# *Codage des nombres*

*Dr Zekrifa Djabeur*

# *Représentation de l'information*

- Un ordinateur manipule des données
- Besoin de coder et représenter ces données, pouvant être
  - De nature différente
    - Des nombres
    - Des chaînes de caractères
    - Des informations de tout genre
  - De taille différente
    - Taille fixe de X chiffres : numéro de téléphone, code postal ...
    - De taille variable : nom, adresse, texte, film vidéo ...

# Codage des nombres

- Plusieurs bases de codage possibles
  - Base 10 (décimale) : base de calcul usuelle
  - Base 24 : heures
  - Base 60 : minutes, secondes, degrés
  - Base 12 : douzaine
- Bases les plus utilisées
  - Pour les êtres humains : base décimale
  - Pour un ordinateur
    - Base binaire (2) et dérivées : base hexadécimale (16) ou octale (8)
    - Origine de l'utilisation du binaire : absence ou présence de courant électrique (0 ou 1) comme base de codage

# *Historique*

- Codage des nombres : dans un but de calcul
- Apparition du calcul
  - Dès la préhistoire on comptait avec des cailloux et avec ses doigts
  - Calcul vient du latin *calculi* signifiant caillou
- Antiquité
  - Chaque civilisation (Grecs, Romains, Chinois, Mayas ...) avait développé des
    - Systèmes et bases de numérotation
    - Méthodes pour compter et calculer

# *Historique*

- Origine des systèmes de numérotation
  - Base 10 : nombre des doigts des 2 mains
    - Chiffres romains : V = 5 et X = 10
  - Base 20 : mains et pieds
    - Moins pratique, a disparu ...
  - Base 12 : 3 phalanges et 4 doigts (le pouce sert à positionner le chiffre)
  - Base 60 : les doigts de la deuxième main comptent le deuxième chiffre (60 = 5 x 12)

# *Codage en base B*

- Pour une base *B*, il y a *B* symboles différents (les chiffres de cette base)
  - Base 10 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Base 2 : 0, 1 un chiffre = un bit (*binary digit*)
  - Base 4 : ▲, ◆, ■, ●
  - Base 8 : 0, 1, 2, 3, 4, 5, 6, 7
  - Base 16 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

# *Codage en base B*

- Dans une base *B*, un nombre entier positif *N* s'écrit sous la forme :
  - $(N)_B = a_n\ a_{n-1}\ a_{n-2}\ \dots\ a_1\ a_0$
  - Avec $a_x$ qui est un des *B* chiffres de la base
- Exemples
  - Base décimale : 1234
    - De droite à gauche : chiffre des unités, des dizaines, des centaines, des milliers...
  - Base binaire : 11001
  - Base hexadécimale : 1C04
  - Base 4 : ◆◆■●▲

# Base B vers décimal (addition)

- Valeur en décimal (base 10) d'un nombre « $a_n\ a_{n-1}\ ...\ a_1\ a_0$ » codé dans une base B
  - $a_nB^n + a_{n-1}B^{n-1} + ...\ a_1B + a_0$
  - En prenant la valeur décimale de chaque chiffre $a_x$
- Exemples
  - $(1234)_{10} = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10 + 4$
  - $(11001)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2 + 1$
    $= 16 + 8 + 1 = 25$
  - $(1C04)_{16} = 1 \times 16^3 + 12 \times 16^2 + 0 \times 16 + 4$
    $= 4096 + 12 \times 256 + 0 + 4 = 7172$

  *avec A = 10, B = 11, C = 12, D = 13, E = 14, F = 15*

# *Base B vers décimal (Horner)*

- Schéma de Horner
  - Pour calculer la valeur décimale N de « $a_n a_{n-1} \ldots a_1 a_0$ » codé en base B
    - $P_n = a_n B + a_{n-1}$
      $P_{n-1} = P_n B + a_{n-2}$
      $P_{n-2} = P_{n-1} B + a_{n-3}$
      ...
      $P_1 = P_2 B + a_0 = N$
- Exemple pour $(1234)_{10}$, B=10, n=3
  - $p_3 = a_3 \times B + a_2 = 1 \times 10 + 2 = 12$
    $p_2 = p_3 \times B + a_1 = 12 \times 10 + 3 = 123$
    $p_1 = p_2 \times B + a_0 = 123 \times 10 + 4 = 1234$

