Compléments sur les représentations numériques signées

Entiers signés

- La convention usuelle pour représenter les valeurs entières toujours positives (non signées) dite positionnelle base 2

Pour l'octet composé des 8 bits suivants :

$$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$$

Valeur non signée (u8) =

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$$

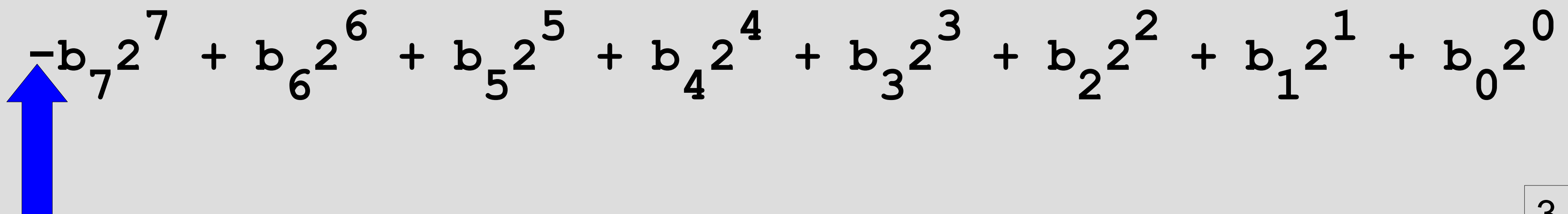
Entiers signés

- La convention usuelle pour représenter les valeurs entières positives ou négatives (signées) est dite "complément à 2"
- Le coef de poids le plus fort est négatif

Pour l'octet composé des 8 bits suivants :

$$b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0$$

Valeur signée (s8) =



Entiers signés

- La même information binaire a donc une interprétation signée et une interprétation non signée
- Quand le bit de poids fort est nul les 2 interprétations concordent

$$(01101010)_2$$

Non signé :

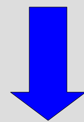
$(u8) = 0.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 0.2^0 = (106)_{10}$

Signé :

$(s8) = -0.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 0.2^0 = (106)_{10}$

Entiers signés

- Quand le bit de poids fort est à 1 les 2 interprétations ne concordent plus
- Quand le bit de poids fort est à 1 l'interprétation signée est négative : le bit de poids fort est le **bit de signe**



$$(\mathbf{1}1101010)_2$$

Non signé :

(u8) = $1.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 0.2^0 = (234)_{10}$

Signé :

(s8) = $-1.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 0.2^0 = (-22)_{10}$

Entiers signés

- Intérêt de cette convention pour les signés :
  - Il n'y a qu'une manière de représenter 0
  - Un seul bit à tester pour savoir si valeur négative
  - Représentation compatible avec l'addition binaire usuelle

```
     Binaire        Hexa      u8       s8
       ,,
  0b01001110        0x4E      78       78
+ 0b00100101      + 0x25    + 37     + 37
  0b01110011        0x73     115      115


      ,,,,
  0b01001110        0x4E      78       78
+ 0b10011111      + 0x9F    +159    + -97
  0b11101101        0xED     237      -19
```

Entiers signés

- Pour trouver l'opposé d'une valeur en signé on prend le **complément à 2**
- C'est le complément à 1 (NOT)   +1

```
Départ             0b11101101    0xED   -19
  On inverse
Complément à 1     0b00010010    0x12   +18
  On ajoute 1
Complément à 2     0b00010011    0x13   +19


Départ             0b00000110    0x06   +6
  On inverse
Complément à 1     0b11111001    0xF9   -7
  On ajoute 1
Complément à 2     0b11111010    0xFA   -6
```

Entiers signés

- Il faut travailler dans toute la largeur de la variable
- Exemples jusqu'à maintenant sur 8 bits (char)
- Si on est en **int** ou **unsigned int** il faut être en 32 bits :
- Exemple complément à 2 de la valeur 1 sur 32 bits

Départ	**0b0000...0001**	**0x00000001**	**+1**
Compl. 1	**0b1111...1110**	**0xFFFFFFFE**	**-2**
Compl. 2	**0b1111...1111**	**0xFFFFFFFF**	**-1**

```
unsigned int a=0xFFFFFFFF;
printf("%d\n",a); // affiche -1
printf("%u\n",a); // affiche 4294967295
```

Entiers signés

- La position variable du bit de poids fort selon le format de stockage pose des problèmes de transtypage ...
- Pour les **non signés**, passer d'un stockage compact à un stockage large se fait simplement en complétant avec 0

```
Départ sur un octet

  0b01101010
= 0x6A = 106(u8)

Arrivée sur 4 octets

  0b00000000000000000000000001101010
= 0x0000006A = 106(u32)

Résultat OK
```

Entiers signés

- Pour les **signés**, passer d'un stockage compact à un stockage large se fait en complétant avec 0...
quand le nombre est positif (bit de signe à 0)

```
Départ sur un octet

  0b01101010
= 0x6A = 106(s8)

Arrivée sur 4 octets

  0b00000000000000000000000001101010
= 0x0000006A = 106(s32)

Résultat OK
```

Entiers signés

- Pour les **signés**, passer d'un stockage compact à un stockage large **ne peut pas se faire en complétant avec 0** quand le nombre est négatif (bit de signe à 1)

```
Départ sur un octet

  0b11101010
= 0xEA = -22(s8)

Arrivée sur 4 octets

  0b0000000000000000000000000011101010
= 0x000000EA = 234(s32)

Résultat non conforme !
```

Entiers signés

- Pour les **signés**, passer d'un stockage compact à un stockage large nécessite de propager le bit de signe à gauche (mettre des 0 si 0, mettre des 1 si 1)
- C'est l'**extension du bit de signe**

```
Départ sur un octet

  0b11101010
= 0xEA = -22(s8)

Arrivée sur 4 octets

  0b11111111111111111111111111101010
= 0xFFFFFFEA = -22(s32)

Résultat OK !
```

Entiers signés

- En C tout ceci est géré automatiquement à la compilation à partir des types déclarés pour les variables ...

```
unsigned char uc=0xEA;
unsigned int  ui;
signed   char sc=0xEA;
signed   int  si;

ui=uc;
printf("%08X\n",ui); // affiche 000000EA

si=sc; // extension du bit de signe
printf("%08X\n",si); // affiche FFFFFFEA
```

Entiers signés

- En C tout ceci est géré automatiquement à la compilation mais il faut comprendre ce qu'on fait ...

```
char chaine[]="éléphant";
int val_ascii;
int i,histo[256]={0};

for (i=0;i<strlen(chaine);i++)
{
   val_ascii=chaine[i];
   histo[val_ascii]++;  // Débordement
   printf("%d ",val_ascii);
}
// affiche -23 108 -23 112 104 97 110 116
```

Entiers signés

- En assembleur les données qui passent par les registres ne sont pas typées
- Il existe bcp d'instructions mixtes signé/non signé (add, sub)
- Sinon il faut choisir la bonne instruction...
- En particulier il existe 2 types de décalage à droite
  - Décalage logique à droite LSR (pour les non signés)
  - Décalage arithmétique à droite ASR (pour les signés)
    Le bit de signe est alors dupliqué par la gauche
    au lieu de compléter par des 0
  - Ces 2 décalages de n bits correspondent bien à une division par 2^n respectivement pour *unsigned* ou *signed*