Codage des données en machine.

Plan

Soit $\beta \in \mathbb{N}$, $\beta > 1$, une base. Tout $n \in \mathbb{N}$ peut être représenté de manière unique par sa *représentation positionnelle en base* β :

$$(x_{p-1}x_{p-2}\cdots x_1x_0)_\beta := \sum_{i=0}^{p-1} x_i\beta^i.$$

- Les $x_i \in \{0, 1, \ldots, \beta - 1\}$ sont les *chiffres* de l'écriture de n en base β.
- On attribue un symbole à chaque chiffre :
  - $\beta = 2$, chiffres 0 et 1 ;
  - $\beta = 10$, chiffres de 0 à 9 ;
  - $\beta = 16$, chiffres de 0 à F.
- p est le nombre de chiffres nécessaires pour écrire de l'entier naturel n.

Ex : $(5134)_{10} = 5{\cdot}10^3 + 1{\cdot}10^2 + 3{\cdot}10 + 4 = 5134$ en écriture décimale.

Plan

Utiliser la définition de la notation positionnelle

Soient β la base de départ et γ celle d'arrivée. On écrit $n = \sum_{i=0}^{p-1} x_i \beta^i$.

Il est toujours possible de

- convertir les x_i et β vers leurs écritures en base γ,
- de calculer $\sum_{i=0}^{p-1} x_i \beta^i$ en effectuant toutes les opérations en base γ.

On déduit l'écriture de n en base γ : il suffit de calculer dans la base d'arrivée γ.

Ex - conversion binaire vers décimal : avec $n = (10100)_2$. On ajoute les puissances de 2 correspondant aux bits non-nuls.

Utiliser la définition de la notation positionnelle

Soient β la base de départ et γ celle d'arrivée. On écrit $n = \sum_{i=0}^{p-1} x_i \beta^i$.

Il est toujours possible de

- convertir les x_i et β vers leurs écritures en base γ,
- de calculer $\sum_{i=0}^{p-1} x_i \beta^i$ en effectuant toutes les opérations en base γ.

On déduit l'écriture de n en base γ : il suffit de calculer dans la base d'arrivée γ.

Ex - conversion binaire vers décimal : avec $n = (10100)_2$. On ajoute les puissances de 2 correspondant aux bits non-nuls.

chiffre	1	0	1	0	0
position	4	3	2	1	0
poids	2^4	0	2^2	0	0

Donc $(10100)_2 = 2^4 + 2^2 = 16 + 4 = 20$.

Divisions euclidiennes successives

Le reste dans la division euclidienne d'un entier naturel n par β donne le chiffre de poids faible dans la représentation de n en base β. En effet,

$$\begin{aligned} n &= x_{p-1}\cdot\beta^{p-1} + x_{p-2}\cdot\beta^{p-2} + \cdots + x_2\cdot\beta^2 + x_1\cdot\beta^1 + x_0, \\ &= \underbrace{\left(x_{p-1}\cdot\beta^{p-2} + x_{p-2}\cdot\beta^{p-3} + \cdots + x_2\cdot\beta^1 + x_1\right)}_{\text{quotient}}\cdot\beta + \underbrace{x_0}_{\text{reste}}, \end{aligned}$$

avec $0 \leq x_0 < \beta$. En d'autres termes, $n \mod \beta = x_0$.

On peut donc obtenir tous les chiffres de l'écriture d'un entier naturel n par des divisions euclidiennes successives, en s'arrêtant au premier quotient nul :
on obtient les chiffres de poids faibles d'abord !

Ex : soit $n = (423)_{10}$, à convertir en base 2.

$$\begin{aligned}
423 &= 211 \times 2 + 1 \\
211 &= 105 \times 2 + 1 \\
105 &= 52 \times 2 + 1 \\
52 &= 26 \times 2 + 0 \\
26 &= 13 \times 2 + 0 \\
13 &= 6 \times 2 + 1 \\
6 &= 3 \times 2 + 0 \\
3 &= 1 \times 2 + 1 \\
1 &= 0 \times 2 + 1
\end{aligned}$$

On en déduit que $n = (110100111)_2$.

