PF1 — Principes de Fonctionnement des machines binaires

Jean.Baptiste.Yunes@univ-paris-diderot.fr

Version 1.13

Les nombres en machine

On ne manipule que rarement de très très très grands nombres ou des nombres avec une précision arbitrairement grande

des choix sont faits pour faciliter l'utilisation des nombres courants

leur représentation est de taille fixe, c'est un avantage…

Sur les ordinateurs on utilise la base 2

l'électronique s'y prête bien…

$2^{32} \approx 4.10^{9}$ (4 milliards)

$2^{64} \approx 18.10^{18}$ (18 trillions)

pour info

10^{80} atomes dans l'univers

environ 10^{23} grains de sable sur terre,

le nombre d'Avogadro est environ 10^{23}

environ 10^{13} cigarettes fumées chaque année dans le monde

environ 10^{15} µ-secondes par siècle

environ $\$10^{13}$ de masse monétaire

PIB mondial $\approx \$72.10^{12}$

Aujourd'hui les machines sont couramment à architecture 32 ou 64 bits

Cela signifie, entre autres, que leur arithmétique interne opère ordinairement sur des nombres représentés sur 32 ou 64 bits (respectivement).

Étudions l'arithmétique sur n bits (où n est fixé)…

Si on prend 32 bits on peut écrire 2^{32} mots binaires différents (~4 milliards)

il suffit de choisir quels nombres ces 2^{32} mots représentent

insistons sur le fait que le **choix** peut être aussi **arbitraire** que l'on souhaite…

en pratique on **essaie** d'établir une **correspondance pratique** entre les mots binaires et la représentation en base 2 des nombres (inutile de trop se compliquer la vie)

ordre du dictionnaire avec 0<1

Mots sur {0,1}	$(\text{Nombre})_2$	$(\text{Nombre})_{10}$
00000000000000000000000000000000	0	0
00000000000000000000000000000001	1	1
00000000000000000000000000000010	10	2
00000000000000000000000000000011	11	3
…	…	…
11111111111111111111111111111110		4294967294
11111111111111111111111111111111		4294967295

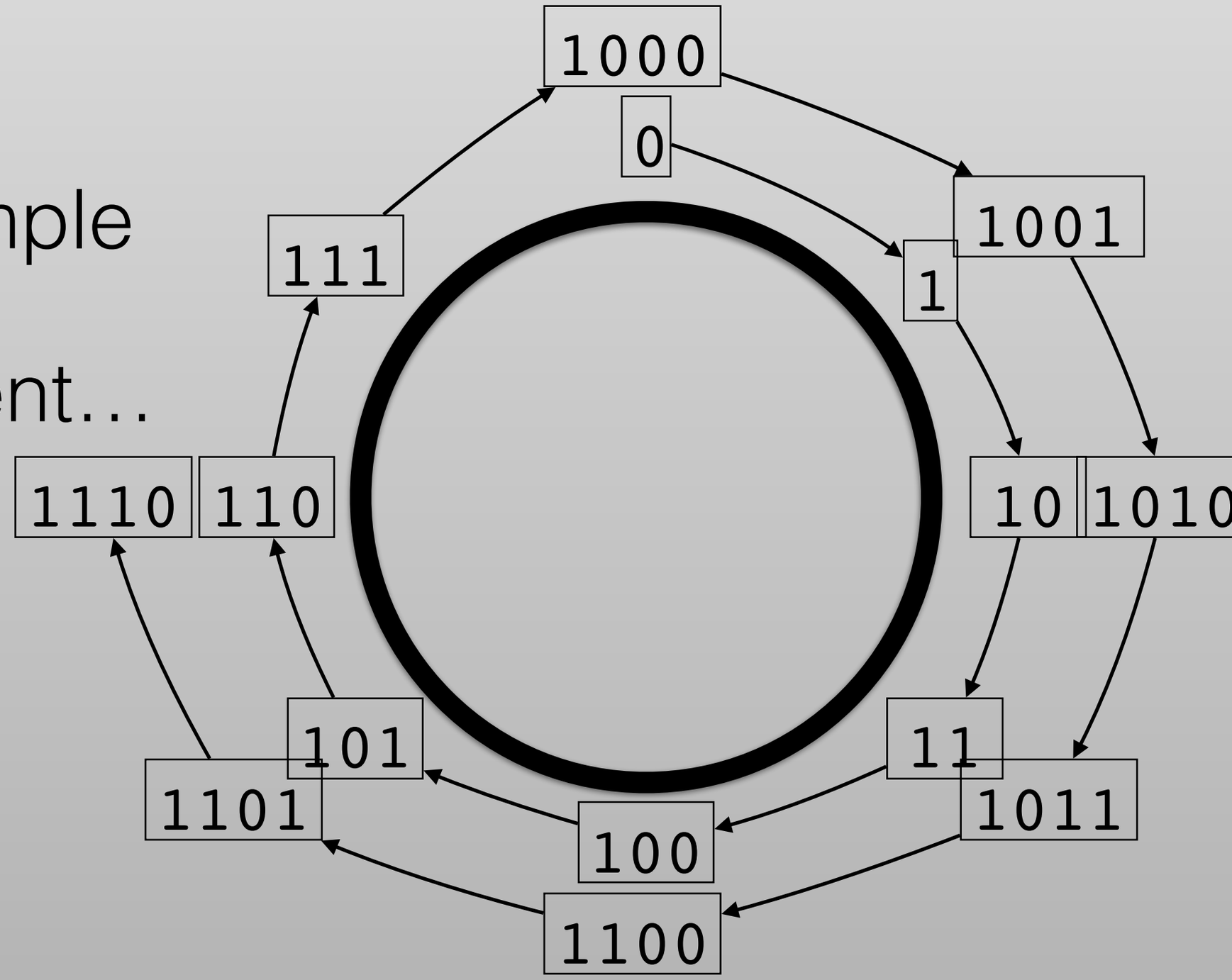
Cette représentation est celle des entiers non signés…

