Ch. 2
Représentation de nombres naturels et entiers

B. Quoitin
(bruno.quoitin@umons.ac.be)

Table des Matières

Notation des Nombres

Nombres dans un Ordinateur

Représentation des entiers

Objectifs

- Formaliser la manière dont nous représentons les nombres en utilisant la **notation décimale** (une représentation apprise à l'école primaire).
- Se baser sur le formalisme de la notation décimale positionnelle pour découvrir des représentations non-décimales de nombres → **notation positionnelle généralisée**.
- Les représentations non-décimales **utiles** en informatique sont le binaire, l'octal et l'hexadécimal.
- Apprendre comment effectuer des **conversions** entre les différentes représentations.

Représentation décimale

Chiffres/symboles

- La représentation décimale se base sur l'utilisation de 10 symboles, les chiffres décimaux : “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8” et “9”. A chaque chiffre correspond un nombre entre 0 et 9. Ainsi,
  - le chiffre « 0 » représente le nombre 0,
  - le chiffre « 1 » représente le nombre 1,
  - et ainsi de suite.
- Attention, un chiffre est un *symbole* permettant de représenter un nombre. Il est important de réaliser que **chiffre** (*représentation*) et **nombre** (*représenté*) sont des concepts différents.

Séquence et position

- Pour représenter des nombres plus grands que 9, une séquence de chiffres est utilisée. La position d’un chiffre dans cette séquence est importante car elle donne un poids à ce chiffre. Il s’agit d’une représentation dit « **positionnelle** ».

Représentation décimale

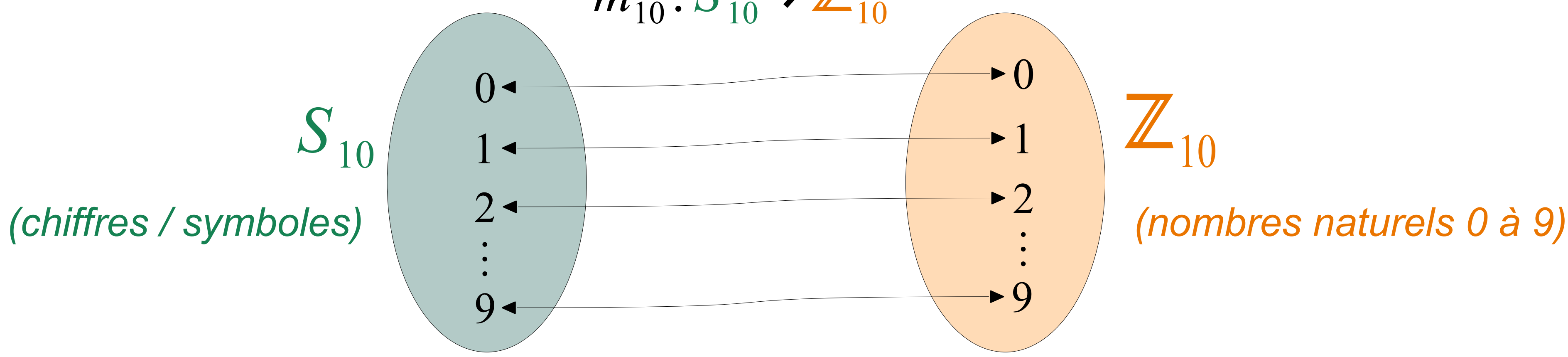
Tentative de formalisation

- Plus formellement, la représentation décimale utilise un ensemble de 10 chiffres

$$S_{10}=\{0,1,2,3,4,5,6,7,8,9\}$$

- Une bijection m_{10} fait correspondre chaque chiffre de S_{10} à un unique nombre naturel compris entre 0 et 9 inclus.

$$m_{10} : S_{10} \rightarrow \mathbb{Z}_{10}$$



Note : en représentation décimale, nous confondrons souvent les chiffres décimaux avec les nombres qu'ils représentent lorsqu'il n'y a pas d'ambiguïté.

Tentative de formalisation

- Un nombre naturel x est représenté par un **mot** w composé de N chiffres w_i pris dans S_{10}

$$w = w_{N-1} w_{N-2} \dots w_1 w_0$$

- Interpréter une représentation w consiste à déterminer le nombre x représenté. On utilise à cet effet la formule suivante. Le chiffre w_i de position i est associé au **poids** 10^i où 10 est la **base** du système.

$$x = \sum_{i=0}^{N-1} m_{10}(w_i) \cdot 10^i$$

- Exemple : que représente w = « 32768 » ($w_0 = 8$, $w_1 = 6$, …, $w_4 = 3$) ?

$$\begin{aligned} x &= \sum_{i=0}^{N-1} w_i \cdot 10^i \\ &= m_{10}(3) \cdot 10^4 + m_{10}(2) \cdot 10^3 + m_{10}(7) \cdot 10^2 + m_{10}(6) \cdot 10^1 + m_{10}(8) \cdot 10^0 \\ &= 3 \cdot 10^4 + 2 \cdot 10^3 + 7 \cdot 10^2 + 6 \cdot 10 + 8 \;=\; 32768 \end{aligned}$$

Poids des chiffres

- Chaque chiffre d'un mot w est associé à un poids en fonction de sa position.
  - le chiffre le plus à gauche est appelé **chiffre de poids (le plus) fort**.
  - le chiffre le plus à droite est appelé **chiffre de poids (le plus) faible**.

$$\boxed{w = w_{N-1} w_{N-2} \ldots w_1 w_0}$$

$\longleftarrow$ poids fort $\quad$ poids faible $\longrightarrow$

Chiffres significatifs

- Les chiffres significatifs d'une représentation sont les chiffres qui suivent le premier chiffre différent de 0, y compris ce dernier. Le nombre représenté ne change pas si on supprime les chiffres non significatifs.
- Exemple : dans 004302, il y a 4 chiffres significatifs, 4302.

Représentation décimale

Nombre de mots différents

- Il existe 10^N mots de N chiffres. Il est donc possible avec N chiffres de représenter 10^N nombres différents.

$$\boxed{w = w_{N-1} w_{N-2} \ldots w_1 w_0}$$

(10 possibilités pour w_{N-1})* ... * (10 possibilités pour w_1) * (10 possibilités pour w_0)

Intervalle représentable

- L'intervalle de nombres représentables avec N chiffres est égal à

$$\boxed{[0, 10^N - 1]}$$

Représentation décimale

Taille de mot

- Etant donné un nombre naturel x, quelle doit être la valeur minimale de N pour pouvoir représenter x ?
- Pour que x soit représentable sur N bits, il faut qu'il soit inférieur ou égal au plus grand nombre représentable

$$\begin{aligned} x &\leq 10^N - 1 \\ x+1 &\leq 10^N \\ \log_{10}(x+1) &\leq \log_{10}(10^N) = N \end{aligned}$$

- Le plus petit nombre de chiffres nécessaires pour représenter x est donc obtenu comme

$$\boxed{N \geq \lceil \log_{10}(x+1) \rceil}$$

- <u>Exemple</u> : combien de chiffres significatifs sont nécessaires pour représenter le nombre $x = 1234$?

Table des Matières

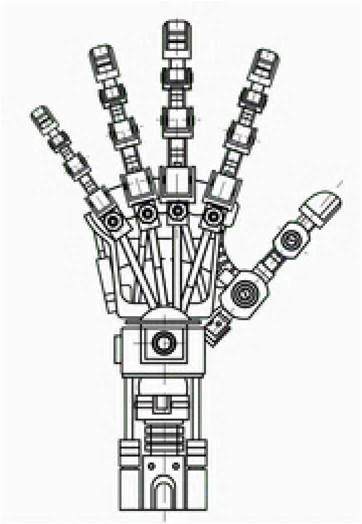
Notation des Nombres

- Représentation décimale
- Notation Positionnelle Généralisée
- Changement de Base

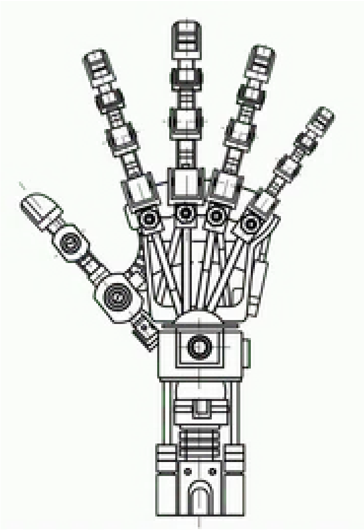
Nombres dans un Ordinateur

- Représentation des naturels
- Représentation des entiers

Notation Positionnelle Généralisée



Les ordinateurs n'ont pas dix doigts !



Objectif

- Nous sommes familiarisés avec la notation décimale (base 10). Les ordinateurs utilisent la notation binaire (base 2).
- Deux symboles/chiffres, notés 0 et 1, qui peuvent être implémentés par <u>2 états d'un système électronique</u>.
- La <u>notation positionnelle généralisée</u> permet de représenter des nombres en utilisant d'autres bases.

Notation Positionnelle Généralisée

Généralisation

- Soit $k \geq 2$ la base utilisée.
- Soit S_k un ensemble de k symboles différents. $\boxed{S_k = \{ s_0, \ldots, s_{k-1} \}}$
- Soit m_k une bijection qui fait correspondre chaque symbole de S_k à un nombre compris entre 0 et k-1.

$$\boxed{m_k : S_k \rightarrow \mathbb{Z}_k : s_k \rightarrow k}$$

- Un nombre naturel x est représenté par un mot w de N symboles pris dans S_k.

$$\boxed{w = w_{N-1} w_{N-2} \ldots w_1 w_0 \quad (\forall i, w_i \in S_k)}$$

- Le nombre x représenté par la séquence w est obtenu par la formule généralisée

$$\boxed{x = \sum_{i=0}^{N-1} m_k(w_i) \cdot k^i}$$

Notation Positionnelle Généralisée

Exemples

- $k = 10$, chiffres décimaux
  - Identique à la notation positionnelle décimale.
  - Ensemble des symboles : $S_{10} = \{0, \ldots, 9\}$
  - Bijection : $m_{10} : S_{10} \rightarrow \mathbb{Z}_{10} : \{(0,0),(1,1),(2,2),\ldots,(\overset{\text{symbole}}{9},\overset{\text{nombre}}{9})\}$

- $k = 10$, lettres
  - Nous pourrions utiliser des symboles différents des chiffres décimaux pour travailler en base 10. Par exemple, les lettres A à J
  - Ensemble des symboles : $S_{10} = \{A, B, C, D, E, \ldots, J\}$
  - Bijection : $m_{10} : S_{10} \rightarrow \mathbb{Z}_{10} : \{(A,0),(B,1),(C,2),\ldots,(I,8),(J,9)\}$
  - Quel nombre représente le mot « CHD » ?

$$x = \sum_{i=0}^{2} m_{10}(w_i) \cdot 10^i$$
$$= m_{10}(C) \cdot 10^2 + m_{10}(H) \cdot 10^1 + m_{10}(D)$$
$$= 200 + 70 + 3 = \boxed{273}$$

Notation Positionnelle Généralisée

Représentation binaire

- Base $k=2$
- En binaire, les symboles sont appelés **bits** (contraction de « *binary digits* »). Un groupe de 8 bits est appelé **octet** (« *byte* » en anglais).
- Ensemble des symboles : $S_2=\{0,1\}$
- Bijection : $m_2 : S_2 \rightarrow \mathbb{Z}_2 : \{(0,0),(1,1)\}$

- Quel nombre représente la séquence « 110101 » ?

$$\begin{aligned} x &= \sum_{i=0}^{5} m_2(w_i) \cdot 2^i \\ &= 1\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 \\ &= 1\cdot 32 + 1\cdot 16 + 0\cdot 8 + 1\cdot 4 + 0\cdot 2 + 1\cdot 1 = \boxed{53} \end{aligned}$$

Notation Positionnelle Généralisée

Représentation hexadécimale

- Base $k=16$
- La représentation hexadécimale est fort utilisée en informatique car la conversion du / vers le binaire est aisée. La représentation hexadécimale est également plus compacte qu'en binaire.
- Ensemble des symboles : $S_{16}=\{0,1,\ldots,8,9,A,B,\ldots,F\}$
- Bijection : $m_{16}: S_{16} \rightarrow \mathbb{Z}_{16}: \{(0,0),\ldots,(9,9),(A,10),\ldots,(F,15)\}$
- <u>Note</u> : pour indiquer que l'on travaille en base 16, les nombres hexadécimaux sont parfois notés avec le préfixe "`0x`". Par exemple, « CAFE » sera souvent noté `0xCAFE`.
- Quel nombre représente la séquence « $F3AD$ » ?

$$x=\sum_{i=0}^{3} m_{16}(w_i)\cdot 16^i$$
$$=m_{16}(F)\cdot 16^3+m_{16}(3)\cdot 16^2+m_{16}(A)\cdot 16+m_{16}(D)$$
$$=15\cdot 4096+3\cdot 256+10\cdot 16+13= \boxed{62381}$$

Notation Positionnelle Généralisée

Exemples de nombres représentés avec différentes bases

Décimal	Binaire	Octal	Hexadécimal
0	0	0	0
2	10	2	2
8	1000	10	8
11	1011	13	B
20	10100	24	14
42	101010	52	2A
117	1110101	165	75
228	11100100	264	E4

Notation Positionnelle Généralisée

Intervalle de nombres représentables

- Un mot w de longueur N peut représenter les nombres naturels dans l'intervalle

$$\boxed{[0,\ k^N - 1]}$$

- Exemple : en binaire, avec N=4, cet intervalle est $[0,\ 15]$.