Ex : Conversion décimal vers octal. Soit $n = (3452)_{10}$, à convertir en base 8.

$$\begin{aligned}
3452 &= 431 \times 8 + 4 \\
431 &= 53 \times 8 + 7 \\
53 &= 6 \times 8 + 5 \\
6 &= 0 \times 8 + 6
\end{aligned}$$

Donc $n = (6574)_8$.

Entre les bases 2, 8 et 16 : méthodes directes

Comme $8 = 2^3$, chaque chiffre octal est représenté par un entier sur trois bits, et chaque entier sur trois bits est représenté par un chiffre octal :

$$(x_8x_7x_6x_5x_4x_3x_2x_1x_0)_2 = \underbrace{(x_8x_7x_6)_2}_{=y_2}\cdot 8^2 + \underbrace{(x_5x_4x_3)_2}_{=y_1}\cdot 8^1 + \underbrace{(x_2x_1x_0)_2}_{=y_0}\cdot 8^0 = (y_2y_1y_0)_8.$$

Ex - octal vers binaire : $n = (34521)_8$.

On fait un tableau de conversion entre nombres binaires et chiffres octaux :

rep. binaire	chiffre octal	rep. binaire	chiffre octal
000	0	100	4
001	1	101	5
010	2	110	6
011	3	111	7

On a $n = (011'100'101'010'001)_2$.

Plan

En machine, le nombre de bits p de la représentation positionnelle est fixé pour chaque *formats de codage* (entiers 8, 16, 32 ou 64 bits).

Si le résultat d'une opération requiert plus de p bits pour être représenté :

- le résultat calculé est formé des p bits de poids faibles du résultat exact,
- le drapeau de dépassement de capacité de l'ALU est levé.

Cela n'implique pas un arrêt des calculs, car le résultat peut être exploité.

Soit m un entier codé sur $q \geq p$ bits :

$$m = \sum_{i=0}^{q-1} m_i 2^i = \underbrace{\left(\sum_{i=p}^{q-1} m_i 2^{i-p} \right)}_{\text{quotient de } m \text{ par } 2^p} \times 2^p + \underbrace{\sum_{i=0}^{p-1} m_i 2^i}_{\text{reste}} .$$

Lorsqu'on effectue une opération arithmétique entre entiers naturels, et que l'on représente le résultat m sur p bits, le résultat obtenu est $m \mod 2^p$.

Plan

Représentation en complément à 2

Soit un entier relatif n, $-2^{p-1} \leq n \leq 2^{p-1}-1$, à coder en machine sur p bits.
La notation utilisée pour le codage de n en complément à 2 sur p bits est :

$$n = (c_{p-1}c_{p-2}\dots c_1 c_0)_{\overline{2}}.$$

On adopte la définition suivante :

$$(c_{p-1}c_{p-2}\dots c_1 c_0)_{\overline{2}} := -c_{p-1}2^{p-1} + \sum_{i=0}^{p-2} c_i 2^i.$$

On peut montrer que :

$$\begin{cases} n \geq 0 & \text{ssi} \quad c_{p-1} = 0, \\ n < 0 & \text{ssi} \quad c_{p-1} = 1. \end{cases}$$

Une manière d'interpréter le codage en complément à 2 est donc de considérer que le bit le plus à gauche a un poids négatif (-2^{p-1}).

Ex : par exemple, supposons qu'on effectue un codage sur 8 bits ($p = 8$).

- Quelle est la valeur décimale codée par $(10000011)_{\overline{2}}$?

Une manière d'interpréter le codage en complément à 2 est donc de considérer que le bit le plus à gauche a un poids négatif (-2^{p-1}).

Ex : par exemple, supposons qu'on effectue un codage sur 8 bits ($p = 8$).

- Quelle est la valeur décimale codée par $(10000011)_{\overline{2}}$?

$(10000011)_{\overline{2}} = -128 + 1 + 2 = (-125)_{10}$.

- Comment coder $(-120)_{10}$ en complément à 2 ?

Une manière d'interpréter le codage en complément à 2 est donc de considérer que le bit le plus à gauche a un poids négatif (-2^{p-1}).

