 2^k - 1]$

Pour coder un entier relatif de la base 10 à la base 2 en le **biaisant**

1. on ajoute un biais (2^{k-1})
2. on code le résultat avec `unsigned`

Pour décoder un entier relatif **biaisé** de la base 2 à la base 10

1. on décode l'entier par `unsigned` (sans complément)
2. on retranche le biais (2^{k-1})

Exemple

Codage *biaisé*

Translation de l'intervalle de représentation

- comparer des nombres codés en complément à 2 sur k bits est difficile
- au lieu de représenter les entiers dans $[-2^{k-1}, 2^{k-1}-1]$
- on représente dans $[0, 2^k-1]$

Pour coder un entier relatif de la base 10 à la base 2 en le **biaisant**

1. on ajoute un biais (2^{k-1})
2. on code le résultat avec `unsigned`

Pour décoder un entier relatif **biaisé** de la base 2 à la base 10

1. on décode l'entier par `unsigned` (sans complément)
2. on retranche le biais (2^{k-1})

Exemple

Sur un octet, -42

- en codage à complément à 2 ..... `11010110`
- en représentation biaisée à 128 ..... `01010110`
  - le biais est 128 (2^{8-1})
  - -42 + 128 = 86

Codage *biaisé*

Translation de l'intervalle de représentation

- comparer des nombres codés en complément à 2 sur k bits est difficile
- au lieu de représenter les entiers dans $[-2^{k-1}, 2^{k-1}-1]$
- on représente dans $[0, 2^k-1]$

Pour coder un entier relatif de la base 10 à la base 2 en le **biaisant**

1. on ajoute un biais (2^{k-1})
2. on code le résultat avec `unsigned`

Pour décoder un entier relatif **biaisé** de la base 2 à la base 10

1. on décode l'entier par `unsigned` (sans complément)
2. on retranche le biais (2^{k-1})

Exemple

Sur un octet, +42

- en codage à complément à 2 ..... `00110100`
- en représentation biaisée à 128 ..... `11010100`
  - le biais est 128 (2^{8-1})
  - 42 + 128 = 170

Codage biaisé (exemple)

Exemple

Sans biais : `signed-`k

$$
\begin{aligned}
1000 &\rightarrow -8 \\
1001 &\rightarrow -7 \\
1010 &\rightarrow -6 \\
1011 &\rightarrow -5 \\
1100 &\rightarrow -4 \\
1101 &\rightarrow -3 \\
1110 &\rightarrow -2 \\
1111 &\rightarrow -1 \\
0000 &\rightarrow 0 \\
0001 &\rightarrow +1 \\
0010 &\rightarrow +2 \\
0011 &\rightarrow +3 \\
0100 &\rightarrow +4 \\
0101 &\rightarrow +5 \\
0110 &\rightarrow +6 \\
0111 &\rightarrow +7
\end{aligned}
$$

Codage biaisé (exemple)

Exemple

Avec biais de 2^{4-1} : `unsigned`

$$
\begin{aligned}
0000 &\rightarrow -8 \\
0001 &\rightarrow -7 \\
0010 &\rightarrow -6 \\
0011 &\rightarrow -5 \\
0100 &\rightarrow -4 \\
0101 &\rightarrow -3 \\
0110 &\rightarrow -2 \\
0111 &\rightarrow -1 \\
1000 &\rightarrow 0 \\
1001 &\rightarrow +1 \\
1010 &\rightarrow +2 \\
1011 &\rightarrow +3 \\
1100 &\rightarrow +4 \\
1101 &\rightarrow +5 \\
1110 &\rightarrow +6 \\
1111 &\rightarrow +7
\end{aligned}
$$

Bit caché

- normalisation ⇒ les chiffres significatifs de la mantisse sont à droite de la virgule
- en base 2 le chiffre juste à droite de la virgule est donc toujours 1
- représenter explicitement ce chiffre ne sert donc à rien et *gâche* un 1 bit

IEEE 754 cache donc ce bit (en considérant qu'il est toujours présent)

- on *gagne* 1 bit sur la représentation
  - 2 valeurs disponibles en plus
  - ∞ et `NaN`
- on a 1 bit significatif de plus