# Base B vers décimal (Horner)

- Autres exemples
  - $(11001)_2$, B=2, n=4
    - $p_4 = 1 \times 2 + 1 = 3$
      $p_3 = 3 \times 2 + 0 = 6$
      $p_2 = 6 \times 2 + 0 = 12$
      $p_1 = 12 \times 2 + 1 = 25$
  - $(1C04)_{16,}$ B=16, n=3
    - p3 = 1 x 16 + 12 = 28
      p2 = 28 x 16 + 0 = 448
      p1 = 448 x 16 + 4 = 7172

# *Décimal vers base B*

- On procède par division entière par *B*
- Division du nombre décimal *N* par *B* : donne une valeur $v_0$ et un reste $r_0$
- On divise $v_0$ par *B* : donne $v_1$ et reste $r_1$
- On recommence pour *v1* et ainsi de suite
- Quand $v_x < B$, c'est fini
- $(N)_B = v_x r_{x-1} \ldots r_1 r_0$
- Note : si on continue une étape de plus, $r_{x+1} = v_x$ avec $v_{x+1} = 0$ et on ne peut plus diviser

# *Décimal vers base B*

- Exemple : $(1234)_{10}$ en décimal
  - 1234 / 10 = 123 reste 4
    123 / 10 = 12 reste 3
    12 / 10 = 1 reste 2
    1 < 10 donc on arrête
    Résultat : 1234
  - ```
    1234 | 10
      4  +----+
           123 | 10
             3 +----+
                 12 | 10
                  2 +---
                      1
    ```
    ```

# *Décimal vers base B*

- Exemple : $(25)_{10}$ en binaire
  - 25 / 2 = 12 reste 1
    12 / 2 = 6 reste 0
    6 / 3 = 2 reste 0
    3 / 2 = 1 reste 1
    1 < 2 donc on arrête
    Résultat : $(11001)_2$

- 
```
25 | 2
 1 +-----+
     12 | 2
      0 +----+
          6 | 2
          0 +----+
              3 | 2
              1 +---
                  1
```

# *Décimal vers base B*

- Exemple : $(7172)_{10}$ en hexadécimal
  - 7172 / 16 = 448 reste 4
    448 / 16 = 28 reste 0
    28 / 16 = 1 reste 12 = C
    1 < 16 donc on arrête
    Résultat : $(1C04)_{16}$
  - ```
    7172 | 16
       4  +----+
           448 | 16
             0  +----+
                 28 | 16
                  C +---
                      1
    ```
    ```

# Cas particuliers

- Conversion du binaire à l'octal/hexadécimal ou inverse
  - 1 chiffre octal = un groupe de 3 chiffres binaires
  - 1 chiffre hexadécimal = un groupe de 4 chiffres binaires
  - $(000)_2 = 0$, $(001)_2 = 1$.... $(110)_2 = 6$, $(111)_2 = 7$
    Avec 3 bits on code les 8 chiffres de la base octale
  - $(0000)_2 = 0$, $(0001)_2 = 1$ .... $(1110)_2 = 14 = (E)_{16}$, $(1111)_2 = 15 = (F)_{16}$
    Avec 4 bits, on code les 16 chiffres de la base hexadécimale

# *Cas particuliers*

- Exemple : $(101100001101)_2$ en octal
- On regroupe par groupes de 3 bits :
  <u>*0*10</u> <u>110</u> <u>001</u> <u>101</u>
  - On rajoute des zéros au début au besoin
- $(010)_2 = 2$, $(110)_2 = 6$, $(001)_2 = 1$, $(101)_2 = 5$
- $(10110001101)_2 = (2615)_8$

# *Cas particuliers*

- Exemple : $(101100011101)_2$ en hexadécimal
- On regroupe par groupes de 4 bits :
  <u>*0*011</u> <u>1000</u> <u>1101</u>
- $(0011)_2 = 5$ , $(1000)_2 = 8$, $(1101)_2 = 13$
- $(101100011101)_2 = (58D)_{16}$

# *Cas particuliers*

- Exemple : $(254)_8$ en binaire
  - $2 = (010)_2$, $5 = (101)_2$, $4 = (100)_2$
  - On concatène dans l'autre base ces groupes de 3 bits : $(254)_8 = (10101100)_2$
- Exemple : $(D46C)_{16}$ en binaire
  - $D = 13 = (1101)_2$, $4 = (0100)_2$, $6 = (0110)_2$, $C = 12 = (1100)_2$
  - On concatène dans l'autre base ces groupes de 4 bits : $(D46C)_{16} = (1101010001101100)_2$

# *Codage des nombres*

- On a vu le codage des nombres entiers positifs dans différentes bases
- Mais on doit aussi pouvoir manipuler des
  - Nombres réels
  - Nombres négatifs

# *Codage des nombres réels*

- Codage d'un nombre entier positif en base B :
  $(N)_B = a_n\ a_{n-1}\ a_{n-2}\ ...\ a_1\ a_0$
- Pour coder un nombre réel positif : on rajoute une partie fractionnaire après une virgule
  - $(N)_B = a_n\ a_{n-1}\ ...\ a_1\ a_0\ \mathbf{,}\ b_1\ b_2\ ...\ b_{m-1}\ b_m$
  - La valeur en décimal d'un tel nombre est alors donnée par le calcul de
    - $a_nB^n + a_{n-1}B^{n-1} + ...\ a_1B + a_0 +$
      $b_1B^{-1} + b_2B^{-2} + ...\ b_{m-1}B^{-m+1} + b_mB^{-m}$

# *Conversion réel base B en décimal*

- Exemples :
  - $123{,}45 = 1 \times 10^{2} + 2 \times 10^{1} + 3 \times 10^{0} + 4 \times 10^{-1} + 5 \times 10^{-2}$
  - $(101{,}101)_2 = 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} + 1 \times 2^{-1}$
    $+ 0 \times 2^{-2} + 1 \times 2^{-3}$
    $= 4 + 1 + 0{,}5 + 0{,}125 = 5{,}625$
  - $(AB{,}4E)_{16} = 10 \times 16^{1} + 11 \times 16^{0} + 4 \times 16^{-1} + 14 \times 16^{-2}$
    $= 160 + 11 + 4 \times 0{,}0625 + 14 \times 0{,}00390625$
    $= 171{,}3046875$

# *Conversion réel décimal en base B*

- Conversion d'un nombre décimal réel en base B
  - Pour la partie entière
    - Utiliser la méthode de la division entière comme pour les entiers
  - Pour la partie fractionnaire
    - Multiplier la partie fractionnaire par B
    - Noter la partie entière obtenue
    - Recommencer cette opération avec la partie fractionnaire du résultat et ainsi de suite
    - Arrêter quand la partie fractionnaire est nulle
      - Ou quand la précision souhaitée est atteinte
      - Car on ne peut pas toujours obtenir une conversion en un nombre fini de chiffres pour la partie fractionnaire
    - La partie fractionnaire dans la base B est la concaténation des parties entières obtenues dans l'ordre de leur calcul

# Conversion réel décimal en base B

- Exemple : conversion de 12,6875 en binaire
  - Conversion de 12 : donne $(1100)_2$
  - Conversion de 0,6875
    - ```
      0,6875 x 2 = 1,375 = 1 + 0,375
      0,375  x 2 = 0,75  = 0 + 0,75
      0,75   x 2 = 1,5   = 1 + 0,5
      0,5    x 2 = 1     = 1 + 0
      ```
    - $(12,6875)_{10} = (1100,1011)_2$
- Exemple : conversion de 171,3046875 en hexadécimal
  - Conversion de 171 : donne $(AB)_{16}$
  - Conversion de 0,3046875
  - ```
    0,3046875 x 16 = 4,875 = 4 + 0,875
    0,875     x 16 = 14,0  = 14 + 0
    ```
  - $(171,3046875)_{10} = (AB,4E)_{16}$

# Codage des nombre réels en virgule flottante

- Principe et intérêts
  - Avoir une virgule flottante et une précision limitée
  - Ne coder que des chiffres significatifs
  - $N = +/- M \times B^{E}$
    - N = nombre codé
    - M = mantisse : nombre de *X* chiffres de la base *B*
    - E = exposant : nombre de *Y* chiffres de la base *B*
    - +/- = codage du signe : positif ou négatif
  - Le nombre est présenté sous forme normalisée pour déterminer la mantisse et exp.
    - Pas de chiffre avant la virgule : $0,XXXXX \times B^{E}$

# *Codage des nbs réels en virgule flottante*

- Exemple : 1234,5 en base 10
  - On normalise pour n'avoir que des chiffres après la virgule : $0{,}12345 \times 10^4$
  - Mantisse codée = 12345, exposant = 4, signe = +
- Standard IEEE 754 : codage binaire de réels en virgule flottante
  - Précision simple : 32 bits
    1 bit de signe, 8 bits exposant, 23 bits mantisse
  - Précision double : 64 bits
    1 bit de signe, 11 bits exposant, 52 bits mantisse
  - Précision étendue : sur 80 bits

# *Codage des entiers signés en binaire*

- Codage des entiers signés en binaire :
  trois méthodes
  - Utiliser un bit de signe et coder la valeur absolue
  - La méthode du complément logique
  - La méthode du complément arithmétique
- Pour toutes ces solutions
  - On aura toujours un bit utilisé pour préciser le signe du nombre

# *Entier signé binaire : méthode du signe et valeur absolue*

- Principe : considérer que le bit de poids fort code le signe
  - 0 = entier positif, 1 = entier négatif
  - Bit de poids fort : le plus à gauche
    - Bit de poids faible : le plus à droite
  - Les autres bits codent le nombre en valeur absolue
    - Nécessité de savoir sur combien de bits on code le nombre pour déterminer quel bit code quoi
- Exemples si codage sur 4 bits
  - $(\mathbf{0}111)_2$ = 7 car bit de poids fort à 0
  - $(\mathbf{1}111)_2$ = -7 car bit de poids fort à 1

# *Codage sur n bits*

- Un ordinateur manipule des nombres binaires sous forme de 8 bits = un octet
- Données codées sur un ou plusieurs octets :
  8 bits, 16 bits, 32 bits, 64 bits ...
  - Avec *p* bits, on code *N* valeurs avec $0 \leqslant N \leqslant 2^p - 1$
  - Avec 16 bits, on peut coder $2^{16} = 65536$ valeurs différentes
  - Soit les entiers de 0 à 65535 pour les entiers positifs

# *Codage sur* n *bits : entiers signés*

- Pour un entier signé sur 16 bits :
  - Nombres positifs : 0XXXXXXXXXXXXXXX
  - Nombres négatifs : 1XXXXXXXXXXXXXXX
- On a 15 bits pour coder la valeur absolue du nombre soit $2^{15} = 32768$ valeurs possibles
  - Pour le positif : de 0 à 32767
  - Pour le négatif : de -0 à -32767
- Pour *p* bits : $-(2^{p-1} - 1) \leqslant N \leqslant 2^{p-1} - 1$
- Inconvénient : on code 2 fois le 0

# *Entier signé binaire : complément à 1*

- Complément logique d'un nombre binaire
  - Les 1 deviennent 0 et les 0 deviennent 1
  - Complément logique est dit « complément à 1 »
- Codage des nombres signés avec complément logique
  - Nb positif : comme pour un entier non signé
  - Nb négatif : complément logique de son opposé positif
  - Bit de poids fort code le signe : 0 = positif, 1 = négatif
- Exemple, codage sur un octet :
  - $(00000111)_2 = 7$
  - Complément à 1 : $(11111000)_2 = -7$ (et pas 248)
- Inconvénient : toujours 2 façons de coder le 0

# *Entier signé binaire : complément à 2*

- Complément arithmétique
  - Complément logique du nombre auquel on rajoute la valeur de 1
  - Dit « complément à 2 »
- Codage nombres signés avec complément arithmétique
  - Nb positif : comme pour un entier non signé
  - Nb négatif : complément arithmétique de son opposé positif
  - Bit de poids fort code le signe : 0 = positif, 1 = négatif
- Exemple : 6 = $(0110)_2$ avec précision de 4 bits
  - Complément à 1 : 1001
  - Complément à 2 : 1001 + 1 = 1010

# *Entier signé binaire : complément à 2*

- Pour *p* bits, on code $-2^{p-1} \leqslant N \leqslant 2^{p-1} - 1$ valeurs
  - Sur 16 bits : $-32768 \leqslant N \leqslant 32767$
- Ce codage est le plus utilisé, c'est le standard de fait pour coder les entiers signés
- Intérêts
  - Plus qu'une seule façon de coder le 0
    - Grace au « +1 » qui décale l'intervalle de codage des négatifs
  - Facilite les additions/soustractions en entier signé
- Propriétés du complément à 2
  - $comp_2(N) + N = 0$
  - $comp_2(comp_2(N)) = N$

# *Entiers signés en binaire : résumé*

- Exemple pour codage de -57 pour les 3 méthodes, sur 8 bits
  - $57 = (00111001)_2$
  - Signe et valeur absolue : 10111001
  - Complément à 1 : 11000110
  - Complément à 2 : 11000111
- Dans tous les cas
  - Si bit de poids fort = 0 : entier positif
  - Si bit de poids fort = 1 : entier négatif

# *Complément sur les compléments*

- Compléments arithmétique et logique
  - Utilisables dans n'importe quelle base, pas que en binaire
  - Avec les mêmes propriétés dans toute base
- Complément logique d'un nombre N en base B
  - Nombre pour lequel chaque chiffre $a_x$ de N est remplacé par le chiffre de valeur $B - 1 - a_x$
  - Exemple en base 8 : $comp_{log}$ (235) = 542
- Complément arithmétique = complément logique + 1
  - Exemple en base 8 : $comp_{ari}$ (235) = 542 + 1 = 543
  - Rajoute la valeur 1 quelle que soit la base considérée

# *Calculs dans une base B*

- Les opérations arithmétiques (addition, sous-traction, multiplication, division) sont réalisables dans toute base B
  - Avec mêmes règles que pour la base décimale
  - Retenues également mais dépendant de la base
  - Quand on additionne 2 chiffres *a* et *b* dans la base B
    - Si la somme des valeurs décimales de *a* et *b* dépasse ou égale B alors il y a une retenue
- Exemple : principes de l'addition binaire
  - 0 + 0 = 0
    0 + 1 = 1
    1 + 1 = 10  soit 0 avec une retenue de 1

# *Addition binaire*

- Exemple : 10 + 1011 :

```
    0010    = 2
  + 1011    = 11
   ------
    1101    = 13
```

- Autre exemple : 1101 + 1010 :

```
    1101    = 13
  + 1010    = 10
  -------
   10111    = 23
```

  - Addition de 2 nombres de 4 bits : on a besoin dans cet exemple de 5 bits
  - Potentiel problème de débordement

# *Débordement*

- Débordement : la taille allouée (8, 16 ... bits) au codage d'un entier est trop petite pour coder ou stocker le résultat d'un calcul
- Exemple avec addition, sur 8 bits, non signé :
  - 10110011 + 10000101 = 1$\underline{00111000}$
  - Besoin de 9 bits pour coder le nombre
  - Stockage du résultat impossible sur 8 bits
- Exemple avec addition, sur 8 bits, signé :
  - **0**1110011 + **0**1000101 = **1**0111000
  - Addition de 2 positifs donne un négatif !

# *Multiplication binaire*

- Comme en décimal
- N'utilise que du décalage de bits et additions
- Exemple : 101 x 110 :

```
      101 = 5
  x   110 = 6
  -------
      000
+    1010
+   10100
-------
    11110 = 30
```

- Décalage d'un bit vers la gauche = multiplication par 2
- Décalage d'un bit vers la droite = division entière par 2

# *Soustraction binaire*

- Soustraction binaire : peut faire comme en décimal
  - Exemple : 1101 – 1011

```
   1101 = 13
-  1011 = 11
  ------
   0010 = 2
```

- Autre technique
  - Utiliser les compléments à 2 et ne faire que des additions

# *Addition/soustraction binaire en signé*

- Codage en complément à 2
  - Simplifie les additions et soustractions
  - On peut additionner directement des nombres, quels que soient leurs signes, le résultat sera directement correct (si pas de débordement) et « bien codé »
  - Soustraction d'un nombre = addition de son complément à 2
    - $A – B = A + comp_2 (B)$
    - Valable dans tous les cas, quels que soient les signes de A et B
    - Là aussi le résultat est directement valide si pas de débordement