Longueur minimale de mot

- Par un raisonnement similaire à celui utilisé en décimal, on peut montrer que la longueur N du plus petit mot permettant de représenter un naturel x est donnée par

$$\boxed{N \geq \lceil \log_k(x+1) \rceil}$$

- Exemple : combien de symboles significatifs sont nécessaires pour représenter le nombre 1234 en hexadécimal ?

$$N \geq \lceil \log_{16}(1234+1) \rceil = \lceil \log_{16}(1235) \rceil = \lceil 2{,}566\ldots \rceil = 3$$

Table des Matières

Notation des Nombres

- Représentation décimale
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

- Représentation des naturels
- Représentation des entiers

Changement de base

Objectifs

- **Changement de base**
  - Comment convertir la représentation d'un nombre d'une base dans une autre ?
  - Par exemple, comment passer de la base 10 à la base 2 et vice-versa ? comment convertir entre les représentations binaire, hexadécimale et octale ?

- **Trucs / astuces / algorithmes de conversion**
  - binaire/hexadécimal → décimal
  - binaire ↔ hexadécimal ↔ octal
  - décimal → binaire/hexadécimal

Changement de base

Conversion base $k \rightarrow$ décimal

- La conversion est « immédiate ». Il suffit d'appliquer la formule donnant le nombre représenté par une séquence w en base k pour obtenir sa représentation en décimal (pourquoi?).

$$x = \sum_{i=0}^{N-1} m_k(w_i) \cdot k^i$$

- Exemples
  - « 1011010 » en binaire $\rightarrow$ $2^6 + 2^4 + 2^3 + 2^1 = 90$
  - « 50 » en hexadécimal $\rightarrow$ $5 \times 16 = 80$
  - « 13F » en hexadécimal $\rightarrow$ $1 \times 256 + 3 \times 16 + 15 = 319$

Changement de base

Conversion décimal → base k

- Il est possible d'utiliser la division euclidienne de façon répétée.
- La séquence w qui représente le nombre x en base k vérifie l'équation

$$\begin{aligned} x &= \sum_{i=0}^{N-1} m_k(w_i) \cdot k^i \\ &= m_k(w_0) \cdot k^0 + m_k(w_1) \cdot k^1 + \ldots + m_k(w_{N-1}) \cdot k^{N-1} \end{aligned}$$

- En factorisant k chaque fois que cela est possible, cette équation peut être ré-écrite comme suit

$$x = \boxed{m_k(w_0)} + k \cdot \boxed{\left(m_k(w_1) + k \cdot \left(\ldots + k \cdot m_k(w_{N-1})\right)\right)}$$

- En effectuant la division euclidienne de x par la base k, on obtient
  - quotient: $q = x / k = m_k(w_1) + k \cdot \left(\ldots + k \cdot m_k(w_{N-1})\right)$
  - reste: $r = x \bmod k = m_k(w_0) \xrightarrow[m_k^{-1}]{} w_0$

Conversion décimal → non-décimal (suite)

- L'application répétée de la division sur le quotient obtenu permet d'obtenir successivement w_0, ..., w_{N-1}

$$x = m_k(w_0) + k \cdot \left(m_k(w_1) + k \cdot \left(\ldots + k \cdot m_k(w_{N-1})\right)\right)$$

$$q = m_k(w_1) + k \cdot \left(\ldots + k \cdot m_k(w_{N-1})\right)$$
$$r = m_k(w_0)$$

La même opération est répétée en partant de $x{=}q$

$$x = m_k(w_1) + k \cdot \left(\ldots + k \cdot m_k(w_{N-1})\right)$$

$$q = m_k(w_2) + k \cdot \left(\ldots + k \cdot m_k(w_{N-1})\right)$$
$$r = m_k(w_1)$$

$$\vdots$$

$$x = m_k(w_{N-1})$$

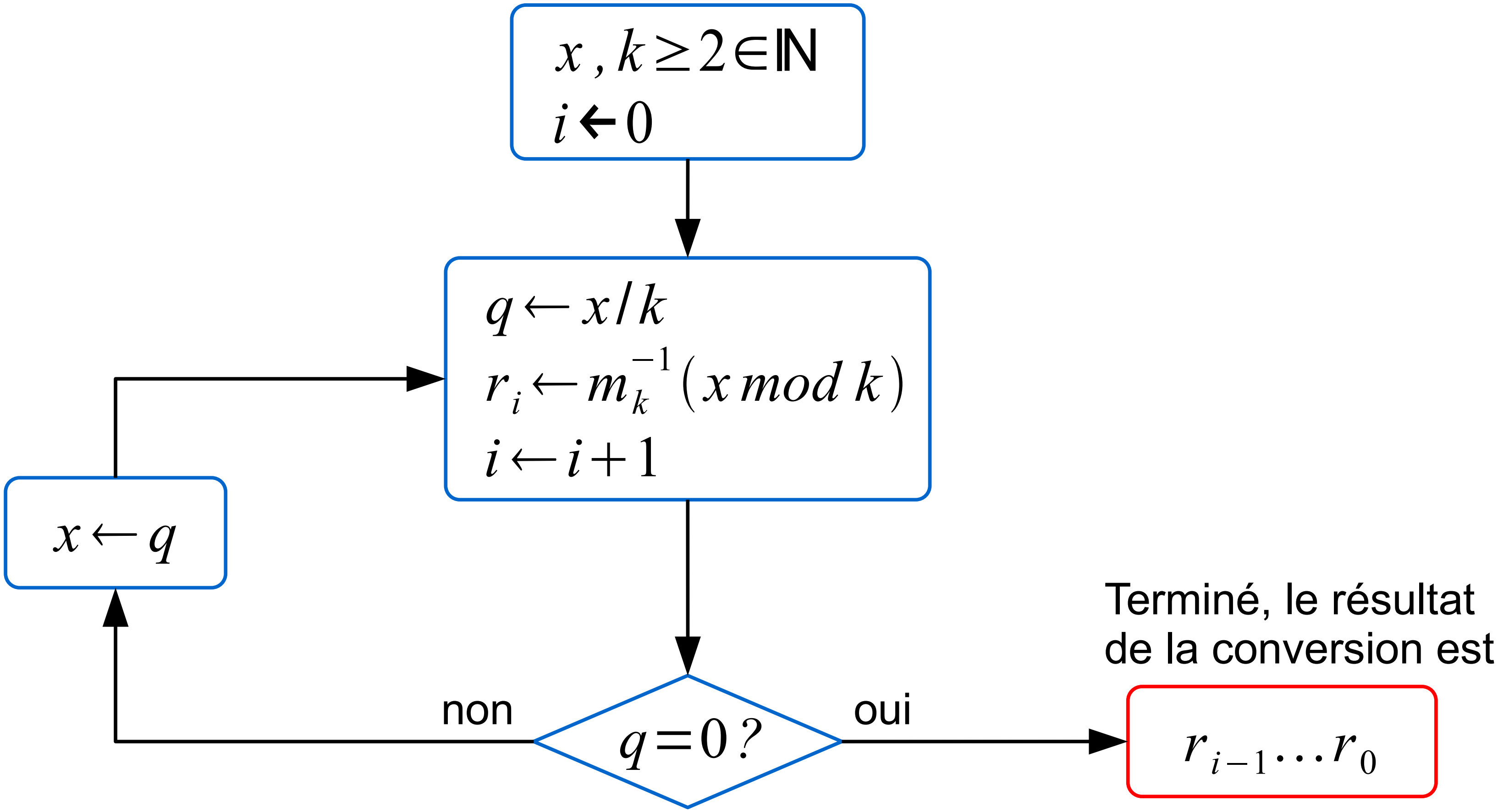
$$q = 0$$
$$r = m_k(w_{N-1})$$

La conversion est terminée lorsque le quotient vaut 0

Changement de base

Algorithme

- La conversion de décimal en non-décimal peut être réalisée à l'aide de l'algorithme suivant :



Changement de base

Implémentation en Python3

```
import sys

SYMBOLS = "0123456789ABCDEF"

base_s = input("Entrez la base: ")
base = int(base_s, 10)
if (base < 2) or (base > 16):
    sys.exit("Erreur: la base doit etre " \
             "comprise entre 2 et 16")

x_s = input("Entrez le nombre: ")
x = int(x_s, 10)
if (x < 0):
    sys.exit("Erreur: le nombre doit etre positif")

word = "0" if (x == 0) else ""
while (x != 0):
  q = x // base
  r = x % base
  word = "%c%s" % (SYMBOLS[r], word)
  x = q

print("en base %d : %s" % (base, word))
```

Exemple

- Quelle est la représentation binaire du nombre 1984 ?

x	quotient	reste
1984 / 2	992	0
992 / 2	496	0
496 / 2	248	0
248 / 2	124	0
124 / 2	62	0
62 / 2	31	0
31 / 2	15	1
15 / 2	7	1
7 / 2	3	1
3 / 2	1	1
1 / 2	0	1

bit de poids faible

11111000000_2

bit de poids fort

Exemple

- Quelle est la représentation hexadécimale du nombre 1984 ?

x	quotient	reste
1984 / 16	124	0
124 / 16	7	12
7 / 16	0	7

m_{16}^{-1} symbole de poids faible

$7C0_{16}$

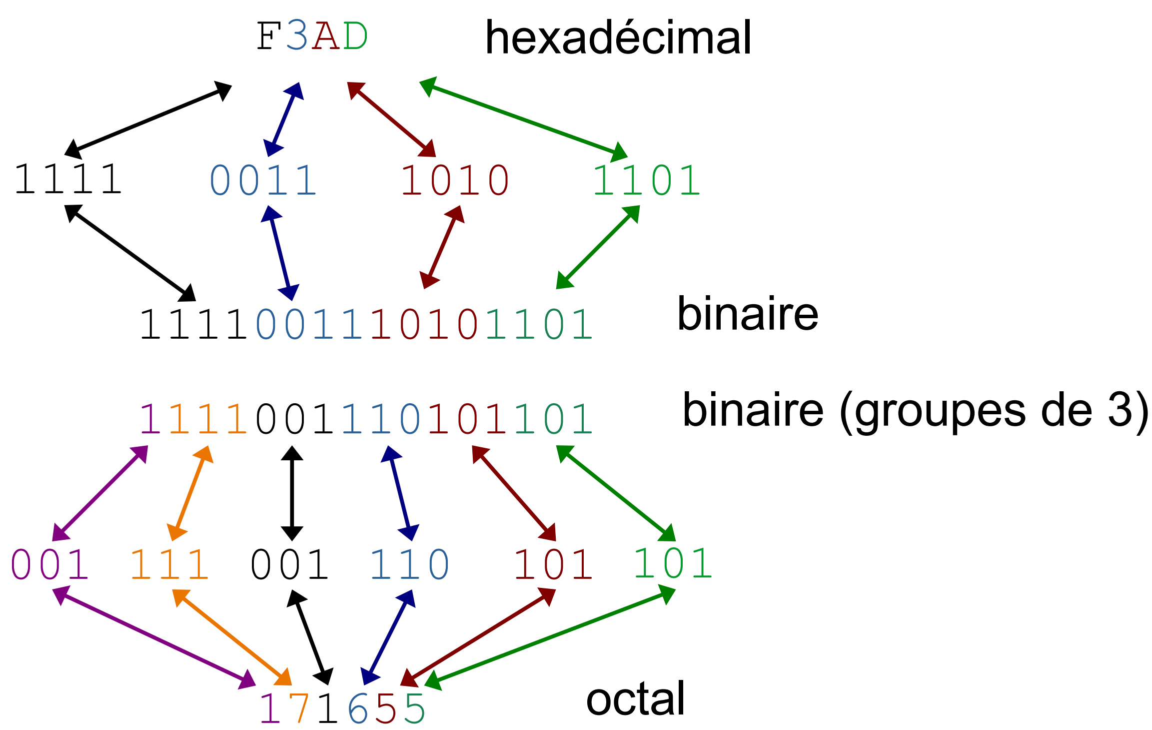
m_{16}^{-1} symbole de poids fort

$$m_{16}^{-1}:\mathbb{Z}_{16}\rightarrow S_{16}:\{(0,0),\ldots,(9,9),(10,A),\ldots,(15,F)\}$$

Conversion binaire ↔ hexadécimal/octal

- Les représentations octales et hexadécimales utilisent des bases qui sont des exposants de 2. Cela signifie qu'un symbole octal ou hexadécimal correspond directement à plusieurs bits.
  - Octal, $k = 8 = 2^3$ → 1 symbole octal = 3 bits
  - Hexadécimal, $k = 16 = 2^4$ → 1 symbole hexadécimal = 4 bits

- <u>Exemple</u>



Hex	Dec	Bin
D	13	1101
A	10	1010
3	3	0011
F	15	1111

Table des Matières

Notation des Nombres

- Représentation décimale
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

- Représentation des naturels
- Représentation des entiers

Nombres dans un Ordinateur

Représentation binaire

- La majorité des ordinateurs d'aujourd'hui représentent des nombres **sous forme binaire**. Les bits 0 et 1 sont manipulés à l'intérieur d'un processeur sous forme de deux tensions électriques différentes.
  - par exemple : 0 ↔ 0 V et 1 ↔ 5 V
- Un processeur utilise des **mots binaires**, le plus souvent **de taille fixe**, pour représenter des nombres. Les tailles usuelles de ces mots sont des exposants de 2, plus spécifiquement 8, 16, 32 et 64 bits.

8 bits	16 bits	32 bits	64 bits

- Les opérations arithmétiques sont effectuées sur la représentation des nombres sous forme de mots binaires. Dans un processeur, ces opérations sont réalisées par l'**Unité Arithmétique et Logique** (ALU).
- Les processeurs utilisent des conventions différentes pour représenter les ***naturels*** (dits « *non-signés* ») et les ***entiers*** (dits « *signés* »). Nous les discutons séparément dans ce chapitre.

Représentation des Naturels

Principe

- Les *nombres naturels* (*entiers non signés*) sont représentés en utilisant la **notation positionnelle**, dans des **mots binaires de taille fixe**, . Un mot de N bits permet de représenter des valeurs dans l'intervalle

$$\boxed{[0, 2^N - 1]}$$

- Chaque processeur est capable de manipuler des mots de tailles spécifiques.
- Certains langages de programmation permettent de contrôler la taille de représentation en associant des types spécifiques aux variables.
  - Exemple, en langage C sur un processeur 32 bits:

N	Type	Intervalle de nombres
8	`unsigned char`	[0, 255]
16	`unsigned short`	[0, 65535]
32	`unsigned int`	[0, 4294967296]
64	`unsigned long int`	[0, 18446744073709551616]

Table des Matières

Notation des Nombres

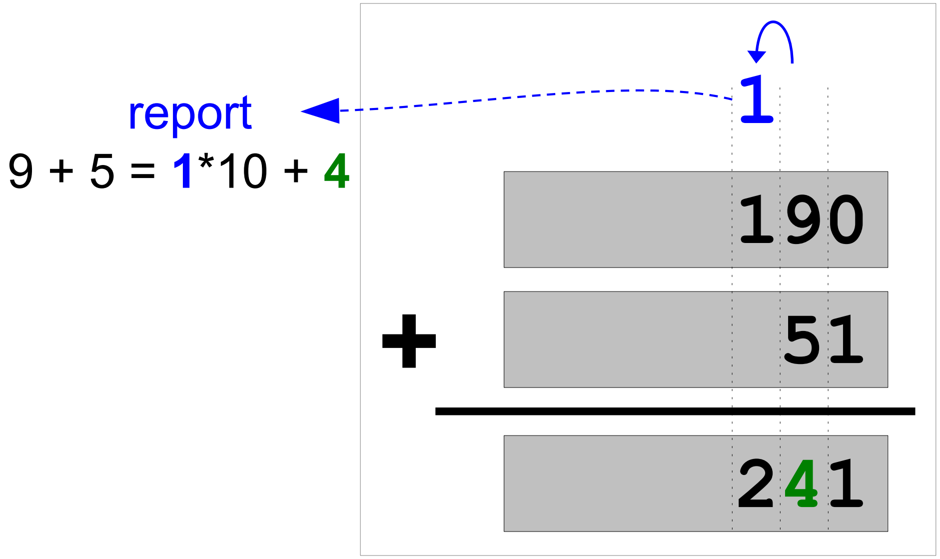
- Représentation décimale
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

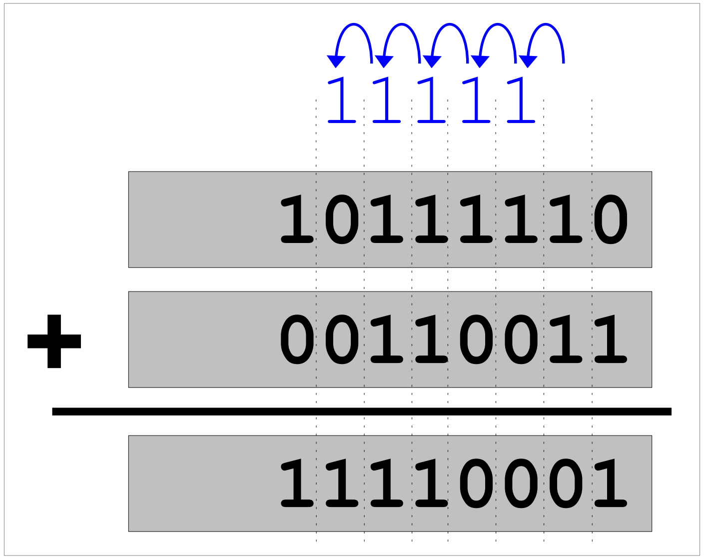
- Représentation des naturels
  - Addition
  - Soustraction
  - Décalages à gauche et à droite
  - Multiplication
  - Division
- Représentation des entiers

Addition en décimal vs en binaire

- Pour effectuer l'addition de deux nombres naturels x et y, nous utilisons un algorithme qui travaille sur la représentation décimale/binaire de ces nombres.
- L'algorithme procède **rang par rang**, en commençant par les chiffres/bits de poids faible. A chaque rang i $(0 \leq i < N)$, un chiffre/bit s_i de la somme est produit sur base des chiffres/bits x_i et y_i. Un **report** (*carry*) peut être propagé au rang suivant.



En décimal



En binaire

Opérations sur les Naturels

Exemple en langage C

```c
#include <stdio.h>

int main() {
  unsigned char x = 190;
  unsigned char y = 141;
  unsigned char z = x + y;
  printf("%u\n", z);
  return 0;
}
```

Qu’affichera le programme ?

Notes :
pour compiler l’exemple ci-dessus (`main.c`), utilisez la commande

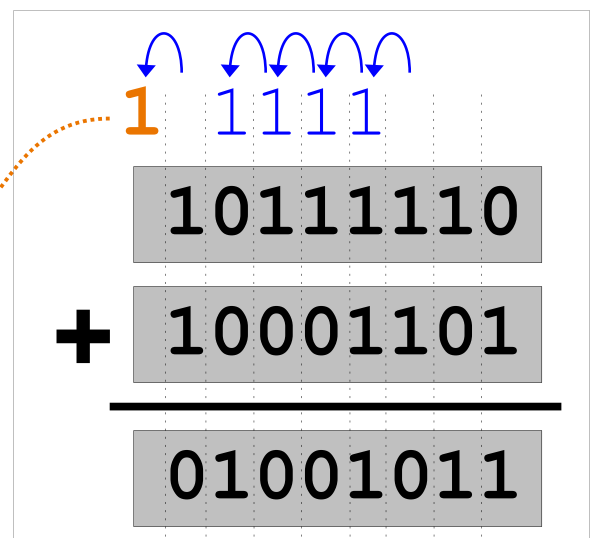
```
gcc -Wall -Werror -o main main.c
```

pour l’exécuter, utilisez

```
./main
```

Dépassement (*overflow*)

- L'addition a lieu **sur N bits**.
  - Un mot peut représenter des nombres compris entre 0 et 2^N-1.
  - La somme de deux nombres est comprise entre 0 et $2\cdot(2^N\text{-}1) = 2^{N+1}\text{-}2$
- Si le résultat n'est pas représentable sur N bits, i.e. si $x + y > 2^N - 1$, alors il y a **dépassement** (*overflow*). Seuls les N bits de poids faible du résultat sont conservés. Lorsqu'un dépassement a lieu, il y a un report au dernier rang et le résultat calculé est incorrect.
- Exemple (N=8)



Le bit de report de rang N-1 indique qu'il y a dépassement.

(190)

(141)

(75) Le résultat est incorrect !

Opérations sur les Naturels

Addition modulo 2^N

- Lors d'un dépassement, le bit de report est ignoré. Par conséquent, l'addition de nombres naturels représentés avec N bits dans un ordinateur est une addition « modulo 2^N ».
- Si on note l'addition de naturels (addition non-signée) par l'opérateur $+_{ns}$, on a que

$$x \; +_{ns} \; y = \begin{cases} x+y & si\ x+y<2^N \\ x+y-2^N & si\ x+y \geq 2^N \end{cases}$$

- <u>Exemple (N=8)</u> : $190 +_{ns} 141 = 331 - 256 = 75$

- <u>Pour les matheux</u> : l'ensemble des nombres naturels représentables avec N bits muni de l'addition $+_{\mathrm{ns}}$ forme un groupe commutatif.
  Question subsidiaire : comment le prouver ?