Le langage C autorise la définition de variables entières non signées, par exemple **`unsigned int`**

Une arithmétique possible est l'arithmétique modulaire

ici modulo 8
pour l'exemple

3 bits suffisent…



Qu'est ce que l'**arithmétique modulaire** ?

C'est l'arithmétique dans laquelle on ne s'intéresse pas aux nombres eux-mêmes mais uniquement à leurs restes par la division euclidienne

dans l'arithmétique modulo p (pour p fixé), les nombres i+kp (avec $0 \leq i < p$) sont tous équivalents.

ex: $13 \equiv 5 \pmod 8$

On en a l’habitude avec les horloges…

pour les heures on compte modulo 12

combien a t-on mesuré ? 1:53 ? 13:53 ?

25:53 ?

Idem avec les odomètres et autres compteurs variés

L’arithmétique modulaire (ex.: mod 2^3)

Certaines opérations provoquent un échappement de retenue ou l’entrée d’une retenue

111+010 donne 001 retenue 1 (qui est oubliée)

7+2 = 1 (mod 8)

000 - 010 donne 110 retenue 1 (que l’on peut oublier)

0-2 = 6 (mod 8)

Une retenue représente le « passage d’un tour »

Si on veut prendre des valeurs négatives le problème se complique ?

ok un bit pour le signe (disons 0 pour + et 1 pour -) et 31 bits pour la valeur absolue…

$$\mathbf{s}c_{30}c_{29}\ldots c_1c_0$$

C’est la représentation en **signe et magnitude**

11111111111111111111111111111111	$-2^{31}-1$
11111111111111111111111111111110	$-2^{31}-2$
11111111111111111111111111111101	$-2^{31}-3$
…	
10000000000000000000000000000011	-3
10000000000000000000000000000010	-2
10000000000000000000000000000001	-1
10000000000000000000000000000000	- 0
00000000000000000000000000000000	+0
00000000000000000000000000000001	+1
00000000000000000000000000000010	+2
00000000000000000000000000000011	+3
…	
01111111111111111111111111111110	$+2^{31}-2$
01111111111111111111111111111111	$+2^{31}-1$

problème : il y a deux zéros -0 et +0

pas grave en soi, mais pour les opérations arithmétiques c'est embêtant…

autre problème : le calcul du successeur (+1) pour les négatifs n'est pas le même que pour les positifs…

l'arithmétique est plus compliquée

Pour additionner il faut tenir compte des signes et faire des opérations différentes

0000 + 1 = 0001 (0+1=1)

1010 + 1 = 1001 (-2+1=-1) il faut soustraire 1!!!

