Représentation des entiers

Prof d'info

Lycée Thiers

Crédits

- Wikipédia : Complément à deux et ce Wiki
- cpprefrence (Fonction de bibliothèques en C)
- Les cours de mes collègues Nicolas Pécheux et Quentin Fortier.

Boutisme

- En informatique, certaines données telles que les nombres entiers peuvent être représentées sur plusieurs octets. L'ordre dans lequel ces octets sont organisés en mémoire ou dans une communication est appelé endianness (mot anglais traduit par « boutisme » ou par « endianisme »).

Boutisme

- En informatique, certaines données telles que les nombres entiers peuvent être représentées sur plusieurs octets. L'ordre dans lequel ces octets sont organisés en mémoire ou dans une communication est appelé endianness (mot anglais traduit par « boutisme » ou par « endianisme »).
- De la même manière que certains langages humains s'écrivent de gauche à droite, et d'autres s'écrivent de droite à gauche, il existe une alternative majeure à l'organisation des octets représentant une donnée : l'orientation *big-endian* et l'orientation *little-endian*.

Boutisme

- En informatique, certaines données telles que les nombres entiers peuvent être représentées sur plusieurs octets. L'ordre dans lequel ces octets sont organisés en mémoire ou dans une communication est appelé endianness (mot anglais traduit par « boutisme » ou par « endianisme »).
- De la même manière que certains langages humains s'écrivent de gauche à droite, et d'autres s'écrivent de droite à gauche, il existe une alternative majeure à l'organisation des octets représentant une donnée : l'orientation *big-endian* et l'orientation *little-endian*.
- En maths, la convention d'écriture des polynômes est le big-endian (ou *mot de poids fort en tête*) comme dans $X^3 + X^2 + 1$.

Gulliver

- Les termes *big-endian* et *little-endian* ont été popularisés dans le domaine informatique par Dany Cohen, en référence aux ≪ Voyages de Gulliver ≫, roman satirique de Jonathan Swift.

Gulliver

- Les termes *big-endian* et *little-endian* ont été popularisés dans le domaine informatique par Dany Cohen, en référence aux ≪ Voyages de Gulliver ≫, roman satirique de Jonathan Swift.
- En 1721, Swift décrit comment de nombreux habitants de Lilliput refusent d'obéir à un décret obligeant à manger les œufs à la coque par le petit bout.

Gulliver

- Les termes *big-endian* et *little-endian* ont été popularisés dans le domaine informatique par Dany Cohen, en référence aux « Voyages de Gulliver », roman satirique de Jonathan Swift.
- En 1721, Swift décrit comment de nombreux habitants de Lilliput refusent d'obéir à un décret obligeant à manger les œufs à la coque par le petit bout.
- La répression pousse les rebelles, dont la cause est appelée big-endian, à se réfugier dans l'empire rival de Blefuscu ce qui entretient une guerre longue et meurtrière entre les deux empires.

FIGURE – un oeuf

Big-endian (gros-boutisme)

- Soit un entier sur 32 bits à écrire en en mémoire, par exemple $0\mathtt{x}A0B70708$ en notation hexadécimal. Pour une structure de mémoire fondée sur une unité atomique de 1 octet et un incrément d'adresse de 1 octet, la convention big-endian consiste à enregistrer `A0` à l'adresse mémoire la plus petite et 08 à la plus grande.

...	$A0$	$B7$	07	08	...

Big-endian (gros-boutisme)

- Soit un entier sur 32 bits à écrire en en mémoire, par exemple $0\mathtt{x}A0B70708$ en notation hexadécimal. Pour une structure de mémoire fondée sur une unité atomique de 1 octet et un incrément d'adresse de 1 octet, la convention big-endian consiste à enregistrer `A0` à l'adresse mémoire la plus petite et 08 à la plus grande.

...	$A0$	$B7$	07	08	...

- Tous les protocoles TCP/IP communiquent en big-endian. Il en va de même pour le protocole PCI Express.
  Les processeurs Motorola 68000, les SPARC (Sun Microsystems), les System/370 (IBM) sont des architectures qui respectent cette règle.

Little-endian (petit-boutisme)

- En little-endian (≪ mot de poids faible en tête ≫), le nombre `0xA0B70708` est enregistré, pour une structure de mémoire fondée sur une unité atomique de 1 octet et un incrément d'adresse de 1 octet, avec `08` à l'adresse mémoire la plus petite, `A0` à la plus grande.

...	08	07	*B*7	*A*0	...

Little-endian (petit-boutisme)

- En little-endian (≪ mot de poids faible en tête ≫), le nombre `0xA0B70708` est enregistré, pour une structure de mémoire fondée sur une unité atomique de 1 octet et un incrément d'adresse de 1 octet, avec `08` à l'adresse mémoire la plus petite, `A0` à la plus grande.

...	08	07	*B*7	*A*0	...

- Par exemple, les processeurs x86 ont une architecture petit-boutiste.

Little-endian (petit-boutisme)

- L'inconvénient du little-endian est la moindre lisibilité du code machine par le programmeur. Son intérêt réside dans la plus grande rapidité des opérations arithmétiques.