Table des Matières

Notation des Nombres

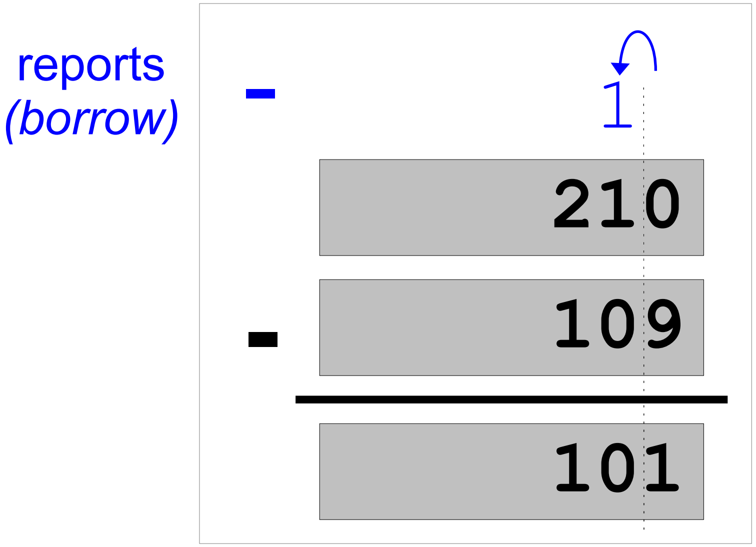
- Représentation décimale
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

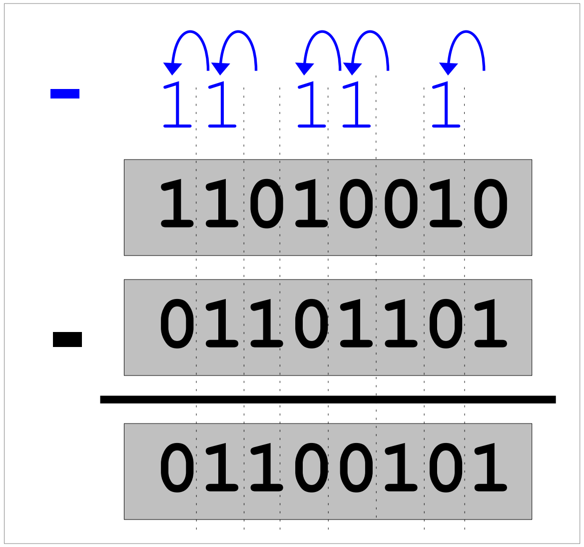
- Représentation des naturels
  - Addition
  - Soustraction
  - Décalages à gauche et à droite
  - Multiplication
  - Division
- Représentation des entiers

Soustraction en décimal vs en binaire

- Comme pour l'addition, l'algorithme de soustraction fonctionne rang par rang. La différence est qu'à chaque rang, une opération de soustraction est effectuée et qu'une **retenue** (*borrow*) est effectuée. Celle-ci est soustraite au rang suivant.



En décimal



En binaire

Opérations sur les Naturels

Résultat négatif (*underflow*)

- La soustraction de nombres naturels peut ne pas avoir un résultat naturel, si $x < y$, alors $x\text{-}y < 0$ n'est pas représentable.
- Dans cette situation, il existe un report au dernier rang et le résultat est incorrect.
- Exemple ($N = 8$)

Il y a un **report final non nul**.
Le résultat est négatif
(pas un naturel).

```
 - 1 1 1 1 1 1 1 1
      11010010      (210)
 -    11011011      (219)
   -------------
      11110111      (247)
```

Le résultat est incorrect !

  – Comme on travaille sur N=8 bits, le résultat vaut 247 au lieu de -9. L'explication est similaire à celle de l'addition modulo 2^N : ($247 = 2^N - 9 = 2^8 - 9 = 256 - 9$)

Table des Matières

Notation des Nombres

- Représentation décimale
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

- Représentation des naturels
  - Addition
  - Soustraction
  - Décalages à gauche et à droite
  - Multiplication
  - Division
- Représentation des entiers

Décalages à gauche et à droite

- Les opérations de décalage à gauche (notée « **<<** ») et à droite (notée « **>>** ») ne sont pas à proprement parler des opérations arithmétiques. Elles permettent cependant d'effectuer efficacement des multiplications et divisions entières par des exposants de 2.

<u>Décalage à gauche</u>

```
      00010101   (21)
<<           2
   -----------
      01010100   (84)
```

y bits de poids fort sont perdus.

y bits de poids faible valant 0 sont insérés.

<u>Décalage à droite</u>

```
      01111011   (123)
>>           3
   -----------
      00001111   (15)
```

y bits de poids faible sont perdus.

y bits de poids fort valant 0 sont insérés.

Décalages à gauche et à droite

- Si x comporte au plus N-y chiffres significatifs(1), alors le décalage à gauche est équivalent à la multiplication par 2^y.

$$x \ll y \;=\; x \cdot 2^y$$

- Le décalage à droite est équivalent à la division entière par 2^y.

$$x \gg y \;=\; x / 2^y$$

(1) Ce qui est équivalent à $x < 2^{N-y}$. Par conséquent $x \cdot 2^y < 2^{N-y+y} = 2^N$ ce qui implique que $x \cdot 2^y$ est représentable avec N bits.

Exemples en langages C et Java

- Des opérateurs permettant d'effectuer des décalages à gauche et à droite sont proposés par de nombreux langages de programmation, tels que Java, C et C++. Ces opérateurs sont notés “**<<**” pour le décalage à gauche et “**>>**” pour le décalage à droite.

C

```c
#include <stdio.h>
int main() {
  unsigned char x= 20;
  unsigned char y= x << 3;
  printf("%d\n", y);
  return 0;
}
```

Le programme affiche “160”.

Représentation de `x` en mémoire:

00010100

Représentation de `y` en mémoire:

10100000

Java

```java
public class Test {
  public static void main(String [] args) {
    byte x= 20;
    byte y= x << 3;
    System.out.println(y);
  }
}
```

Le programme affiche “-96”.

Représentation de `x` en mémoire:

00010100

Représentation de `y` en mémoire:

10100000

Table des Matières

Notation des Nombres

- Représentation décimale
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

- Représentation des naturels
  - Addition
  - Soustraction
  - Décalages à gauche et à droite
  - Multiplication
  - Division
- Représentation des entiers

Multiplication en décimal

- Pour effectuer une multiplication entre deux nombres naturels représentés en décimal, l'algorithme généralement appliqué consiste à effectuer des multiplications intermédiaires.
- Chaque produit intermédiaire est obtenu en multipliant le *multiplicande* (x) par un chiffre du *multiplicateur* (y). Le produit final est obtenu en additionnant les résultats des multiplications intermédiaires.

```
      11    x
x     13    y
  --------
      33          (3*11)
+    110          (1*11*
  --------
     143    p
```

produits intermédiaires

Note : 11 décalé d'un chiffre vers la gauche

Multiplication en décimal vs en binaire

- On peut écrire plus formellement la multiplication en décimal effectuée par l'algorithme du slide précédent.
- On se rappelle que le multiplicateur y est représenté avec la notation positionnelle.

$$y = \sum_{i=0}^{M-1} y_i \cdot 10^i \qquad\qquad y = \sum_{i=0}^{M-1} y_i \cdot 2^i$$

- Le produit est calculé comme suit

y_i vaut 0 ou 1,
donc $y_i \cdot x$ vaut 0 ou x

$$p = x \cdot y = \sum_{i=0}^{M-1} \underbrace{y_i \cdot x \cdot 10^i}_{\text{produits intermédiaires}}$$

$$p = x \cdot y = \sum_{i=0}^{M-1} y_i \cdot x \cdot 2^i = \sum_{i=0}^{M-1} \left(y_i \cdot x\right) \ll i$$

La multiplication par 10^i ou 2^i correspond à un décalage de i chiffres vers la gauche.

Opérations sur les Naturels

Exemple

	11	x	
x	13	y	
	33		(3*11)
+	110		(1*11)
	143	p	

N bits

	1011	x
x	1101	y
	1011	$y_0.x$
+	00000	$y_1.x << 1$
+	101100	$y_2.x << 2$
+	1011000	$y_3.x << 3$
	10001111	p

$2N$ bits

dans le pire des cas, le produit sera $(2^N\text{-}1)^2 = 2^{2N} - 2^{N+1} + 1$
ce qui nécessite $2N$ bits pour être représenté.

Accumulation des produits partiels

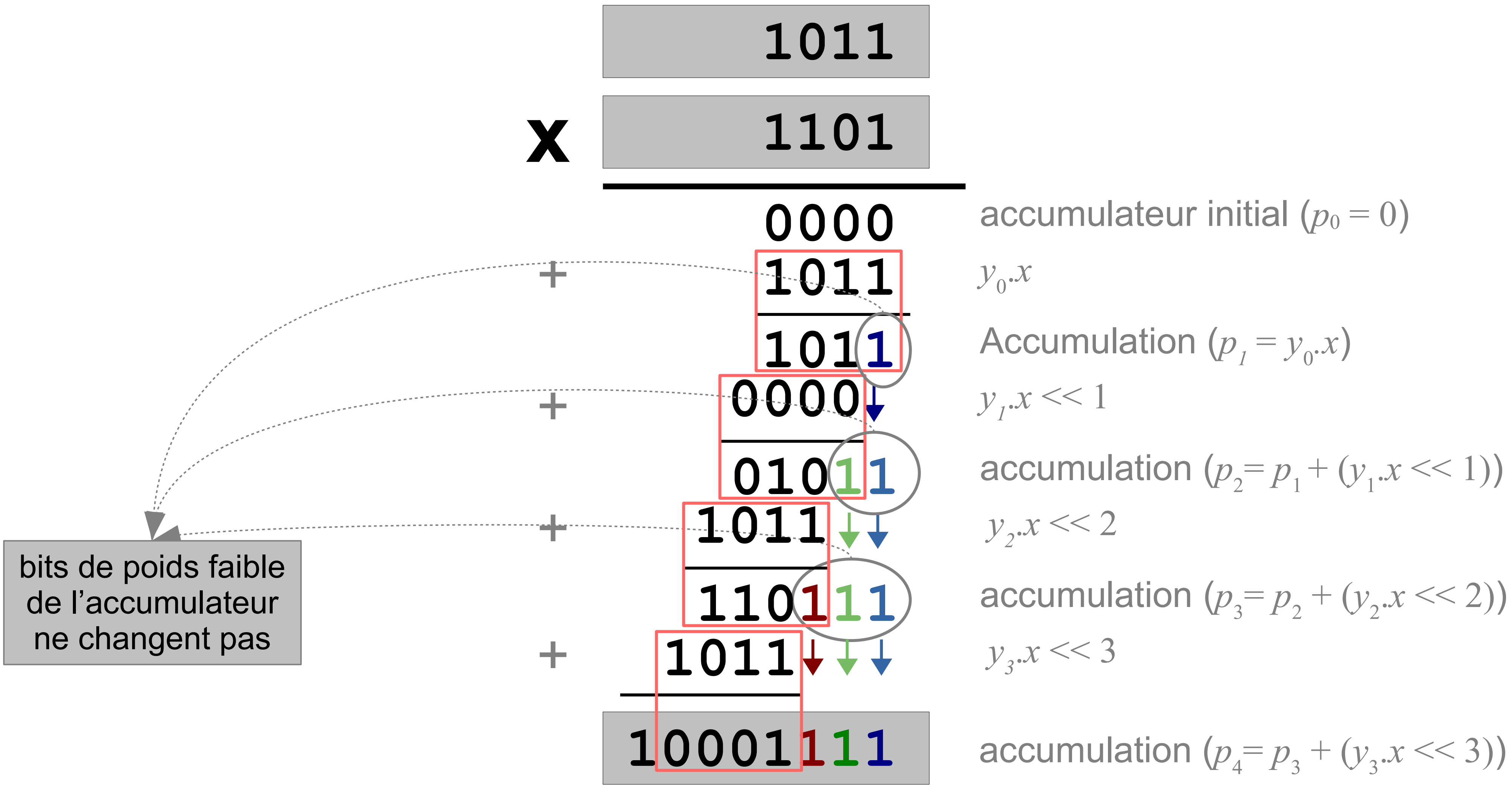
- Il reste un problème avec cet algorithme qui le rend difficile à implémenter sous forme matérielle (électronique). Il faut en effet calculer les produits partiels, les conserver et les additionner ensuite.

- On propose d'additionner les produits partiels au fur et à mesure de leur production dans un accumulateur p. Au début p vaut 0 et en fin d'algorithme il vaudra le produit de x et y.
- A chaque itération i ($0 \leq i < N$), p est mis à jour comme suit

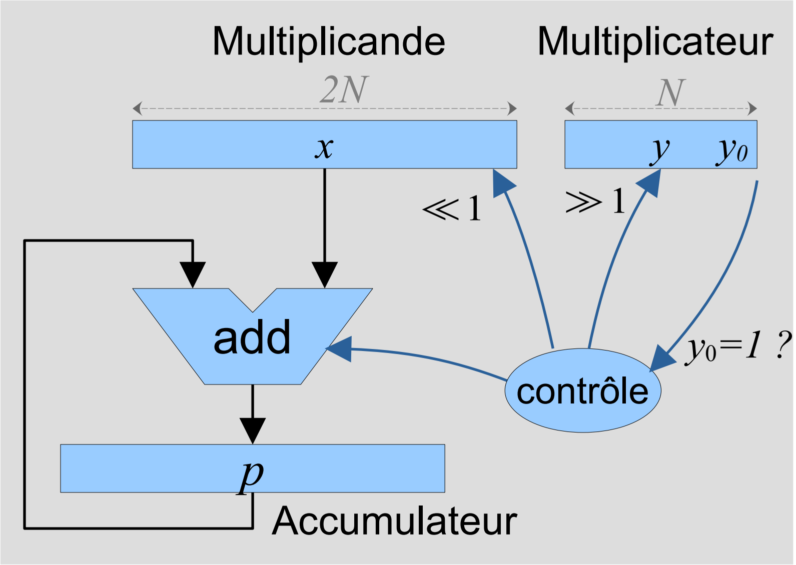
$$p_{i+1} = p_i + \left(\left(y_i \cdot x \right) \ll i \right)$$

- En raison du décalage par i du produit partiel à l'étape i , les i bits de poids faible restent inchangés lors de l'addition à l'accumulateur. Par conséquent, cette addition peut être réalisée sur seulement N bits.

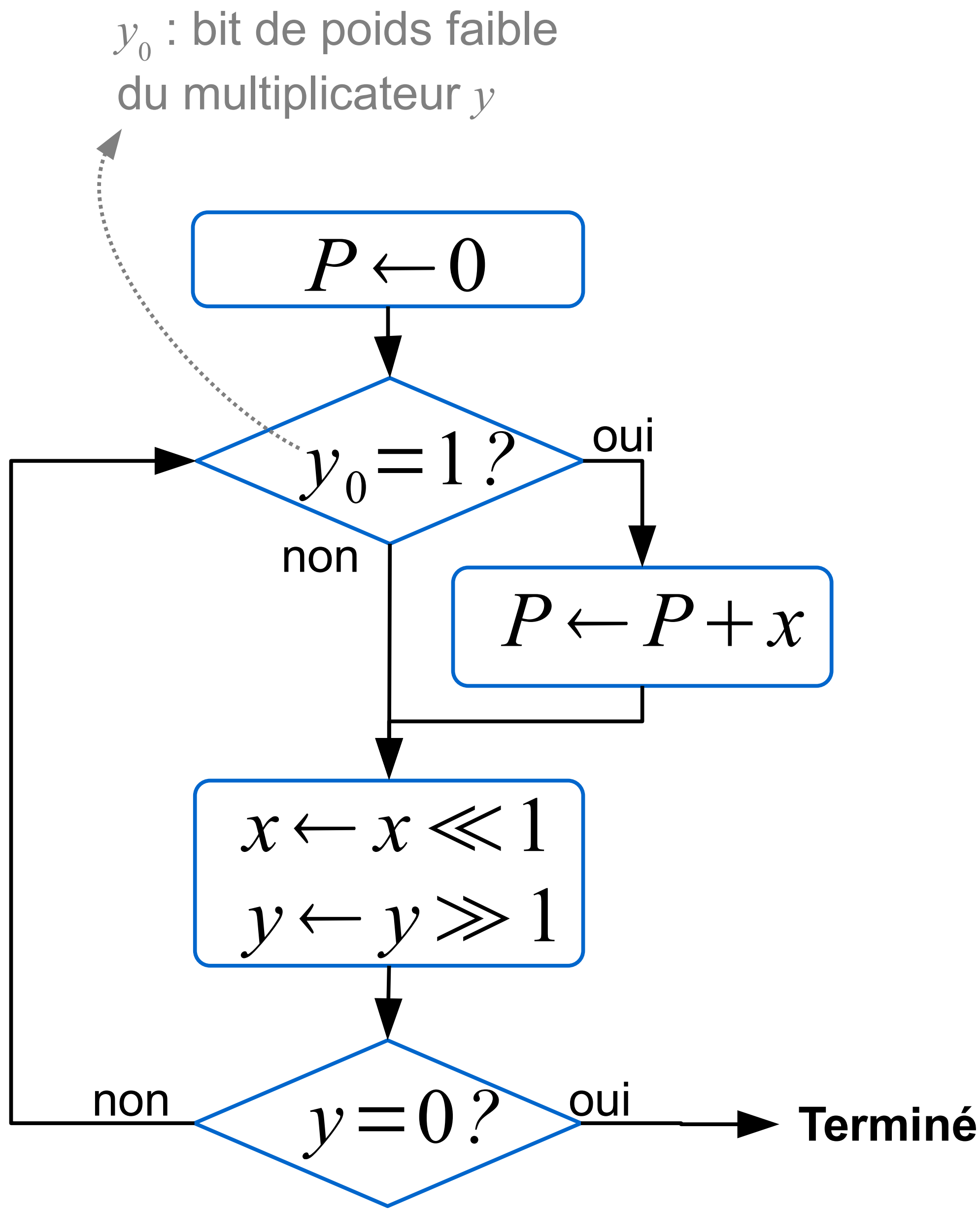
Exemple – accumulation des produits partiels



Opérations sur les Naturels



<u>Explication</u> : x, y et p sont stockés dans des registres. Le registre x peut être décalé vers la gauche (<< 1), y vers la droite (>> 1). Le contrôle de l'algorithme teste si $y_0=1$, effectue l'addition de x et p, décale les registres et répète ces opérations.



Exemple

- Multiplication de $x=7$ (0111) par $y=5$ (0101)
  - init
    $P \leftarrow 0$
  - étape 1
    $y_0 \neq 0 \Rightarrow P \leftarrow P+x$ (111)
    $x \leftarrow x << 1$ (1110)
    $y \leftarrow y >> 1$ (10)
  - étape 2
    $y_0 = 0$
    $x \leftarrow x << 1$ (11100)
    $y \leftarrow y >> 1$ (1)
  - étape 3
    $y_0 \neq 0 \Rightarrow P \leftarrow P+x$ (100011 i.e. 35 en décimal)
    $x \leftarrow x << 1$ (111000)
    $y \leftarrow y >> 1$ (0)
  - fin ($y = 0$)

Table des Matières

Notation des Nombres

- Représentation décimale
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

- <u>Représentation des naturels</u>
  - Addition
  - Soustraction
  - Décalages à gauche et à droite
  - Multiplication
  - Division
- Représentation des entiers

Division en décimal

- La division entière en décimal (dite « division longue ») fonctionne suivant un algorithme avec décalage et soustraction successifs.

- Exemple : division de 217 par 11.

```
  217   /11
  110          (1*11*10^1)
-----
  107
   99          (9*11*10^0)
-----
    8      19
(reste)  (quotient)
```

Dividende= 217

Divisible par un multiple entier de $11*10^1$?

Oui : quotient= 1, reste= 107.

Nouveau dividende= 107

Divisible par un multiple entier de $11*10^0$?

Oui : quotient= 9, reste= 8.

Reste= 8. Fini.

Division en décimal vs en binaire

- Par définition, la division entière d'un dividende (D) par un diviseur (d) donne un quotient (q) et un reste (r). Elle est exprimée comme :

$$D = d \cdot q + r$$

- Si le quotient est exprimé en représentation positionnelle, on obtient

$$\begin{aligned} D &= d \cdot q + r \\ &= d \cdot \left(\sum_{i=0}^{N-1} q_i \cdot 10^i \right) + r \\ &= q_{N-1} \cdot d \cdot 10^{N-1} + \underbrace{q_{N-2} \cdot d \cdot 10^{N-2} + \ldots + q_0 \cdot d \cdot 10^0 + r}_{< \; d \cdot 10^{N-1}} \end{aligned}$$

- On constate que diviser par $d \cdot 10^{N-1}$ fournit q_{N-1} comme quotient et le reste de la somme comme reste.

- En répétant cette division par $d.10^{N-2}$, $d.10^{N-3}$, …, $d.10^0$, sur le reste obtenu à l'étape précédente, toutes les autres composantes q_{N-2}, …, q_0 du quotient sont obtenues.

Opérations sur les Naturels

Division en décimal vs en binaire

- Le principe de la division en binaire est identique, <u>à l'exception qu'à chaque étape, la composante q_i du quotient obtenue vaut soit 1 soit 0</u>.

$$\begin{aligned} D &= d \cdot q + r \\ &= d \cdot \left(\sum_{i=0}^{N-1} q_i \cdot 2^i \right) + r \\ &= \underbrace{q_{N-1}}_{0 \text{ ou } 1} \cdot d \cdot 2^{N-1} + \underbrace{q_{N-2} \cdot d \cdot 2^{N-2} + \ldots + q_0 \cdot d \cdot 2^0 + r}_{< \; d \cdot 2^{N-1}} \end{aligned}$$

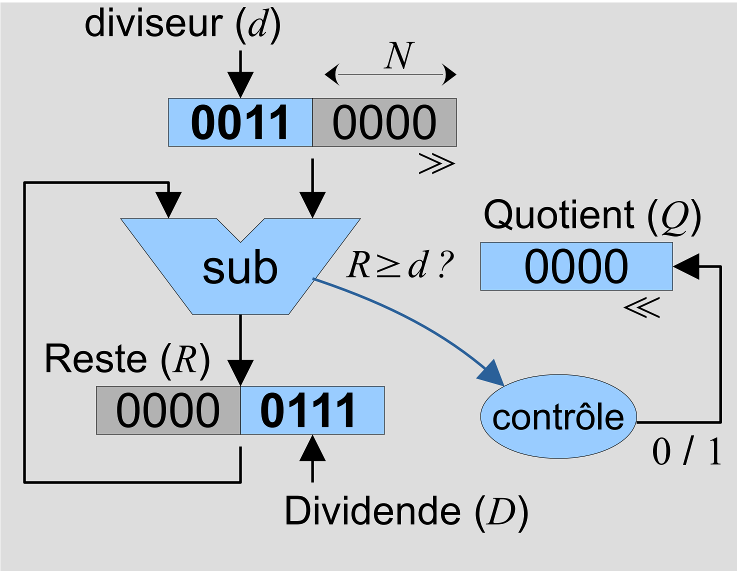
- <u>Conséquences</u> :
  - il ne faut donc pas effectuer de division entière à chaque étape mais simplement tester si le dividende courant est supérieur à $d \cdot 2^i$.
  - de plus, $d \cdot 2^i$ est égal à $d << i$, ce qui peut être facilement réalisé par une implémentation matérielle.

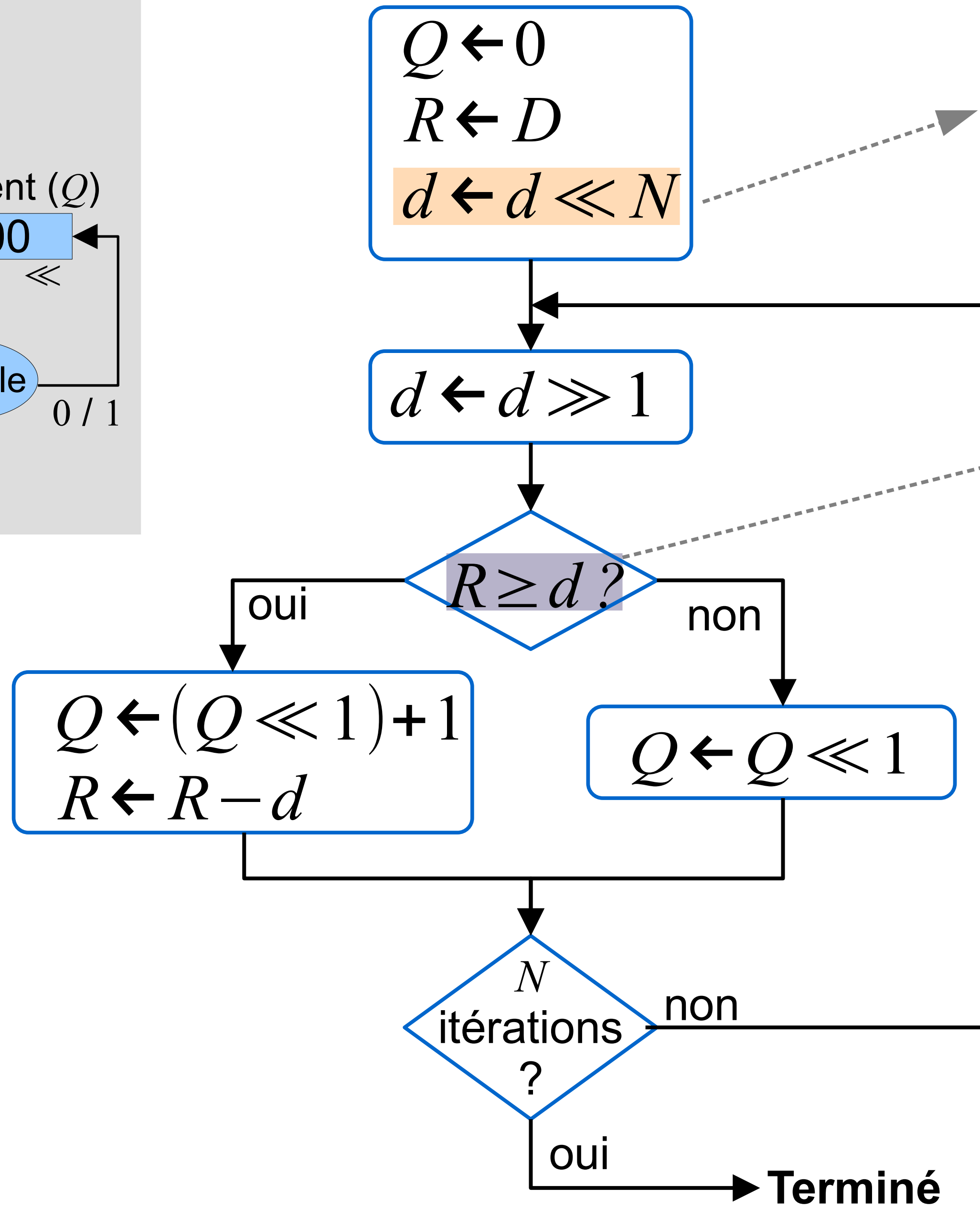
Exemple

- Division de 217 par 11 en binaire.

```
11011001   / 1011
1011                 (1*1011) << 4
--------
 0101
 0000                (0*1011) << 3
--------
  1010
  0000               (0*1011) << 2
--------
  10100
   1011              (1*1011) << 1
--------
   10011
    1011             (1*1011)
--------
    1000             10011
   (reste)           (quotient)
```

Opérations sur les Naturels





Le processeur n'est pas assez "*malin*" pour faire le décalage optimal.

Tester $R \geq \mathrm{d}$ est typiquement effectué en testant $R\text{-}d \geq 0$

Q : Quotient
R : reste
D : Dividende
d : diviseur
N : largeur (bits)

Exemple

- **Division binaire**
  - division de $D=7$ (0111) par $d=3$ (0011), $N=4$
    - init
      $Q \leftarrow 0 \;\; ; \;\; R \leftarrow D\,(111) \;\; ; \;\; d \leftarrow \mathrm{d} << \mathrm{N}\,(110000)$
    - itération 1
      $d \leftarrow d >> 1 \quad (11000)$
      $R < d \;\Rightarrow\; Q \leftarrow Q << 1 \quad (0)$
    - itération 2
      $d \leftarrow d >> 1 \quad (1100)$
      $R < d \;\Rightarrow\; Q \leftarrow Q << 1 \quad (0)$
    - itération 3
      $d \leftarrow d >> 1 \quad (110)$
      $R \geq d \;\Rightarrow\; Q \leftarrow (Q << 1) + 1 \quad (1)$
      $R \leftarrow R - d \quad (1)$
    - itération 4
      $d \leftarrow d >> 1 \quad (11)$
      $R < d \;\Rightarrow\; Q \leftarrow Q << 1 \quad (10)$
    - fin: quotient = 10 (2), reste = 1 (1)

Exemple

- **Division binaire**
  - division de *D*=51 (110011) par *d*=9 (001001), *N*=6
    - init: Q=0 ; R=D ; d=1001000000
    - itération 1
      d=100100000 ; (R < d)
    - itération 2
      d=10010000 ; (R < d)
    - itération 3
      d=1001000 ; (R < d)
    - itération 4
      d=100100 ; (R ≥ d) => Q=1 ; R=1111
    - itération 5
      d=10010 ; (R < d) => Q=10
    - itération 6
      d=1001 ; (R ≥ d) => Q=101 ; R=110
    - fin: quotient = 101 (5), reste = 110 (6)

Table des Matières

Notation des Nombres

- Notation Décimale Positionnelle
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

- Représentation des naturels
- Représentation des entiers

Représentation des Entiers

Principe

- **Représentation des entiers**
  - Le principe de représentation des *entiers* est très similaire à celui de la représentation des *naturels*.
  - La différence est qu'il faut pouvoir distinguer les nombres positifs des nombres négatifs. Il faut également adapter les opérations arithmétiques.

- **Nombres signés**
  - Pour représenter un nombre négatif, nous le préfixons avec un symbole particulier, le tiret (−).
  - En informatique, d'autres moyens sont utilisés. Nous allons en considérer deux
    - l'utilisation d'un **bit de signe**
    - la représentation en **complément à deux**

Première tentative – utilisation d'un bit de signe

- Le principe consiste à réserver un bit dans la représentation pour indiquer s'il s'agit d'un nombre positif ou d'un nombre négatif. Par exemple : 0 positif / 1 négatif
  - Si $w_{N-1} = 0$, alors $x \geq 0$. Sinon, $x \leq 0$

$$\boxed{w = w_{N-1} w_{N-2} \ldots w_1 w_0}$$

- La valeur du nombre x est alors donnée par l'équation suivante.

$$\boxed{x = (-1)^{w_{N-1}} \cdot \sum_{i=0}^{N-2} w_i \cdot 2^i}$$

- Exemple ($N = 12$) :

$$w = 1\,10011000011 \qquad (-1)^1 \cdot (1024 + 128 + 64 + 2 + 1) = -1219$$

$$w = 0\,10011000011 \qquad (-1)^0 \cdot (1024 + 128 + 64 + 2 + 1) = 1219$$

Première tentative – utilisation d'un bit de signe

- **Problème (1)** : l'utilisation d'un bit de signe donne lieu à l'existence de 2 représentations différentes du nombre zéro.

$w = 0\,0\ldots00$ représente $x = +0$

$w = 1\,0\ldots00$ représente $x = -0$

- **Problème (2)** : l'addition binaire ne fonctionne pas toujours
  - Exemple (N=4)
    - $x = 2$ → représenté par `0010`
    - $-x = -2$ → représenté par `1010`
    - addition directe des représentations donne `1100` (-4)
  - Il serait possible, en fonction des signes des nombres à additionner, de travailler avec les valeurs absolues des nombres et de faire selon les cas des additions ou soustractions. Cependant, cela est moins facile à implémenter sous forme matérielle.

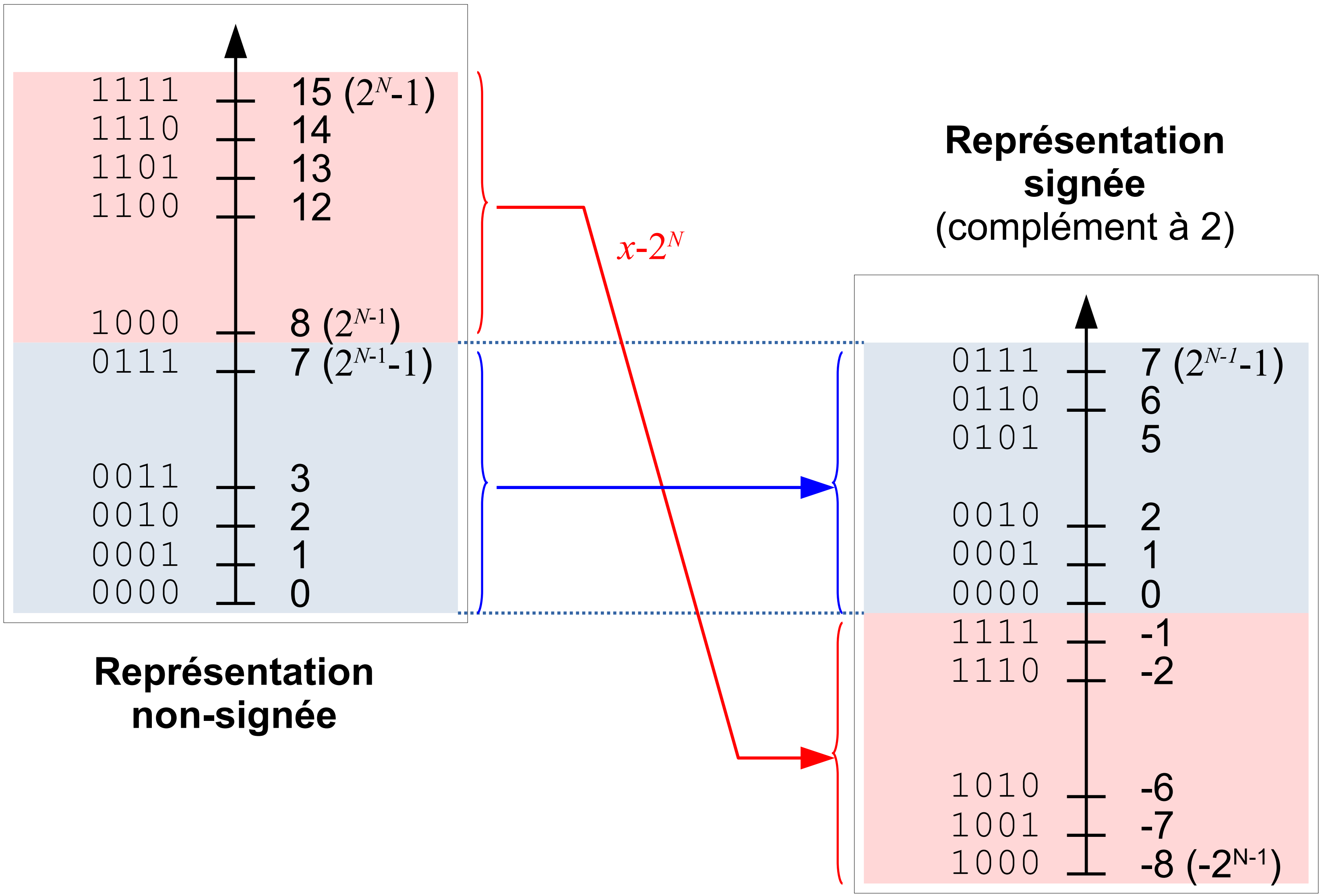
Seconde tentative – complément à 2

- Objectif : trouver une représentation dans laquelle l'addition binaire directe des représentations d'un nombre et de son opposé (son complément) donne 0.
- Il existe différents systèmes possédant cette propriété. Le plus répandu est la **représentation en complément à 2** (*two's complement*). Elle est utilisée dans la plupart des processeurs aujourd'hui.
- Il existe d'autres représentations possédant cette propriété, comme le “complément à 1” (*ones' complement*) que nous n'aborderons pas ici [1].

(1) Note : nous aurons la chance (!) d'en parler lors du cours de Réseaux I.

Représentation des Entiers

Seconde tentative – complément à 2



Représentation des Entiers

Seconde tentative – complément à 2

- Définition : sur N bits, la représentation de l'opposé (le complément) d'un nombre x ($0 < x < 2^{N-1}$) est donnée par la représentation non signée de 2^N-x.

- Exemple (N=4) :
  - $x = 2 \rightarrow$ représenté par $w =$ `0010`
  - $-x = -2 \rightarrow$ représentation de 2^N-$x = 14$ par $w =$ `1110`

- Propriétés :
  - On peut vérifier que $(x + 2^N - x) \bmod 2^N = 0$
  - Le complément du complément de x est $2^N-(2^N-x) = x$
  - Le complément de 0 est $(2^N - 0) \bmod 2^N = 0$

- Exemple :
  - 2 + (-2) peut être calculé en additionnant les représentations

```
  0010
+ 1110
  ----
  0000
```

Représentation en complément à 2

- Dans la représentation en **complément à 2** (*two's complement*), un entier x est représenté par un mot w de N bits et le bit de poids fort a un poids négatif

$$\boxed{w = w_{N-1} w_{N-2} \ldots w_1 w_0}$$

- Le nombre x est obtenu par

$$\boxed{x = w_{N-1} \cdot \left(\underbrace{-2^{N-1}}_{\text{poids négatif}}\right) + \sum_{i=0}^{N-2} w_i \cdot 2^i}$$

- Exemple (N=12) :
  - L'entier -1219 est représenté par $w = 1\,01100111101$

$$\begin{aligned} x = -1219 &= -2048 + 512 + 256 + 32 + 16 + 8 + 4 + 1 \\ &= -2^{11} + 2^9 + 2^8 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0 \end{aligned}$$

Représentation en complément à 2

- L'intervalle de nombre représentables avec N bits est

$$\left[-2^{N-1}, 2^{N-1}-1\right]$$

- Exemples:
  - Exemples en langage C et Java

N	Type		Intervalle de nombres
	C	Java	
8	`char`	`byte`	[-128, 127]
16	`short`	`short`	[-32768, 32767]
32	`int`	`int`	[-2147483648, 2147483647]
64	`long int`	`long`	[-9223372036854775808, 9223372036854775807]

Note : La borne inférieure est obtenue avec le bit de poids fort à 1 et les autres à 0. La borne supérieure est obtenue comme le nombre maximum représentable en non-signé avec N-1 bits. $\sum_{i=0}^{N-2} 2^i = 2^{N-1}-1$

Représentation en complément à 2

- Voici une astuce pour obtenir facilement la représentation en complément à 2 sur N bits d'un nombre négatif x, avec $2^{N\text{-}1} < x < 0$

  1) Obtenir la représentation non-signée, w, de $| x |$
  2) Déterminer w' en complémentant chaque bit de w
  3) Ajouter 1 à w'

- Exemple ($N = 4$) :
  - comment représenter $x = \text{-}6$?

$$w = 0110 \xrightarrow{\text{complémenter chaque bit}} w' = 1001 \xrightarrow{\text{ajouter 1}} 1010$$

représentation non-signée de $|x|=6$

représentation signée de x=-6

Note : pourquoi cela fonctionne-t-il ?

$$2^N - |x| = \left(\left(2^N - 1\right) - |x|\right) + 1$$

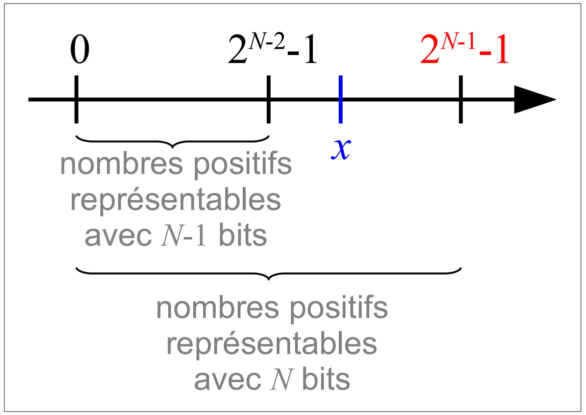
$$\left(2^N - 1\right) - |x| = \left(\sum_{i=0}^{N-1} 2^i\right) - x = \sum_{i=0}^{N-1} \left(1 - w_i\right) \cdot 2^i \qquad 1 - w_i = w'_i, \forall\, 0 \leq i < N$$

Taille de mot

- Quelle taille de mot minimum, N, faut-il utiliser pour représenter un entier x donné ?
- Pour $x \geq 0$, il faut trouver N (naturel) tel que

$$2^{N-2}-1 < x \leq 2^{N-1}-1$$

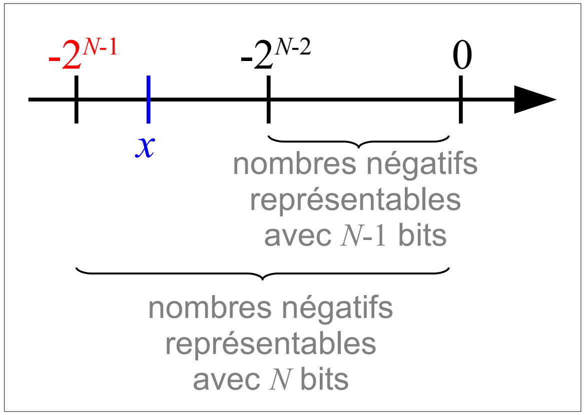
$$\boxed{N = \lceil \log_2(x+1)+1 \rceil}$$



- Pour $x < 0$, il faut trouver N (naturel) tel que

$$-2^{N-1} \leq x < -2^{N-2}$$

$$\boxed{N = \lceil \log_2(-x)+1 \rceil}$$



- <u>Exemples</u>:
  - $x = 234 \rightarrow N = \lceil \log_2(235)+1 \rceil = 9$
  - $x = -39 \rightarrow N = \lceil \log_2(39)+1 \rceil = 7$

Représentation des Entiers

Types des variables et transtypage

- Dans un langage de programmation tel que Java, C ou C++, le type d'une variable est intimement lié à la représentation du contenu de la variable en mémoire.
- Par exemple
  - en Java, type **`int`** → 32 bits, complément à 2
    type **`byte`** → 8 bits, complément à 2
  - en C et en C++, type **`char`** → 8 bits, complément à 2
    type **`unsigned char`** → 8 bits, non signé
- Il est important de savoir **à quoi correspond un type de variable**, par exemple lorsque l'on utilise des opérations comme le **transtypage**, i.e. lorsque l'on interprète le contenu d'une variable avec un type différent.

Transtypage en Java et en C

```
public class Test {
  public static void main(String [] args) {
    int x1= 243;
    byte x2= (byte) x1;
    System.out.println(x2);
  }
}
```

$$x_1 = -w_{31} . 2^{31} + \sum_{i=0}^{30} w_i . 2^i = 243$$

Le programme affiche "-13".

x1 en mémoire: 00000000000000000000000011110011

x2 en mémoire: 11110011

$$x_2 = -w_7 . 2^7 + \sum_{i=0}^{6} w_i . 2^i = -13$$

```
#include <stdio.h>

int main() {
  char x_c2= -13;
  unsigned char x_ns= (unsigned char) x_c2;
  printf("%d\n", x_ns);
  return 0;
}
```

Le programme affiche "243".

x_c2 en mémoire: 11110011

x_ns en mémoire: 11110011

$$x_{c2} = -w_7 . 2^7 + \sum_{i=0}^{6} w_i . 2^i = -13$$

$$x_{ns} = \sum_{i=0}^{7} w_i . 2^i = 243$$

Table des Matières

Notation des Nombres

- Notation Décimale Positionnelle
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

- Représentation des naturels
- Représentation des entiers
  - Addition
  - Opposé, soustraction
  - Décalages
  - Multiplication
  - Extension signée

Addition en complément à 2

- L'addition de nombres en complément à deux est effectuée de la même façon que l'addition de nombres naturels (non-signés).
- Cela fonctionne car l'addition est effectuée modulo 2^N.

- <u>Exemples</u> (N=4)

```
  111
  0111   (7)
+ 1010   (-6)
 -----
 10001   (1)
```

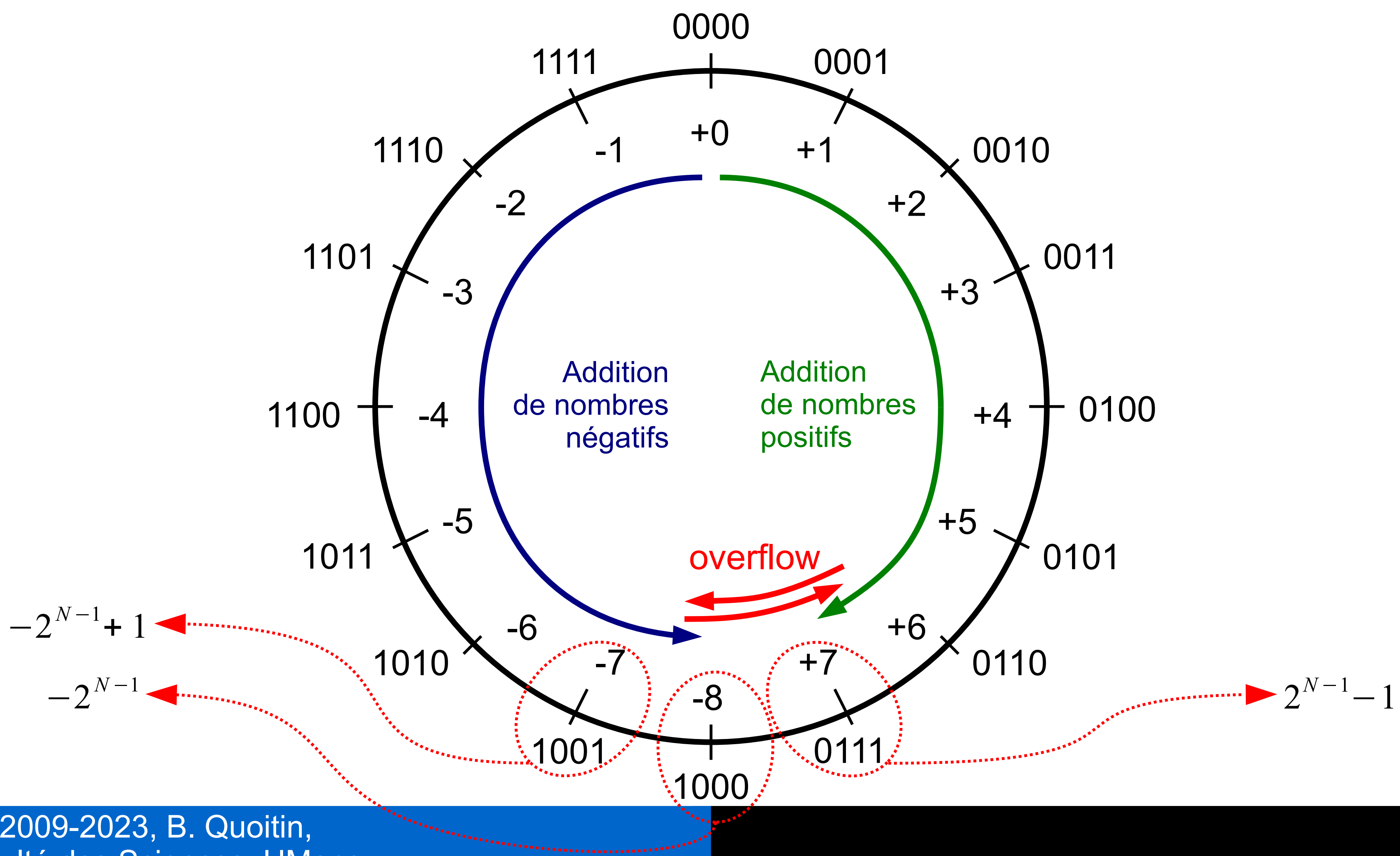
```
  0100   (4)
+ 1001   (-7)
 -----
  1101   (-3)
```

```
  111
  1110   (-2)
+ 1010   (-6)
 -----
 11000   (-8)
```

```
  11
  0110   (6)
+ 1101   (-3)
 -----
 10011   (3)
```

le bit supplémentaire disparaît avec le modulo 2^N

Dépassements – cas N=4



Opérations sur les Entiers

Dépassement en complément à 2

- Il y a dépassement lors de l'addition en complément à 2 si et seulement si les deux conditions suivantes sont vraies
  - x et y sont de même signe
  - la somme est de signe différent

x	y	somme	
> 0	< 0		pas de dépassement
< 0	> 0		pas de dépassement
> 0	> 0	< 0	**dépassement**
> 0	> 0	> 0	pas de dépassement
< 0	< 0	> 0	**dépassement**
< 0	< 0	< 0	pas de dépassement

- Exemples (N=4)

```
                 11
    6           0110
+  -3       +   1101
-----       --------
    3          10011
```

Pas d'overflow :-)

```
                  1
    5           0101
+   6       +   0110
-----       --------
   11           1011  (-5)
```

Overflow :-(

Opérations sur les Entiers

Dépassement en complément à 2

- Explication : un dépassement n'est possible que si les 2 nombres sont de même signe (vérifiez !). Ensuite, on peut traiter séparément les cas où x et y sont tous les deux positifs ou tous les deux négatifs.
- Cas x et y positifs (cas négatif similaire, vérifiez!):

$$0 \leq x, y \leq 2^{N-1} - 1$$
$$0 \leq x + y \leq 2.\left(2^{N-1} - 1\right) = 2^{N} - 2$$

  - Il y a dépassement lorsque $x+y > 2^{N-1}-1$, donc l'intervalle des résultats qui correspondent à un dépassement est

$$2^{N-1} \leq x + y \leq 2^{N} - 2$$

  - Ces valeurs de la somme x+y ont un bit de poids fort qui vaut 1 alors que les bits de poids fort de x et y valent 0.

Addition en complément à 2

- L'addition en complément à 2, notée $+_{c2}$, peut être résumée comme suit. Sur l'intervalle d'entiers $[-2^{N-1}, 2^{N-1}-1]$, elle forme un groupe commutatif.

$$x +_{c2} y = \begin{cases} x + y - 2^N & si \quad x + y \geq 2^{N-1} \\ x + y & si \quad -2^{N-1} \leq x + y < 2^{N-1} \\ x + y + 2^N & si \quad x + y < -2^{N-1} \end{cases}$$

- <u>Exemples</u> avec N=4
  - $x = 3$ $\quad y = 4$ $\quad x + y = 7$ $\quad x +_{c2} y = 7$ (cas 2)
  - $x = 5$ $\quad y = 6$ $\quad x + y = 11$ $\quad x +_{c2} y = -5$ (cas 1)
  - $x = -8$ $\quad y = -8$ $\quad x + y = -16$ $\quad x +_{c2} y = 0$ (cas 3)

Table des Matières

Opposé d'un nombre en complément à 2

- Pour rappel, si un nombre x a la représentation w, alors la représentation de $-x$ est obtenue en complémentant tous les bits de w et en y ajoutant 1.
- Exemple (N=4) :
  - x=-5 est représenté par w=`1011` → `0100` → w'=`0101` représente $-x$=5

- Tous les nombres de l'intervalle $[-2^{N-1}, 2^{N-1}-1]$ ont leur opposé dans le même intervalle, à l'exception de -2^{N-1}. Il s'agit d'une forme de dépassement de capacité.
- Exemple (N=4) :
  - l'opposé de -8 (`1000`) n'appartient pas à [-8, 7]
  - si on applique l'approche ci-dessus, on obtient `0111` en complémentant chaque bit, puis `1000` lorsqu'on ajoute 1.

Soustraction en complément à 2

- La soustraction en complément à 2 se repose sur l'addition en complément à 2. Soustraire y de x revient à additionner x et $-y$.

$$\begin{aligned} x - y &= x + (-y) \\ &= x + (\tilde{y} + 1) \\ &= x + \tilde{y} + 1 \end{aligned}$$

Complémenter les bits de y

Report initial de 1

- <u>Exemple</u> (N=4) :

$$\begin{array}{r} 4 \\ -\;3 \\ \hline 1 \end{array} \qquad \begin{array}{r} 0100 \\ -\;0011 \\ \hline ???? \end{array} \quad \Rightarrow \quad \begin{array}{r} 1\;\;\;1 \\ 0100 \\ +\;1100 \\ \hline 0001 \end{array}$$

Table des Matières

Opérations sur les Entiers

Décalages

- Avec la représentation en complément à 2, des précautions supplémentaires sont nécessaires lorsque l'on effectue des décalages vers la droite.
- En effet, de nouveaux bits de poids fort qui valent 0 sont ajoutés. Si le nombre décalé est négatif (bit de poids fort à 1), le décalage à droite rend le résultat positif !
- Exemple (N=8) :

`11000000` valeur = -64

↓ décalage de 2 bits vers la droite

`00110000` valeur = 48

Opérations sur les Entiers

Décalages

- On distingue par conséquent deux types de décalages vers la droite
  - **décalage logique** (ou décalage non-signé): décale les bits sans se soucier du signe → les nouveaux bits de poids fort valent 0
  - **décalage arithmétique:** conserve le signe → les nouveaux bits de poids fort sont égaux au "bit de signe" du nombre initial

- Exemples

`11000000` valeur = -64

↓ décalage **logique** de 2 bits vers la droite

`00110000` valeur = 48

`11000000` valeur = -64

↓ décalage **arithmétique** de 2 bits vers la droite

`11110000` valeur = -16

Opérations sur les Entiers

Décalages en Java

- Le langage Java offre deux opérateurs distincts pour le décalage vers la droite
  - logique = “**>>>**”
  - arithmétique = “**>>**”

```
public class Test {

  public static void main(String [] args) {
    int x= -63;
    int y= x >> 3;
    int z= x >>> 3;
    System.out.println(y);
    System.out.println(z);
  }

}
```

Le programme affiche “-8” et “536870904”

Représentation de `x` en mémoire:

`11111111111111111111111111000001`

Représentation de `y` en mémoire:

`11111111111111111111111111111000`

Représentation de `z` en mémoire:

`00011111111111111111111111111000`

Table des Matières

Notation des Nombres

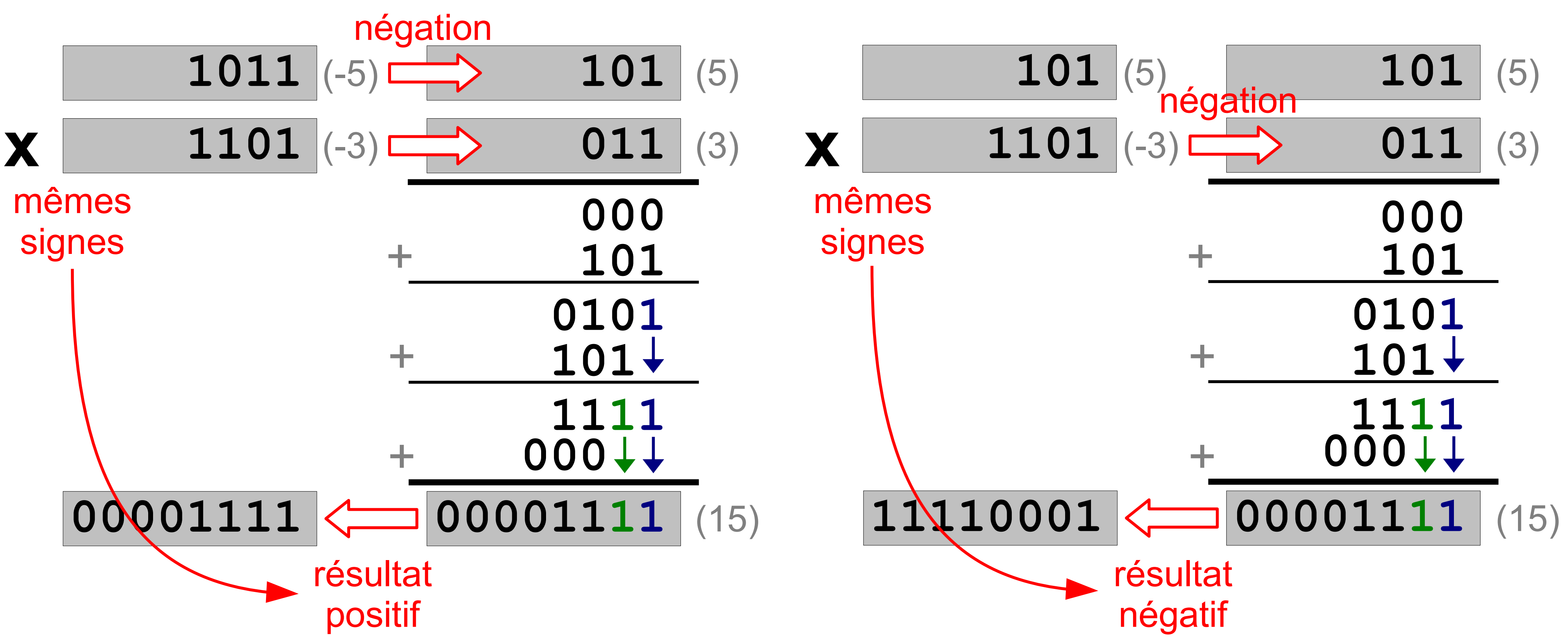
- Notation Décimale Positionnelle
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

- Représentation des naturels
- Représentation des entiers
  - Addition
  - Opposé, soustraction
  - Décalages
  - Multiplication
  - Extension signée

Multiplication en complément à 2

- Une façon d'effectuer la multiplication signée consiste à procéder en valeur absolue, puis de déterminer le signe du résultat sur base des signes des arguments : si signe(x) = signe(y) alors résultat positif, sinon négatif



Multiplication en complément à 2

- En fait, il est possible de multiplier directement 2 nombres représentés en complément à deux sur N bits avec la multiplication non-signée si on ne s'intéresse qu'aux N bits de poids faible du résultat (i.e. résultat modulo 2^N).
- Soit x et y deux mots de N bits.
- Soit x_{c2} et y_{c2}, les valeurs de resp. x et y en complément à 2.
- Soit x_{ns} et y_{ns} les valeurs de resp. x et y en non signé.

$$\begin{aligned}(x_{ns}.y_{ns}) \, mod\, 2^N &= \{(x_{c2}+x_{N-1}.2^N).(y_{c2}+y_{N-1}.2^N)\} \, mod\, 2^N \\ &= \{x_{c2}.y_{c2}+(x_{N-1}.y_{c2}+x_{c2}.y_{N-1}).2^N+x_{N-1}.y_{N-1}.2^N\} \, mod\, 2^N \\ &= (x_{c2}.y_{c2}) \, mod\, 2^N\end{aligned}$$

Ces termes disparaissent avec le modulo 2^N

Multiplication en complément à 2

- Exemple (N=5) :

```
      11011   (-5)          11101   (-3)
x     11101   (-3)          00101   (5)
  ----------            ----------
      11011                 11101
     11011                 11101
  ----------            ----------
    1000111              10010001   (-15)
    11011
  ----------
   101011111
   11011
  ----------
  1100001111  (15)
```

disparaissent avec le modulo 2^5

Table des Matières

Notation des Nombres

- Notation Décimale Positionnelle
- Notation Positionnelle Généralisée
- Changement de Base

Nombres dans un Ordinateur

- Représentation des naturels
- Représentation des entiers
  - Addition
  - Opposé, soustraction
  - Décalages
  - Multiplication
  - Extension signée

Opérations sur les Entiers

Extension signée

- Il est souvent nécessaire de copier la représentation d'un nombre vers un mot plus large. Des précautions sont nécessaires lorsque ce nombre est négatif.

- Exemple :
  - copie d'une variable de N=4 bits vers une variable de M=8 bits.

extension signée

	N=4		M=8	
nombre positif	`0011`	(3)	`00000011`	(3)
nombre négatif	`1101`	(-3)	~~`00001101`~~	(13)
			`11111101`	(-3)

  - La règle qui s'applique est la suivante: les bits de poids fort du mot de taille M sont initialisés avec la valeur du bit de poids fort du mot de taille N.

Opérations sur les Entiers

Extension signée

- Pour une copie vers un mot moins large, il suffit de ne garder que les bits de poids faible. Il faut cependant faire attention aux dépassements de capacité !

- <u>Exemple</u> :
  - copie d'une variable de N=8 bits vers une variable de M=4 bits.

réduction

	N=8		M=4	
nombre positif	00000111	(7)	0111	(7)
nombre négatif	11111001	(-7)	1001	(-7)
nombre trop grand	00001111	(15)	1111	(-1)

Overflow

References

Computer Organization and Design: The Hardware/Software Interface, 4th Edition, D. Patterson and J. Hennessy, Morgan-Kaufmann, 2009

Digital Design: Principles and Practices (3rd Edition), J. Wakerly, Prentice Hall, 2001

Computer Systems: A Programmer's Perspective (2nd edition), R. E. Bryant and D. R. O'Hallaron, Addison-Wesley, 2010

What Every Computer Scientist Should Know About Floating-Point Arithmetic, D. Goldberg, ACM Computing Surveys, Vol 23, No 1, March 1991

A Practical Introduction to Computer Architecture, D. Page, Springer, 2009

Remerciements

Merci à toutes les personnes qui ont permis par leurs remarques de corriger et d'améliorer ces notes de cours.