0010 + 1010 = 0 (2+-2=0) est en fait une soustraction!

Trop complexe, pas assez uniforme

la machinerie électronique en serait d'autant compliquée…

Une autre façon de voir (la bonne ? celle utilisée fréquemment!)

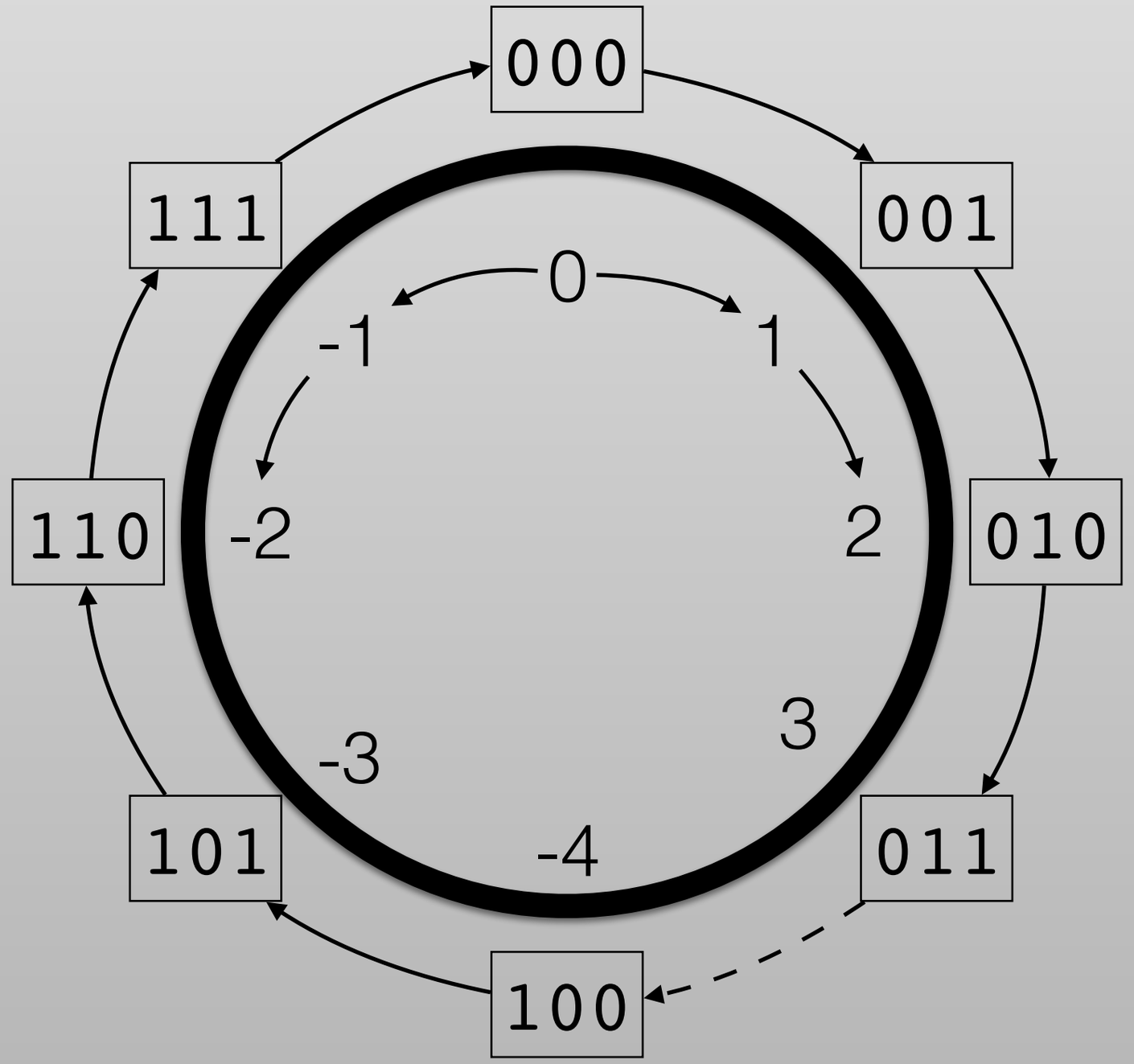
partir de la représentation *naturelle* du 0 et ajouter 1 ou retrancher 1