Little-endian (petit-boutisme)

- L'inconvénient du little-endian est la moindre lisibilité du code machine par le programmeur. Son intérêt réside dans la plus grande rapidité des opérations arithmétiques.
- Dans ce cours, nous représentons les nombres de façon plus lisible pour le programmeur, c'est à dire selon la convention big-endian (bit de poids fort en tête).

Base 2

- Un nombre en base 2 peut-être vu comme un état de case mémoire. La base 2 est de plus très pratique pour les calculs arithmétiques.

Base 2

- Un nombre en base 2 peut-être vu comme un état de case mémoire. La base 2 est de plus très pratique pour les calculs arithmétiques.
- Nombres entiers : Couramment stockés sur 32 ou 64 bits. Dans les exemples ci-dessous : souvent sur 8 bits (pour des raisons de place).

Base 2

- Un nombre en base 2 peut-être vu comme un état de case mémoire. La base 2 est de plus très pratique pour les calculs arithmétiques.
- Nombres entiers : Couramment stockés sur 32 ou 64 bits. Dans les exemples ci-dessous : souvent sur 8 bits (pour des raisons de place).
- Sur 8 bits, entiers entre 0 et $2^8 - 1 = 255$. Sur N bits, on représente tous les entiers de 0 à $2^N - 1$.

Base 2

- Un nombre en base 2 peut-être vu comme un état de case mémoire. La base 2 est de plus très pratique pour les calculs arithmétiques.
- Nombres entiers : Couramment stockés sur 32 ou 64 bits. Dans les exemples ci-dessous : souvent sur 8 bits (pour des raisons de place).
- Sur 8 bits, entiers entre 0 et $2^8 - 1 = 255$. Sur N bits, on représente tous les entiers de 0 à $2^N - 1$.
- Un *octet* = 8 bits. Avec un octet on représente 256 nombres : le plus souvent sur $[\![0, 255]\!]$ ou $[\![-128, 127]\!]$.

Base 2

- Un nombre en base 2 peut-être vu comme un état de case mémoire. La base 2 est de plus très pratique pour les calculs arithmétiques.
- Nombres entiers : Couramment stockés sur 32 ou 64 bits. Dans les exemples ci-dessous : souvent sur 8 bits (pour des raisons de place).
- Sur 8 bits, entiers entre 0 et $2^8 - 1 = 255$. Sur N bits, on représente tous les entiers de 0 à $2^N - 1$.
- Un *octet* = 8 bits. Avec un octet on représente 256 nombres : le plus souvent sur $[\![0, 255]\!]$ ou $[\![-128, 127]\!]$.
- En base 2 sur 32 bits, on représente les entiers de 0 à $2^{32} - 1 = 4294967295$ (4 milliards environ).

Conventions

- Pour distinguer cent un en base 10, du nombre cinq écrit en base 2, on indice l'écriture binaire. 101 (en base 10) ou 101_2 (en base 2).
- Pour les autres bases, on indique le nombre de symboles, par exemple 432_5 représente un nombre en base 5.

Passage de la base 10 à la base k

```
1  fonction changement_base(n,k)
2          entree : n (le nb)
3          entree : k (la base)
4          sortie : a, suite des coefs de n en base k
5          a := suite vide
6          tant_que n!=0 :
7                debut
8                ajouter à a le reste de la division de n par k
9                n := quotient de la division de n par k
10               fin
11         Inverser la suite a /*pour Big-Endian*/
12         renvoyer la suite des restes
```

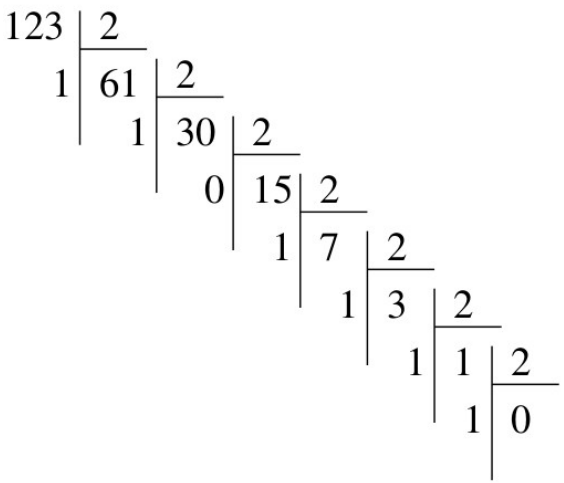
Exemple

Exercice

Écrire 123 en binaire.

$$123 = 111\,1011_2$$

FIGURE – Méthode 1



Base deux à dix

1111011_2 représente 123 :

$$1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 123.$$

Addition en binaire

- Principe :
  - $0 + 0 = 0$
  - $0 + 1 = 1$
  - $1 + 0 = 1$
  - $1 + 1 = 0$ (avec retenue)
- Exemple :

```
1         *  *           *  *       (* : retenue)
2      1  0  1  1  1  1  0  1  1
3   +        1  1  0  0  0  0  1
4   ---------------------------
5   =  1  1  1  0  1  1  1  0  0
```

Ceci est cablé dans le processeur dans la partie réservée aux calculs arithmétiques et logiques (UAL).

Soustraction en binaire

- Principe :
  - 0 - 0 = 0. Retenue : non
  - 0 - 1 = 1 Retenue : oui
  - 0 - (1+1) = 0, où le 1 rouge est la retenue. Retenue : oui
  - 1 - (1+1) = 1, où le 1 rouge est la retenue. Retenue : oui
  - 1 - 0 = 1. Retenue : non
  - 1 - 1 = 0. Retenue : non
- Exemple :

```
1        *     * * *        (* : retenue)
2     1 1 0 1 1 1 0
3  -        1 0 1 1 1
4  ---------------------
5  = 1 0 1 0 1 1 1
```

La retenue est à ajouter aux chiffres sur la ligne du bas.

Exemple : 100 - 011 = 001.

Présentation

- Un entier non signé (`unsingned int`) est un entier positif, qui est stocké en mémoire avec sa représentation en base 2 sans interprétation particulière de son bit de poids fort.

Présentation

- Un entier non signé (`unsingned int`) est un entier positif, qui est stocké en mémoire avec sa représentation en base 2 sans interprétation particulière de son bit de poids fort.
- Si les entiers sont codés sur 4 octets, alors le plus grand entier non signé est $2^{32} - 1$ soit 4294967295.

Présentation

- Un entier non signé (`unsingned int`) est un entier positif, qui est stocké en mémoire avec sa représentation en base 2 sans interprétation particulière de son bit de poids fort.
- Si les entiers sont codés sur 4 octets, alors le plus grand entier non signé est $2^{32} - 1$ soit 4294967295.
- Le comportement de ces nombres est cyclique.
  Conseil : on ne doit les utiliser que si ce comportement cyclique est attendu. Et il faut surtout éviter de mélanger les types signés et non signés dans un même calcul à moins d'avoir bien lu (l'indigeste) norme !

Présentation

- Un entier non signé (`unsingned int`) est un entier positif, qui est stocké en mémoire avec sa représentation en base 2 sans interprétation particulière de son bit de poids fort.
- Si les entiers sont codés sur 4 octets, alors le plus grand entier non signé est $2^{32} - 1$ soit 4294967295.
- Le comportement de ces nombres est cyclique.
  Conseil : on ne doit les utiliser que si ce comportement cyclique est attendu. Et il faut surtout éviter de mélanger les types signés et non signés dans un même calcul à moins d'avoir bien lu (l'indigeste) norme !
- Ces nombres ne peuvent être négatifs, un nombre négatif est donc automatiquement converti en son équivalent modulo 2^{32}

Exemple

- Le code suivant :

```
1 unsigned x = 42;
2 printf("%u\n",x); // %u pour unsigned int
3 x = -1; printf("%u\n",x); // affiche le + grand
    nombre
4 x=4294967295; printf("%u\n",x+1);
5
```

Exemple

- Le code suivant :

```c
unsigned x = 42;
printf("%u\n",x); // %u pour unsigned int
x = -1; printf("%u\n",x); // affiche le + grand
    nombre
x=4294967295; printf("%u\n",x+1);

```

- produit :

```
42
4294967295
0
```

Conversion automatique de types

- La somme d'un entier non signé et d'un entier signé est convertie en un entier signé.
- Que donne l'affichage de ce programme ?

```
1 unsigned a = 0;
2 int b = -1;
3 if (a+b>0)
4   printf("Hello\n");// Mais combien vaut a+b ?
5 else
6   printf("Coucou\n");
7
```

- Perdu : c'est "Hello" !

Conversion automatique de types

- Lorsqu'on compare un entier signé et non signé, le signé est converti en non signé :
- Que donne l'affichage de ce programme ?

```
1 unsigned a = 0;
2 int b = -1;
3 if (a>b)
4   printf("Hello\n");
5 else
6   printf("Coucou\n");
7
```

- Perdu : c'est "Coucou" ! En effet, -1 est converti en $2^{32} - 1$...

En travaux

Mettre le signe en bit de poids fort

- Sur 8 bits on représente les entiers non signés de 0 à 255.

Mettre le signe en bit de poids fort

- Sur 8 bits on représente les entiers non signés de 0 à 255.
- Sur 8 bits on voudrait maintenant représenter les nombres de -128 à 127.

Mettre le signe en bit de poids fort

- Sur 8 bits on représente les entiers non signés de 0 à 255.
- Sur 8 bits on voudrait maintenant représenter les nombres de -128 à 127.
- Notation utilisée sur des écritures de nombres de longueur donnée (8,16,32 bits). Bit de poids fort du nombre pour le signe.

Mettre le signe en bit de poids fort

- Sur 8 bits on représente les entiers non signés de 0 à 255.
- Sur 8 bits on voudrait maintenant représenter les nombres de -128 à 127.
- Notation utilisée sur des écritures de nombres de longueur donnée (8,16,32 bits). Bit de poids fort du nombre pour le signe.
- Première idée $00000010_2 = +2$ en décimal et $10000010_2 = -2$ en décimal. PB :

Mettre le signe en bit de poids fort

- Sur 8 bits on représente les entiers non signés de 0 à 255.
- Sur 8 bits on voudrait maintenant représenter les nombres de -128 à 127.
- Notation utilisée sur des écritures de nombres de longueur donnée (8,16,32 bits). Bit de poids fort du nombre pour le signe.
- Première idée $00000010_2 = +2$ en décimal et $10000010_2 = -2$ en décimal. PB :
  1. Le nombre 0 possède deux représentations 10000000_2 et 00000000_2 (0 et -0).

Mettre le signe en bit de poids fort

- Sur 8 bits on représente les entiers non signés de 0 à 255.
- Sur 8 bits on voudrait maintenant représenter les nombres de -128 à 127.
- Notation utilisée sur des écritures de nombres de longueur donnée (8,16,32 bits). Bit de poids fort du nombre pour le signe.
- Première idée $00000010_2 = +2$ en décimal et $10000010_2 = -2$ en décimal. PB :
  1. Le nombre 0 possède deux représentations 10000000_2 et 00000000_2 (0 et -0).
  2. Il faudrait modifier l'algorithme d'addition. Si un des nombres est négatif : erreur. Ainsi $3 + (-4) = -1$ Mais

$$00000011_2 + 10000100_2 = 10000111_2 \rightarrow -7.$$

Représentation des entiers signés en complément à 2 sur 8 bits

- Prenons l'exemple des mots de 8 bits : on peut représenter les entiers relatifs compris entre $-2^{8-1} = -128$ et $2^{8-1} - 1 = 127$

Représentation des entiers signés en complément à 2 sur 8 bits

- Prenons l'exemple des mots de 8 bits : on peut représenter les entiers relatifs compris entre $-2^{8-1} = -128$ et $2^{8-1} - 1 = 127$
- entier relatif x positif ou nul : pas de changement.

Représentation des entiers signés en complément à 2 sur 8 bits

- Prenons l'exemple des mots de 8 bits : on peut représenter les entiers relatifs compris entre $-2^{8-1} = -128$ et $2^{8-1} - 1 = 127$
- entier relatif x positif ou nul : pas de changement.
- entier relatif x strictement négatif : représenté par l'entier naturel $x + 2^8 = x + 256$, qui est compris entre 128 et 255.

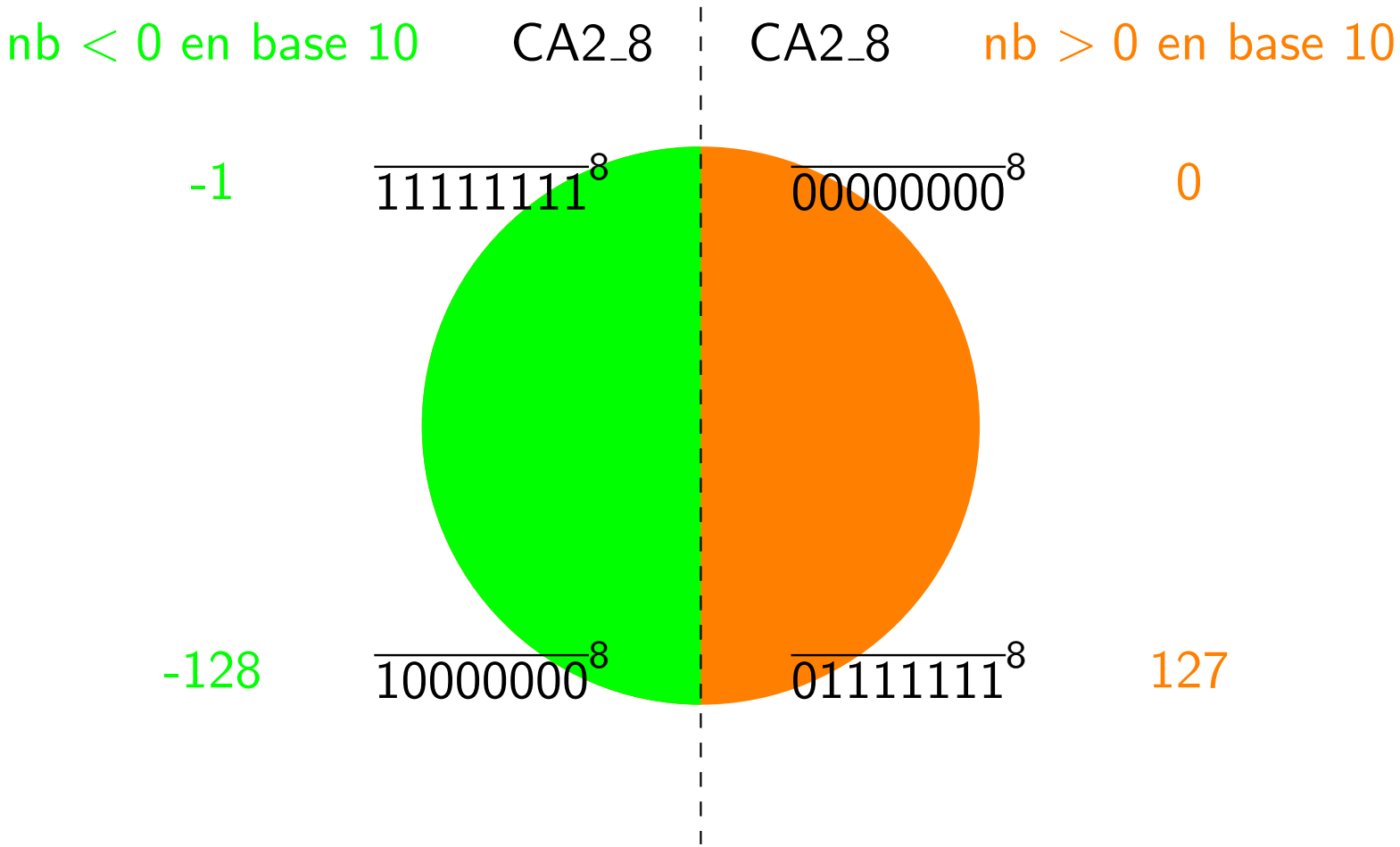
Représentation des entiers signés en complément à 2 sur 8 bits

- Prenons l'exemple des mots de 8 bits : on peut représenter les entiers relatifs compris entre $-2^{8-1} = -128$ et $2^{8-1} - 1 = 127$
- entier relatif x positif ou nul : pas de changement.
- entier relatif x strictement négatif : représenté par l'entier naturel $x + 2^8 = x + 256$, qui est compris entre 128 et 255.
- Ainsi les entiers naturels de 0 à 127 servent à représenter les entiers relatifs positifs ou nul

Représentation des entiers signés en complément à 2 sur 8 bits

- Prenons l'exemple des mots de 8 bits : on peut représenter les entiers relatifs compris entre $-2^{8-1} = -128$ et $2^{8-1} - 1 = 127$
- entier relatif x positif ou nul : pas de changement.
- entier relatif x strictement négatif : représenté par l'entier naturel $x + 2^8 = x + 256$, qui est compris entre 128 et 255.
- Ainsi les entiers naturels de 0 à 127 servent à représenter les entiers relatifs positifs ou nul
- et les entiers naturels de 128 à 255 servent à représenter les entiers relatifs strictement négatifs

Complément à 2 sur 8 bits



Représentation des entiers signés en complément à 2 sur N bits (CA2_N)

Méthode 1

- On veut représenter un nombre $-2^{N-1} \leq X < 2^{N-1}$ en CA2_N (donc $|X|$ tient sur $N-1$ bits) :

Représentation des entiers signés en complément à 2 sur N bits (CA2_N)

Méthode 1

- On veut représenter un nombre $-2^{N-1} \leq X < 2^{N-1}$ en CA2_N (donc $|X|$ tient sur $N-1$ bits) :
  1. Si X est négatif, on calcule la représentation binaire de $2^N + X$. Le premier bit sera automatiquement 1.

Représentation des entiers signés en complément à 2 sur N bits (CA2_N)

Méthode 1

- On veut représenter un nombre $-2^{N-1} \leq X < 2^{N-1}$ en CA2_N (donc $|X|$ tient sur $N-1$ bits) :
  1. Si X est négatif, on calcule la représentation binaire de $2^N + X$. Le premier bit sera automatiquement 1.
  2. Si X est positif, le premier bit sera à 0 et les $N-1$ autres bits seront la représentation de X en base 2 sur $(N-1)$ bits.

Représentation des entiers signés en complément à 2 sur N bits (CA2_N)

Méthode 1

- On veut représenter un nombre $-2^{N-1} \leq X < 2^{N-1}$ en CA2_N (donc $|X|$ tient sur $N-1$ bits) :
  1. Si X est négatif, on calcule la représentation binaire de $2^N + X$. Le premier bit sera automatiquement 1.
  2. Si X est positif, le premier bit sera à 0 et les $N-1$ autres bits seront la représentation de X en base 2 sur $(N-1)$ bits.
- Exemple -67 en complément à 2 sur 8 bits. :

Représentation des entiers signés en complément à 2 sur N bits (CA2_N)

Méthode 1

- On veut représenter un nombre $-2^{N-1} \leq X < 2^{N-1}$ en CA2_N (donc $|X|$ tient sur $N-1$ bits) :
  1. Si X est négatif, on calcule la représentation binaire de $2^N + X$. Le premier bit sera automatiquement 1.
  2. Si X est positif, le premier bit sera à 0 et les $N-1$ autres bits seront la représentation de X en base 2 sur $(N-1)$ bits.
- Exemple -67 en complément à 2 sur 8 bits. :
  1. $2^8 - 67 = 189$

Représentation des entiers signés en complément à 2 sur N bits (CA2_N)

Méthode 1

- On veut représenter un nombre $-2^{N-1} \leq X < 2^{N-1}$ en CA2_N (donc $|X|$ tient sur $N-1$ bits) :
  1. Si X est négatif, on calcule la représentation binaire de $2^N + X$. Le premier bit sera automatiquement 1.
  2. Si X est positif, le premier bit sera à 0 et les $N-1$ autres bits seront la représentation de X en base 2 sur $(N-1)$ bits.
- Exemple -67 en complément à 2 sur 8 bits. :
  1. $2^8 - 67 = 189$
  2. $189 \rightarrow \overline{10111101}^8$.

CA2_n compatibilité avec l'addition

- On reprend l'exemple de $3 + (-4)$.

CA2_n compatibilité avec l'addition

- On reprend l'exemple de $3 + (-4)$.
- En CA2_8, 3 s'écrit comme $\overline{00000011}^8$

CA2_n compatibilité avec l'addition

- On reprend l'exemple de $3 + (-4)$.
- En CA2_8, 3 s'écrit comme $\overline{00000011}^8$
- En CA2_8, pour -4 :

CA2_n compatibilité avec l'addition

- On reprend l'exemple de $3 + (-4)$.
- En CA2_8, 3 s'écrit comme $\overline{00000011}^8$
- En CA2_8, pour -4 :
  - $-4 + 256 = 252 \mapsto \overline{11111100}^8$

CA2_n compatibilité avec l'addition

- On reprend l'exemple de $3 + (-4)$.
- En CA2_8, 3 s'écrit comme $\overline{00000011}^8$
- En CA2_8, pour -4 :
  - $-4 + 256 = 252 \mapsto \overline{11111100}^8$
- Addition en binaire :

```
1    00000011
2   +11111100
3   ----------
4    11111111
```

CA2_n compatibilité avec l'addition

- On reprend l'exemple de $3 + (-4)$.
- En CA2_8, 3 s'écrit comme $\overline{00000011}^8$
- En CA2_8, pour -4 :
  - $-4 + 256 = 252 \mapsto \overline{11111100}^8$
- Addition en binaire :

```
1    00000011
2   +11111100
3   ----------
4    11111111
```

- Que des 1 : il s'agit de -1 en CA2_8. Plus besoin de modifier l'algorithme d'addition !

Représentation des entiers signés en complément à 2 sur n bits

Méthode 2

- Complément à 2 (sur n bits). Ici $n = 8$.

Représentation des entiers signés en complément à 2 sur n bits

Méthode 2

- Complément à 2 (sur n bits). Ici $n = 8$.
  1. Si le nombre est positif, donner son expression en binaire sur n bits.

Représentation des entiers signés en complément à 2 sur n bits

Méthode 2

- Complément à 2 (sur n bits). Ici $n = 8$.
  1. Si le nombre est positif, donner son expression en binaire sur n bits.
  2. Si c'est -2^{N-1} son complément à 2 est 1 suivi de $N-1$ zéros.

Représentation des entiers signés en complément à 2 sur n bits

Méthode 2

- Complément à 2 (sur n bits). Ici $n = 8$.
  1. Si le nombre est positif, donner son expression en binaire sur n bits.
  2. Si c'est -2^{N-1} son complément à 2 est 1 suivi de $N-1$ zéros.
  3. Si le nombre est négatif mais $> -2^{N-1}$. Inverser tous les bits du binaire de sa valeur absolue (ie. transformer 1 en 0 et lycée de Versailles).

Représentation des entiers signés en complément à 2 sur n bits

Méthode 2

- Complément à 2 (sur n bits). Ici $n = 8$.
  1. Si le nombre est positif, donner son expression en binaire sur n bits.
  2. Si c'est -2^{N-1} son complément à 2 est 1 suivi de $N-1$ zéros.
  3. Si le nombre est négatif mais $> -2^{N-1}$. Inverser tous les bits du binaire de sa valeur absolue (ie. transformer 1 en 0 et lycée de Versailles).
  4. Ajouter 1 au bit de poids faible (attention aux retenues) !

Représentation des entiers signés en complément à 2 sur n bits

Méthode 2

- Complément à 2 (sur n bits). Ici $n = 8$.
  1. Si le nombre est positif, donner son expression en binaire sur n bits.
  2. Si c'est -2^{N-1} son complément à 2 est 1 suivi de $N-1$ zéros.
  3. Si le nombre est négatif mais $> -2^{N-1}$. Inverser tous les bits du binaire de sa valeur absolue (ie. transformer 1 en 0 et lycée de Versailles).
  4. Ajouter 1 au bit de poids faible (attention aux retenues) !
- Exemple -67 en complément à 2 sur 8 bits. :

Représentation des entiers signés en complément à 2 sur n bits

Méthode 2

- Complément à 2 (sur n bits). Ici $n = 8$.
  1. Si le nombre est positif, donner son expression en binaire sur n bits.
  2. Si c'est -2^{N-1} son complément à 2 est 1 suivi de $N-1$ zéros.
  3. Si le nombre est négatif mais $> -2^{N-1}$. Inverser tous les bits du binaire de sa valeur absolue (ie. transformer 1 en 0 et lycée de Versailles).
  4. Ajouter 1 au bit de poids faible (attention aux retenues) !
- Exemple -67 en complément à 2 sur 8 bits. :
  1. $67 : 01000011_2$

Représentation des entiers signés en complément à 2 sur n bits

Méthode 2

- Complément à 2 (sur n bits). Ici $n = 8$.
  1. Si le nombre est positif, donner son expression en binaire sur n bits.
  2. Si c'est -2^{N-1} son complément à 2 est 1 suivi de $N-1$ zéros.
  3. Si le nombre est négatif mais $> -2^{N-1}$. Inverser tous les bits du binaire de sa valeur absolue (ie. transformer 1 en 0 et lycée de Versailles).
  4. Ajouter 1 au bit de poids faible (attention aux retenues) !
- Exemple -67 en complément à 2 sur 8 bits. :
  1. 67 : 01000011_2
  2. inversion : 10111100_2

Représentation des entiers signés en complément à 2 sur n bits

Méthode 2

- Complément à 2 (sur n bits). Ici $n = 8$.
  1. Si le nombre est positif, donner son expression en binaire sur n bits.
  2. Si c'est -2^{N-1} son complément à 2 est 1 suivi de $N-1$ zéros.
  3. Si le nombre est négatif mais $> -2^{N-1}$. Inverser tous les bits du binaire de sa valeur absolue (ie. transformer 1 en 0 et lycée de Versailles).
  4. Ajouter 1 au bit de poids faible (attention aux retenues) !
- Exemple -67 en complément à 2 sur 8 bits. :
  1. $67 : 01000011_2$
  2. inversion : 10111100_2
  3. ajout de 1 au bit de poids faible : 10111101_2. Finalement $\overline{1011\,1101}^8$.

Calcul de tête

Si l'on doit transformer un nombre négatif non nul en son complément à deux "de tête", un bon moyen est de garder tous les chiffres depuis la droite jusqu'au premier 1 (compris) puis d'inverser tous les suivants.

- Prenons par exemple le nombre -20

Calcul de tête

Si l'on doit transformer un nombre négatif non nul en son complément à deux "de tête", un bon moyen est de garder tous les chiffres depuis la droite jusqu'au premier 1 (compris) puis d'inverser tous les suivants.

- Prenons par exemple le nombre -20
- Codage binaire de sa valeur absolue 20 : 00010100

Calcul de tête

Si l'on doit transformer un nombre négatif non nul en son complément à deux "de tête", un bon moyen est de garder tous les chiffres depuis la droite jusqu'au premier 1 (compris) puis d'inverser tous les suivants.

- Prenons par exemple le nombre -20
- Codage binaire de sa valeur absolue 20 : 00010100
- On garde la partie à droite telle quelle : 00010**100**

Calcul de tête

Si l'on doit transformer un nombre négatif non nul en son complément à deux "de tête", un bon moyen est de garder tous les chiffres depuis la droite jusqu'au premier 1 (compris) puis d'inverser tous les suivants.

- Prenons par exemple le nombre -20
- Codage binaire de sa valeur absolue 20 : 00010100
- On garde la partie à droite telle quelle : 00010**100**
- On inverse la partie de gauche après le premier un : **11101**100

Calcul de tête

Si l'on doit transformer un nombre négatif non nul en son complément à deux "de tête", un bon moyen est de garder tous les chiffres depuis la droite jusqu'au premier 1 (compris) puis d'inverser tous les suivants.

- Prenons par exemple le nombre -20
- Codage binaire de sa valeur absolue 20 : 00010100
- On garde la partie à droite telle quelle : 00010**100**
- On inverse la partie de gauche après le premier un : **11101**100
- Et voici -20 : $\overline{11101100}^{8}$.

Soustraction binaire grâce au complément à 2

- On veut effectuer $15 - 14$

Soustraction binaire grâce au complément à 2

- On veut effectuer $15 - 14$
- En binaire $15 \mapsto 1111_2$ et $14 \mapsto 1110_2$

Soustraction binaire grâce au complément à 2

- On veut effectuer $15 - 14$
- En binaire $15 \mapsto 1111_2$ et $14 \mapsto 1110_2$
- 15 et -14 peuvent s'écrire en CA2_5. $15 \mapsto \overline{01111}^5$ et $-14 \mapsto \overline{10010}^5$

Soustraction binaire grâce au complément à 2

- On veut effectuer $15 - 14$
- En binaire $15 \mapsto 1111_2$ et $14 \mapsto 1110_2$
- 15 et -14 peuvent s'écrire en CA2_5. $15 \mapsto \overline{01111}^5$ et $-14 \mapsto \overline{10010}^5$
- On effectue ensuite une simple addition en CA2_5.

Soustraction binaire grâce au complément à 2

- On veut effectuer $15 - 14$
- En binaire $15 \mapsto 1111_2$ et $14 \mapsto 1110_2$
- 15 et -14 peuvent s'écrire en CA2_5. $15 \mapsto \overline{01111}^5$ et $-14 \mapsto \overline{10010}^5$
- On effectue ensuite une simple addition en CA2_5.

```
1       ***
2       01111
3       10010
4   +----------
5   (1)00001
```

Le bit le plus à gauche (1) est un *artefact* (Altération du résultat d'un examen due au procédé technique utilisé) qui n'est pas pris en compte. On sait que le résultat de $15 - 14$ tient en CA2_5 puisque ce sont deux nombres *de signes opposés* qui tiennent en CA2_5.

Soustraction binaire grâce au complément à 2

- On veut effectuer $15 - 14$
- En binaire $15 \mapsto 1111_2$ et $14 \mapsto 1110_2$
- 15 et -14 peuvent s'écrire en CA2_5. $15 \mapsto \overline{01111}^5$ et $-14 \mapsto \overline{10010}^5$
- On effectue ensuite une simple addition en CA2_5.
  -
```
1      ***
2      01111
3      10010
4   +----------
5   (1)00001
```
  . On sait que le résultat de $15 - 14$ tient en CA2_5 puisque ce sont deux nombres *de signes opposés* qui tiennent en CA2_5.
- On trouve donc que $15 - 14$ vaut $\overline{00001}^5$, ce qui correspond à 1 en CA2_5.

Du complément à 2 à la notation décimale

- Soit X de complément à 2 sur 8 bits $\overline{1010\,1101}^{8}$.

Du complément à 2 à la notation décimale

- Soit X de complément à 2 sur 8 bits $\overline{1010\,1101}^{8}$.
- Nombre négatif non nul (1 en bit de poids fort).

Du complément à 2 à la notation décimale

- Soit X de complément à 2 sur 8 bits $\overline{1010\,1101}^{8}$.
- Nombre négatif non nul (1 en bit de poids fort).
- On peut coder X en complément à 2 sur 8 bits, donc $-2^{8-1} = -128 \leq X \leq -1$.

Du complément à 2 à la notation décimale

- Soit X de complément à 2 sur 8 bits $\overline{1010\,1101}^{8}$.
- Nombre négatif non nul (1 en bit de poids fort).
- On peut coder X en complément à 2 sur 8 bits, donc $-2^{8-1} = -128 \leq X \leq -1$.
- On calcule le codage décimal de 1010 1101, puis on soustrait 256 (*i.e.* -2^8).

Du complément à 2 à la notation décimale

- Soit X de complément à 2 sur 8 bits $\overline{1010\,1101}^{8}$.
- Nombre négatif non nul (1 en bit de poids fort).
- On peut coder X en complément à 2 sur 8 bits, donc $-2^{8-1} = -128 \leq X \leq -1$.
- On calcule le codage décimal de 1010 1101, puis on soustrait 256 (*i.e.* -2^8).
-

$$-(1 \times 2^8) + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + \ldots$$
$$\cdots + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = -83 = X$$

Parité et signe en complément à 2

- Pour reconnaître un nombre positif en complément à 2 sur n bits, il suffit que son bit de poids fort soit à 0,

Parité et signe en complément à 2

- Pour reconnaître un nombre positif en complément à 2 sur n bits, il suffit que son bit de poids fort soit à 0,
- Négatif : son bit de poids fort à 1.

Parité et signe en complément à 2

- Pour reconnaître un nombre positif en complément à 2 sur n bits, il suffit que son bit de poids fort soit à 0,
- Négatif : son bit de poids fort à 1.
- -1 est toujours l'entier qui, en complément à deux sur n bits, a une représentation ne comportant que des 1.

Représentation des entiers signés en complément à 2

FIGURE – Quelques entiers signés en complément à 2

bit de signe								
0	1	1	1	1	1	1	1	= 127
0	0	0	0	0	0	1	0	= 2
0	0	0	0	0	0	0	1	= 1
0	0	0	0	0	0	0	0	= 0
1	1	1	1	1	1	1	1	= −1
1	1	1	1	1	1	1	0	= −2
1	0	0	0	0	0	0	1	= −127
1	0	0	0	0	0	0	0	= −128

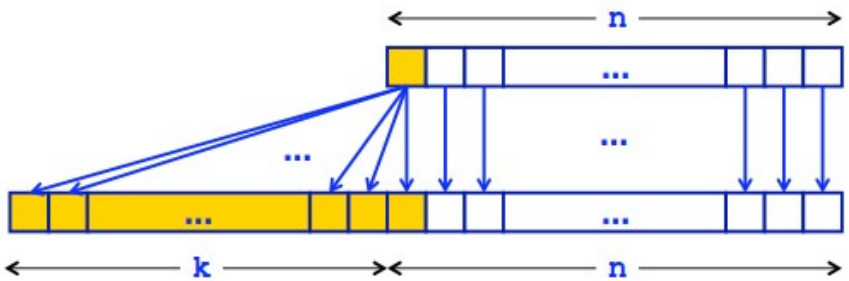
Entiers de 8 bits en complément à deux

En complément à 2 sur N bits on représente tous les entiers de -2^{N-1} à $2^{N-1}-1$. Sur 32 bits : de $-2^{31} = -2147483648$ à $2^{31}-1 = 2147483647$ (plus ou moins 2 milliards).

Extension de format

FIGURE – