Types primitifs de données

Entiers

Flottants

Caractères

Représentation des nombres entiers

Principe

1. coder les valeurs à représenter sous forme binaire
2. stocker ces valeurs sous forme d'une séquence de bits

   dans une ou plusieurs cellules de la mémoire.

Ecriture en base 2

Chaque chiffre du nombre multiplie une puissance de 2

1 1 0 1 0 1

$$\mathbf{1} \times 2^5 + \mathbf{1} \times 2^4 + \mathbf{0} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0$$

= 53 en décimal

Nombres particuliers

100……00 (1 suivi de **n zéros)**

$= \mathbf{2^n}$

11……11 (n fois le chiffre 1)

$= \mathbf{2^{n-1} + 2^{n-2} + 2^{n-3}} + \ldots + \mathbf{2^2 + 2^1 + 2^0}$

$= \mathbf{2^n - 1}$

Transformer du décimal en binaire :

- diviser le nombre par 2
- écrire le reste de la division à gauche du chiffre précédent
- recommencer jusqu'à ce que le nombre soit nul
-

43 : 2 = 21 reste 1	1
21 : 2 = 10 reste 1	11
10 : 2 = 5 reste 0	011
5 : 2 = 2 reste 1	1011
2 : 2 = 1 reste 0	01011
1 : 2 = 0 reste 1	**101011**

Mais pourquoi utiliser le binaire ?

Stockage et transmission : il suffit d'un mécanisme physique à deux états, beaucoup moins cher qu'un mécanisme à 10 états.

Calcul : les tables sont simples => circuits électroniques simples

Table d'**addition** binaire :

```
    0        0        1        1
  + 0      + 1      + 0      + 1
  ___      ___      ___      ___
    0        1        1      1 0 ( 1 + 1 = 2)
```

Table de **multiplication** binaire :

```
    0        0        1        1
  × 0      × 1      × 0      × 1
  ___      ___      ___      ___
    0        0        0        1
```


Nombres entiers de 0 à $2^n–1$

Une cellule de **n bits** peut représenter les nombres de

de **00……00** à **11……11** (n chiffres)

c'est à dire

de **0** à $\mathbf{2^n - 1}$

P.ex. dans un cellule de **8 bits** on peut représenter les entiers positifs

de **0** = [00000000] à **255** = [11111111]

Pour stocker un nombre plus grand ou égal à 2^n

on utilise un nombre suffisant de cellules adjacentes.

Exemple: nombres de 0 à 2'000'000'000 => 4 cellules de 8 bits.

Entiers relatifs (nombres négatifs)

Technique du complément à 2.

Pour représenter **–X** avec **n bits** on utilisera le nombre $\mathbf{2^n - X}$.

Exemple (n = 8), pour représenter **–6** on fait

	(b8)	b7	b6	b5	b4	b3	b2	b1	b0	
	1	0	0	0	0	0	0	0	0	$\mathbf{2^8}$
–		0	0	0	0	0	1	1	0	**6**
=	0	1	1	1	1	1	0	1	0	**– 6**

Avec **n bits** on représente les entiers relatifs

de $\mathbf{-2^{n-1}}$ à $\mathbf{2^{n-1} - 1}$.

Propriété du complément à 2

- Il n'y a qu'un zéro
- X + –X = 0

	(b8)	b7	b6	b5	b4	b3	b2	b1	b0	
		0	0	0	1	0	1	1	0	**22**
+		1	1	1	0	1	0	1	0	**–22**
=	1	0	0	0	0	0	0	0	0	**0**

Trouver la représentations de –X en binaire
- calculer Y = X – 1
- calculer la représentation binaire de Y
- remplacer les 0 par des 1 et les 1 par des 0

Opérations sur les entiers à n bits

Représentation des entiers limitée

=> la définition standard des opérations +, –, × ne fonctionne pas

Exemple

si n = 7

les entiers vont de $-128 = -2^7$ à $2^7 - 1 = 127$,

que vaut 127 + 3 ?

que vaut –100 – 55 ?

Les entiers informatiques ne sont pas les entiers mathématiques

{entier machine} $\subset$ **Z**

Arithmétique modulaire

Principe : prendre le reste de la division par 2^n après chaque opération.

== on ne considère que les n premiers bits du résultat.

Exemple

calcul de 127 + 3 :

	(b8)	b7	b6	b5	b4	b3	b2	b1	b0
		0	1	1	1	1	1	1	1
+		0	0	0	0	0	0	1	1
=		1	0	0	0	0	0	1	0

donc

127 + 3 = –126.

Exemple (2)

–2 + 3 :

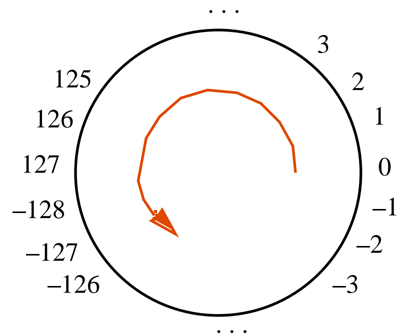
	(b8)	b7	b6	b5	b4	b3	b2	b1	b0
		1	1	1	1	1	1	1	0
+		0	0	0	0	0	0	1	1
=	1	0	0	0	0	0	0	0	1

donc

–2 + 3 = 1

Cercle des nombres

Tout se passe comme si les nombres étaient arrangés sur un cercle :



127 + 1 = −128

Multiplication

La multiplication est également modulaire

Sur les entiers à n bits

$a \times b = ab$ modulo 2^n

Exemples sur 4 bits

-2×-2

$= 1110 \times 1110 = 11000100$

= 0100 (on garde les 4 derniers bits) = 4

$3 \times -1 = 0011 \times 1111 = 101101 = 1101 \pmod{2^4} = -3$

Multiplication par une puissance de 2

2^k = 100……00 (k zéros)

donc $2^k \times m$ = m00……00 (ajouter k zéros à gauche → facile à calculer)

Division et reste

Division entière

p / q = le plus grand entier r tel que

$$rq \le p < (r+1)q$$

p mod q = l'entier m tel que

$$(p / q) \times q + m = p$$

La division par 0 n'est pas définie.

Division par une puissance de 2

Diviser par 2 revient à enlever le chiffre le plus à gauche.

Diviser par 2^k : enlever les k chiffres les plus à gauche

Les modèles courants de nombres entiers proposés par les processeurs

Ces modèles utilisent en général un multiple de 8 bits, on trouve:

le byte: entier de 8 bits, de -128 à +127

le mot: entier de 16 bits, de -32768 à +32767

le double mot: entier de 32 bits, de -2^{31} à $2^{31}-1$

le quadruple mot: entier de 64 bits, de -2^{63} à $2^{63}-1$

Conversions

dans un sens pas de problème (petit -> grand)

dans l'autre sens: s'assurer que le nombre à convertir est suffisamment petit

Les modèles proposés par les langages de programmation

1. Modèle d'entiers = celui du **processeur** utilisé: **C, Pascal, C++**, etc.

   => le même programme a des comportement différents suivant les machines !!!

2. Un langage comme **Ada** permet de **définir des types d'entiers**

   p.ex. **type** MesNombres **is** –2 .. +3000

3. Le langage **Java** propose des types standards **indépendants du processeur**:

- int : entier de 32 bits
- long : entier de 64 bits
- byte : entier de 8 bits
- short : entier de 16 bits

=> même comportement numérique sur toutes les machines

! conversions = troncations (= modulo) => effets inattendus

Nombres entiers illimités

Technique : attribuer à chaque nombre une quantité de mémoire suffisante

sous forme de cellules contigües.

[N, cellule1, …, celluleN]

=> pas de soucis de débordement (jusqu'à la taille de la mémoire)

Mais il faut gérer la taille variable des nombres

les processeurs ne sont pas prévus pour traiter ce type de représentation.

=> écrire des procédures spécifiques qui utilisent l'arithmétique modulaire du processeur

Modèle d'entiers des langages **Python**, **Smalltalk**, **Maple**, **Mathematica, ...**

Types disponibles en **Java**, **C++**, etc.

Quelques algorithmes

Tester la divisibilité

X est divisible par Y s'il existe un entier Q tel que X = Q * Y.

Donc si le reste de la divison de X par Y est nul.

Il suffit donc d'appliquer l'opération **mod (% en Java)**

si (X **mod** Y = 0) Resultat = "divisible" **sinon** Resultat = "non divisible"

En Java :

if (X **%** Y == 0) Resultat = "divisible"; **else** Resultat = "non divisible";

Encodage / Décodage

But: représenter plusieurs valeurs avec un seul nombre entier.

Exemple: une couleur = r % de rouge, v % de vert, b % de bleu

On peut utiliser trois variables R, V, B pour représenter une couleur

Mais on peut faire plus compact :

Encodage :

couleur ← r * 10000 + v * 100 + b

Décodage :

bleu ← couleur mod 100

vert ← (couleur / 100) mod 100

rouge ← couleur / 10000

Extraction des chiffres en base b

Tout nombre X peut se décomposer de manière unique en

$X = c_0 + c_1 b + c_2 b^2 + c_3 b^3 + \ldots + c_n b^n \ (c_i < b)$

Les c_i sont les chiffres qui expriment X en base b

8912 = 2 + 1 * 10 + 9 * 100 + 8 * 1000

8 9 1 2 en base 10

8912 = 0 + 2 * 8 + 3 * 64 + 1 * 512 + 2 * 4096

2 1 3 2 0 en base 8

8912 = 0 + 13 * 16 + 2 * 256 + 2 * 4096

2 2 D 0 en base 16 (D = 13)

Algorithme

```
i <-- 0
tant que x > 0 {
        c_i <-- x mod b
        i <-- i + 1
        x <-- x / b
}
```


Exemple: ISBN

Le dernier chiffre d'un numéro ISBN (International Standard Book Number) est un chiffre de contrôle.

no: 0 - 201 - 54991 - 3

3 = 11 - (0 * 10 + 2 * 9 + 0 * 8 + 1 * 7 + 5 * 6 + 4 * 5 + 9 * 4 + 9 * 3 + 1 * 2) mod 11

But : vérifier s'il n'y a pas eu de fautes de frappe

Méthode : recalculer le dernier chiffre d'après la formule et voir si on trouve le même

Algorithme ISBN

```
mul <-- 2; x <-- isbn; controle <-- 0
tant que x > 0 {
        chiffre <-- x mod 10
        controle <-- controle + mul * chiffre
        mul <-- mul + 1
        x <-- x / 10
}
controle = 11 - controle mod 11
```

(10 est remplacé par 'X' à l'impression)

Plus grand commun diviseur de X et Y

On cherche le plus grand nombre D tel que D est un diviseur de X et de Y

Algorithme "naïf"

essayer successivement D = 1, D = 2, D = 3, ...

et se souvenir de la dernière valeur de D telle que X mod D = 0 et Y mod D = 0

on peut arrêter dès que D est plus grand que X ou Y

```
D ← 1;
PGCD ← D;
tant que (D ≤ X et D ≤ Y) {
        si (X mod D = 0 et Y mod D = 0) PGCD ← D;
        D ← D+1
}
```

Peut-on faire mieux (plus rapide, c-à-d moins d'opérations) ?