C’est la représentation en complément à 2^n

entiers relatifs $[-2^{31},2^{31}[$

10000000000000000000000000000000	-2^{31}
10000000000000000000000000000001	$-2^{31}+1$
10000000000000000000000000000010	$-2^{31}+2$
10000000000000000000000000000011	$-2^{31}+3$
…	
11111111111111111111111111111101	-3
11111111111111111111111111111110	-2
11111111111111111111111111111111	-1
00000000000000000000000000000000	0
00000000000000000000000000000001	1
00000000000000000000000000000010	2
00000000000000000000000000000011	3
…	
01111111111111111111111111111110	$2^{31}-2$
01111111111111111111111111111111	$2^{31}-1$

L’arithmétique obtenue est modulaire



Tout cela est tout à fait ordinaire…

2 heures **moins** 10
ten **to** two

8 heures **(et)** 20
twenty **past** eight

le signe est aussi indiqué par le bit à gauche!

un seul 0 (qui est positif donc)

l’arithmétique est plus uniforme
et donc la machinerie électronique

cette représentation est dite en complément à deux (en fait complément à 2^{32}, on passe sous silence la longueur des mots)

car pour $0 \leq n < 2^{31}$, -n est représenté par le codage de 2^{32}-n (le complément à 2^{32})

Si le complément a 2 paraît bizarre de prime abord les opérations arithmétiques, elles, sont assez simples

Il suffit de procéder normalement!

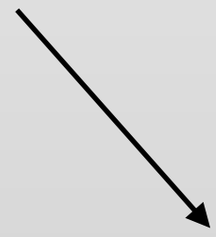
On l'a déjà vu, ce sont les nombres avec une infinité de 0 ou 1 à gauche. Il suffit de les tronquer…

Addition en complément à deux

on utilisera le complément à 2^{16} pour éviter de trop fastidieux calculs

7823 + 273 ?

retenues



```
          111111
  0001111010001111
+ 0000000100010001
------------------
  0001111110100000
```

8096

normal, des 0 à gauche ne changent rien!

retenue sortante…

retenues

-7823 + -273 ?

```
 1111111111111111
  1110000101110001
+ 1111111011101111
------------------
 1110000001100000
```

-8096

ça marche!

normal, les 1 à gauche (le signe) sont conservés…

retenue sortante…

retenues

7823 + -273 ?

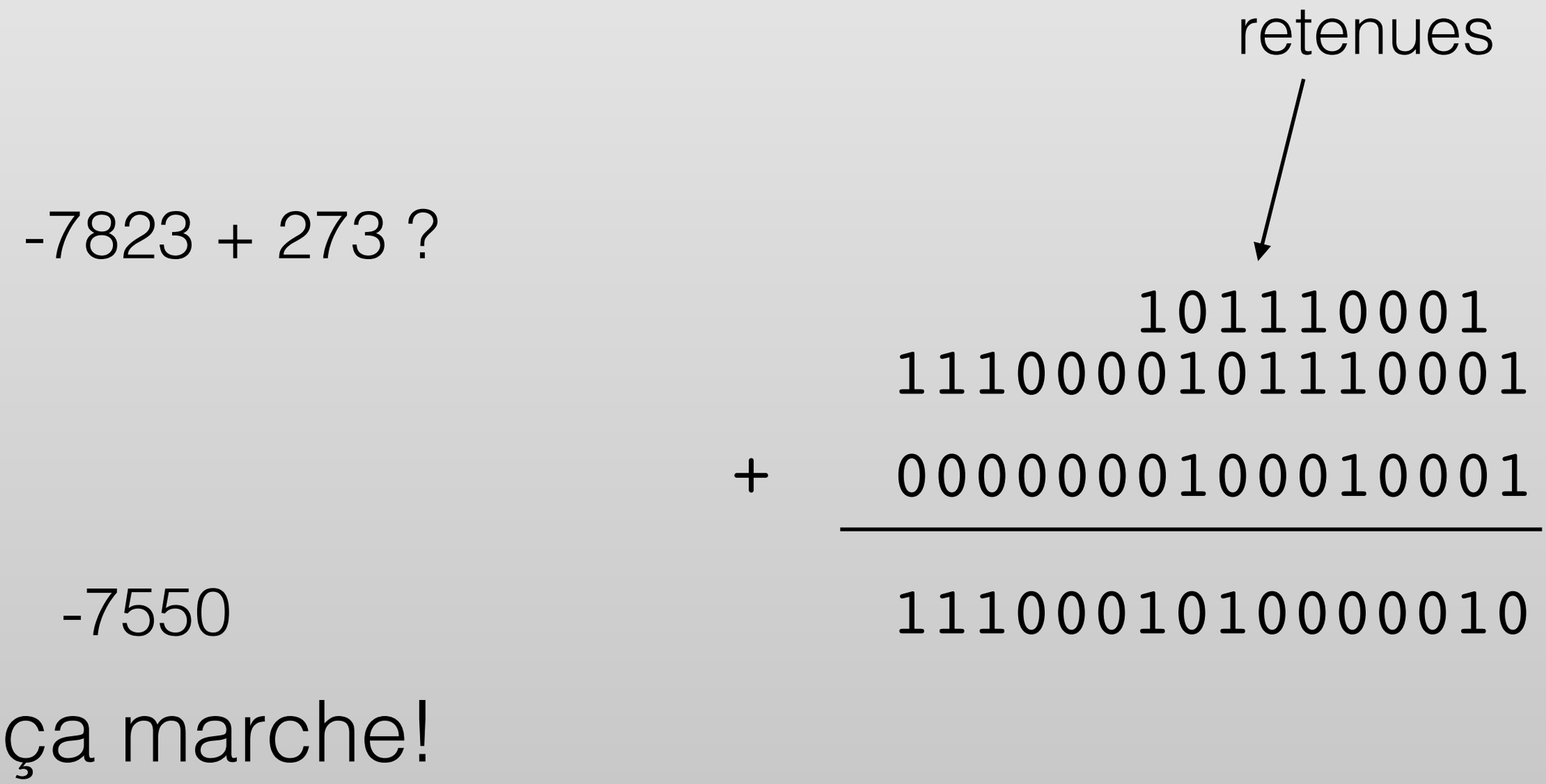
```
  1111111101000111
   0001111010001111
+    1111111011101111
  ------------------
   0001110101111110
```

7550

ça marche!

-7823 + 273 ?

-7550



ça marche!

À quelles conditions cette opération d'addition marche t-elle ?

quand la somme n'est pas représentable…

calculons $(2^{15}-1)+1$

```
  111111111111111
  011111111111111
+ 000000000000001
-----------------
  100000000000000
```

-2^{16}?

c'est l'arithmétique modulaire…

une heure trente + dix minutes = deux heures moins vingt

Regardons en Java, pour voir si le problème apparaît…

```
public class T {
  public static void main(String []args) {
    int a = 0x7FFFFFFF;
    int b = 1;
    System.out.println(a+"+"+b+"="+(a+b));
  }
}
```

```
yunes% java T
2147483647+1=-2147483648
yunes%
```

Soustraction en complément à deux

on utilisera le complément à 2^5 pour éviter la lourdeur des calculs

7 - 4 ?

```
  00111
- 00100
-------
  00011
```

ça marche!

7 + -4

```
  00111
+ 11100
-------
  00011
```

4 - 7 ?

```
  00100
- 00111
-------
  11101
```

ça marche!

4 + -7

```
  00100
+ 11001
-------
  11101
```

7 - -4 ?

```
    00111
-   11100
    11
-------------
    01011
```

ça marche!

7 + 4

```
    00111
+   00100
-------------
    01011
```

4 - -7 ?

```
  00100
- 11001
-------
  01011
```

ça marche!

Détection d’erreurs

Comment savoir si une opération a fourni un résultat correct ?

On est dans l’arithmétique modulaire, les grands nombres sont « réduits » à leur reste modulo…

Cela dépend si l’on fait de l’arithmétique signée ou non…

Pour l'arithmétique non signée il suffit d'**observer la retenue** qui « sort »

pour l'addition c'est la retenue sortante

pour la soustraction c'est la retenue entrante

Si la retenue en question est 1, il y a une erreur!

Sinon l'opération est correcte

```
   01000    8          01000    8
 - 11001   25        - 11000   24
  11111   ret          10000   ret
-------------------------------------
   01111   15          10000   16
```

```
   11000   24          11000   24
 - 01010   10        - 01000    8
   1110   ret          0000   ret
-------------------------------------
   01110   14          10000   16
```

Pour l'arithmétique signée il suffit d'observer les signes des opérandes et du résultat

pour l'addition additionner deux nombres positifs ne peut pas donner un nombre négatif et additionner deux négatifs ne peut donner un positif

pour la soustraction (x-y=x+(-y)) soustraire un positif à un négatif ne peut donner un positif et soustraire un négatif à un positif ne peut donner un négatif!

Il suffit donc d'observer les signes…

```
  01000    8          01000    8
- 11001   -7        - 11000   -8
  11111   ret         10000   ret
---------------    ---------------
  01111   -1          10000  -16
```

```
  11000   -8          11000   -8
- 01010   10        - 01000    8
  01110   ret         00000   ret
---------------    ---------------
  01110   14          10000  -16
```

Types des nombres…

Chaque langage définit des **types** entiers permettant de réaliser des calculs arithmétiques

Qu’est-ce que c’est **un type** ?

C : `char`, `short`, `int`, `long`, `long long`, `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`

Java : `byte`, `short`, `int`, `long`, `char`

Le type d'une variable permet de fixer

sa **taille** en mémoire

l'**interprétation** du mot binaire correspondant

l'ensemble des **opérations** légales et leur sémantique

La bonne maîtrise d'un langage suppose la bonne maîtrise des types de base

Attention, c'est souvent négligé…à tort!

On étudiera ces types petit à petit, ils sont plus complexes qu'on ne croit généralement

Exemple pour Java, le type `int` (entiers ordinaires)

utilise des mots de 32 bits (chiffres binaires)

une représentation en complément à 2

permet l'addition, soustraction, multiplication, etc

```
int x = 367898;
int y = -48747799;
System.out.println(x);
System.out.println(y);

int z = x+y;
System.out.println(z);

z = x-y;
System.out.println(z);

z = x*y;
System.out.println(z); // résultat faux

x++; // rajoute 1 à x, éq. x = x + 1;
System.out.println(x);
```

Attention! Si les machines détectent en interne les erreurs arithmétiques, la plupart des langages de programmation ne répercutent pas celles-ci à l'utilisateur…

Il faut être **conscient** de ces problèmes

Difficile!

Source fréquente de bogues…

```
int joe      = 2000000000;
int jack     = 2000000000;
int william = 2000000000;
int averell =  250000000;

System.out.println("Ils ont gagné "+
            (joe+jack+william+averell));

System.out.println("Soit en moyenne : "+
           (joe+jack+william+averell)/4);
```

Exemple pour Java, le type **`byte`** (petits entiers)

utilise des mots de 8 bits (chiffres binaires)

une représentation en complément à 2

permet l'addition, soustraction, multiplication, etc

```
byte x = (byte)0xE2;

x++;
System.out.println(x);

x++;
System.out.println(x);

x++;
System.out.println(x);

x++;
System.out.println(x);
```

Les types entiers de Java

	taille en octets	taille en bits	représentation	+petite valeur	+grande valeur
byte	1	8	comp. 2	-128	+127
short	2	16	comp. 2	-32768	+32767
int	4	32	comp. 2	-2147483648	+2147483647
long	8	64	comp. 2	-9223372036854775808	+9223372036854775807