Ex : par exemple, supposons qu'on effectue un codage sur 8 bits ($p = 8$).

• Quelle est la valeur décimale codée par $(10000011)_{\overline{2}}$?
$(10000011)_{\overline{2}} = -128 + 1 + 2 = (-125)_{10}$.

• Comment coder $(-120)_{10}$ en complément à 2 ?
$(-120)_{10} = -128 + 8$, donc $(-120)_{10} = (10001000)_{\overline{2}}$.

Addition et complément à 2

Soient $m = (c_m)_{\overline{2}}$ et $n = (c_n)_{\overline{2}}$ en complément à 2 sur p bits.

Addition en complément à 2

En l'absence de dépassement de capacité, le codage de $m + n$ en complément à 2 sur p bits est le codage de $(c_m + c_n) \mod 2^p$ en tant qu'entier naturel.

Dépassement de capacité, *i.e.* le résultat est incorrect :

- si m et n sont de signes opposés aucun dépassement n'est possible.
- si m et n sont de même signe, il y dépassement ssi le signe du résultat calculé diffère du signe des opérandes.

Opposé en complément à 2

En l'absence de dépassement de capacité, le codage de $-n$ en complément à 2 sur p bits est le même que celui de l'entier naturel

$$(\overline{c}_{p-1}\overline{c}_{p-2}\dots\overline{c}_1\overline{c}_0)_2 + 1 \mod 2^p.$$

Ex : $p = 7$, $(36)_{10} = (0100100)_{\overline{2}}$ a pour opposé $(-36)_{10} = (1011100)_{\overline{2}}$.
Notons que si on additionne $(1011100)_{\overline{2}}$ et $(0100100)_{\overline{2}}$, on obtient bien 0 :

$$\begin{array}{cccccccccc}
 & & \scriptstyle 1 & 1^{\scriptstyle 1} & 0^{\scriptstyle 1} & 1^{\scriptstyle 1} & 1^{\scriptstyle 1} & 1 & 0 & 0 \\
+ & & & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\hline
 & & \not{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hookrightarrow & & & & & & & & & 0
\end{array}$$

Plan

Les rationnels sont les nombres de la forme $\frac{p}{q}$, avec $p \in \mathbb{Z}$ et $q \in \mathbb{N} - \{0\}$.

On veut coder dans un format de longueur fixe des nombres dont l'écriture en binaire est potentiellement infinie : il faut se contenter d'une *approximation*.

Tout nombre rationnel $x \in \mathbb{Q}$ positif peut être décomposé en

- une partie entière $\lfloor x \rfloor \in \mathbb{N}$ telle que $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$;
- une partie fractionnaire $\{x\} = x - \lfloor x \rfloor$ avec $0 \leq \{x\} < 1$.

On utilise la notation positionnelle pour l'écriture de $\{x\}$: s'il existe $q \in \mathbb{N}$ t.q.

$$\{x\} = (0, x_{-1} \dots x_{-q})_\beta = \sum_{i=1}^{q} x_{-i} \beta^{-i},$$

alors $x = (x_{p-1} x_{p-2} \dots x_0, x_{-1} x_{-2} \dots x_{-q})_\beta$ est l'écriture x en base β.

Le problème est que l'écriture d'un rationnel en base β n'est pas forcément finie : par contre, on sait que cette écriture est nécessairement périodique.

Ex : en base $\beta = 10$, quelle est l'écriture de $13/7$?

Le problème est que l'écriture d'un rationnel en base β n'est pas forcément finie : par contre, on sait que cette écriture est nécessairement périodique.

Ex : en base $\beta = 10$, quelle est l'écriture de $13/7$?

1	3									7
	6	0								1, 8571428 5 . . .
		4	0							
			5	0						
				1	0					
					3	0				
						2	0			
							6	0		
								4	0	

Dans ce cas, on note $x = (1, \underline{857142})_{10}$, pour indiquer la période.

Changement de base : décimal vers binaire

Soit $0 \leq x < 1$ dont on connaît l'écriture en décimal. On veut déterminer son écriture en binaire :

$$x = (0, x_{-1} \dots x_{-q})_2.$$

On a $2 \times x = (x_{-1}, x_{-2} \dots x_{-q})_2$, donc $x_{-1} = \lfloor 2 \times x \rfloor$.

On procédant par des multiplications successives par 2, on peut ainsi extraire un par un les bits de l'écriture binaire de x.

Ex : convertir $1/10 = (0, 1)_{10}$ en écriture binaire.

Changement de base : décimal vers binaire

Soit $0 \leq x < 1$ dont on connaît l'écriture en décimal. On veut déterminer son écriture en binaire :

$$x = (0, x_{-1} \dots x_{-q})_2.$$

On a $2 \times x = (x_{-1}, x_{-2} \dots x_{-q})_2$, donc $x_{-1} = \lfloor 2 \times x \rfloor$.

On procédant par des multiplications successives par 2, on peut ainsi extraire un par un les bits de l'écriture binaire de x.

Ex : convertir $1/10 = (0,1)_{10}$ en écriture binaire.

$$\begin{array}{lllllll}
1/10 \times 2 & = 0 + 2/10 & & & & & \\
 & 2/10 \times 2 & = 0 + 4/10 & & & & \\
 & & 4/10 \times 2 & = 0 + 8/10 & & & \\
 & & & 8/10 \times 2 & = 1 + 6/10 & & \\
 & & & & 6/10 \times 2 & = 1 + 2/10 & \\
\hline
0, & 0 & 0 & 0 & 1 & 1 &
\end{array}$$

On en déduit que $(0,1)_{10} = (0, \underline{0011})_2$.

Changement de base : binaire vers décimal

On considère un rationnel $0 \leq x < 1$ dont on connaît l'écriture en binaire. On souhaite déterminer son écriture en décimale.

On peut utiliser la même méthode que précédemment :

- Il suffit d'effectuer des multiplications par $(10)_{10} = (1010)_2$, en calculant en binaire : on obtient les chiffres décimaux de l'écriture de x.
- Les calculs à la main sont fastidieux, mais une machine peut les mener efficacement.

Pour des nombres binaires comptant peu de bits après la virgule, il est suffisant de sommer le poids de ces bits.

$$2^{-1} = 0,5 \qquad 2^{-2} = 0,25 \qquad 2^{-3} = 0,125 \qquad 2^{-4} = 0,0625$$

Plan

Définition

On se concentre sur les nombres à virgule flottante binaires à p bits de précision, que nous appellerons *nombres flottants* ou *flottants*.

Un *nombre flottant normalisé* x est soit 0, soit un rationnel de la forme

$$x = (-1)^s \times (1, x_1 \dots x_{p-1})_2 \times 2^e,$$

- $s \in \{0, 1\}$ est le *signe* du flottant considéré,
- $(1, x_1 \dots x_{p-1})_2$ est sa *mantisse fractionnaire*,
- $e \in \mathbb{Z}$ est son *exposant*, t.q. $e_{\min} \leq e \leq e_{\max}$.

Rem :

- le 1 en tête garantit l'unicité de la représentation,
- souvent appelé « notation scientifique » en base 10,
- on ne parlera pas ici des flottants « sous-normaux ».

Ex : liste de tous les flottants normalisés positifs ou nuls avec 3 bits de précision, $e_{\text{min}} = -1$ et $e_{\text{max}} = 2$.

Ex : liste de tous les flottants normalisés positifs ou nuls avec 3 bits de précision, $e_{\min} = -1$ et $e_{\max} = 2$.

e	binaire	décimal
–	0	0
$e = -1$	$(1,00)_2 \times 2^{-1}$	$0,5$
	$(1,01)_2 \times 2^{-1}$	$0,625$
	$(1,10)_2 \times 2^{-1}$	$0,75$
	$(1,11)_2 \times 2^{-1}$	$0,875$
$e = 0$	$(1,00)_2 \times 2^{0}$	1
	$(1,01)_2 \times 2^{0}$	$1,25$
	$(1,10)_2 \times 2^{0}$	$1,5$
	$(1,11)_2 \times 2^{0}$	$1,75$
$e = 1$	$(1,00)_2 \times 2^{1}$	2
	$(1,01)_2 \times 2^{1}$	$2,5$
	$(1,10)_2 \times 2^{1}$	3
	$(1,11)_2 \times 2^{1}$	$3,5$
$e = 2$	$(1,00)_2 \times 2^{2}$	4
	$(1,01)_2 \times 2^{2}$	5
	$(1,10)_2 \times 2^{2}$	6
	$(1,11)_2 \times 2^{2}$	7

Ex : représenter $(0,1)_{10}$ par un flottant normalisé avec 8 bits de précision.

On a vu $(0,1)_{10} = (0,0\underline{0011})_2$. On commence par écrire :

$$(0,1)_{10} = (0,000\underbrace{11001100}_{8\text{ bits}}11\underline{0011})_2 = (\underbrace{1,1001100}_{8\text{ bits}}11\underline{0011})_2 \times 2^{-4}.$$

Donc $(0,1)_{10}$ est dans l'intervalle suivant, délimité par deux flottants consécutifs :

$$[\underbrace{(1,1001100)_2}_{8\text{ bits}} \times 2^{-4}; \underbrace{(1,1001101)_2}_{8\text{ bits}} \times 2^{-4}].$$

Le milieu de cet intervalle est $(\underbrace{1,1001100}_{8\text{ bits}}1)_2 \times 2^{-4}$.

Ici, on choisit d'arrondir au plus proche : comme $(0,1)_{10}$ est supérieur au point milieu, on choisit l'approximation $(0,1)_{10} \approx (\underbrace{1,1001101}_{8\text{ bits}})_2 \times 2^{-4}$.

Représentation en machine : norme IEEE-754

La norme IEEE-754 a été introduite en 1985, et a été révisée en 2008.

$c =$

s	code de l'exposant : c_e	code de la mantisse : c_m
1 bit	k bits	$p-1$ bits

Pour un flottant normal, la valeur codée est

$$f(c) = (-1)^{s(c)} \times m(c) \times 2^{e(c)},$$

avec

- $s(c) = s$ le signe du flottant.
- $m(c) = (1, c_m)_2$, ce qui signifie que le 1 de tête est codé implicitement.
- $e(c)$ dépend de $(c_e)_2$, mais doit être interprété avec précautions...

Codage des flottants en précision p

$c =$	s	code de l'exposant : c_e	code de la mantisse : c_m
	1 bit	k bits	$p-1$ bits

Comme c_e est un code sur k bits, notons que $0 \leq (c_e)_2 \leq 2^k - 1$.

- $(c_e)_2 = 0$ et $(c_m)_2 = 0$ indique que le nombre représenté est zéro.
  Il y a deux codages de 0 : -0 ou $+0$ selon s.
- $(c_e)_2 = 2^k - 1$ indique une valeur exceptionnelle (+Inf, -Inf, NaN) :
  Ex : $1/0 = +\text{Inf}$, $0/0 = \text{NaN}$, $a + \text{Inf} = \text{Inf}$, $+\text{Inf} - \text{Inf} = \text{NaN}$...
- $1 \leq (c_e)_2 \leq 2^k - 2$ indique que le nombre représenté est normal. Alors,

$$m(c) = (1, c_m)_2 \quad \text{et} \quad e(c) = (c_e)_2 - \underbrace{(2^{k-1} - 1)}_{\text{biais}}.$$

Formats standards

La norme IEEE-754 prévoit deux formats standards, la *simple précision* et la *double précision* (types `float` et `double` en `C`) :

format	mantisse		exposant				taille
	précision	$p-1$	k	biais	$e_{\min}$	$e_{\max}$	
simple	$p=24$	23	8	127	-126	127	32 bits
double	$p=53$	52	11	1023	-1022	1023	64 bits

Valeurs extrêmes pour les flottants normalisés :

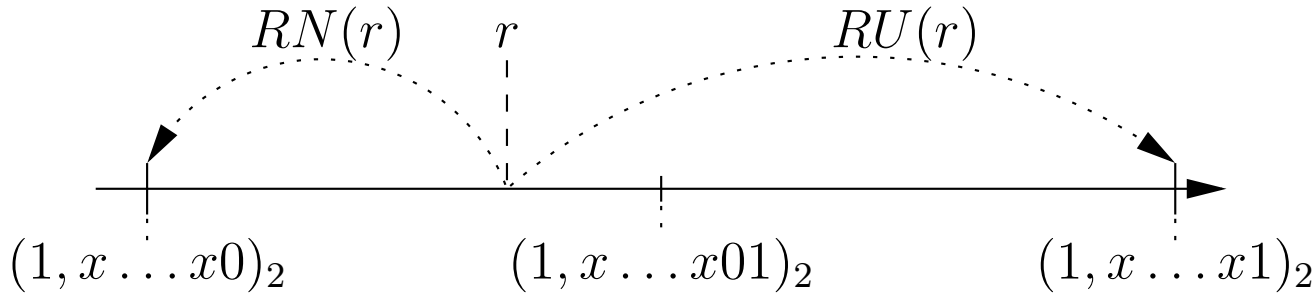
format	plus petit flt normalisé positif	plus grand flottant normalisé
simple	$2^{-126} \approx 1,2 \times 10^{-38}$	$(1-2^{-24}) \times 2^{128} \approx 3,4 \times 10^{38}$
double	$2^{-1022} \approx 2,2 \times 10^{-308}$	$(1-2^{-53}) \times 2^{-1023} \approx 1,8 \times 10^{308}$

Arrondi correct

Pour représenter un rationnel ou un réel r par un nombre flottant, il faut en général arrondir r. La norme IEEE-754 propose 4 modes d'arrondi :

- *arrondi au plus proche :* RN(r). Si r est équidistant de deux flottants consécutifs, on retourne celui dont la mantisse se termine par un 0.
- arrondi vers $-\infty$ (RD), $+\infty$ (RU), et vers 0 (RZ).

Ex :



La norme impose *l'arrondi correct* pour les opérations +, -, $\times$, / et $\sqrt{\cdot}$: le résultat calculé doit être le résultat exact arrondi selon le mode d'arrondi courant. L'arrondi au plus proche est en général le mode d'arrondi par défaut.

Un mot sur les erreurs d'arrondi

A chaque opération flottante, on peut avoir *une erreur d'arrondi*. De plus, les entrées d'un programme ne sont pas toujours représentées exactement.

Souvent, l'effet des erreurs d'arrondis est négligeable sur le résultat final d'une suite d'opérations ; mais leur effet peut parfois être gênant...

Ex : On compile le programme C suivant sur un Intel Core 2 Duo :

```
#include <stdio.h>
int main(void) {
  int i; float x = 0, y = 0.1;

  for(i=1; i<= 100000; i++) x += y;
  printf("Résultat : %8.3f\n", x);
  return(0);
}
```

Un mot sur les erreurs d'arrondi

A chaque opération flottante, on peut avoir *une erreur d'arrondi*. De plus, les entrées d'un programme ne sont pas toujours représentées exactement.

Souvent, l'effet des erreurs d'arrondis est négligeable sur le résultat final d'une suite d'opérations ; mais leur effet peut parfois être gênant...

Ex : On compile le programme C suivant sur un Intel Core 2 Duo :

```
#include <stdio.h>
int main(void) {
  int i; float x = 0, y = 0.1;

  for(i=1; i<= 100000; i++) x += y;
  printf("Résultat : %8.3f\n", x);
  return(0);
}
```

A l'exécution, on obtient :

```
Résultat : 9998.557
```

Plan

Il existe de nombreuses techniques de codage, chacune adaptée au codage d'un type d'information bien particulier. Nous avons vu des codages pour :

- les *entiers naturels* : représentation positionnelle ;
- les *entiers relatifs* : technique du complément à 2 ;
- les *nombres rationnels* : représentation positionnelle périodique ;
- les *nombres à virgule flottante* : norme IEEE-754.

Nous n'avons pas vu :

- les *caractères et le texte* : ASCII, Unicode ;
- des codages permettant la *détection et la correction d'erreurs* ;
- des codages pour la *compression* de données.

Dans la suite, nous donnerons une idée des techniques utilisées pour concevoir des circuits logiques permettant de manipuler ces codes en machine.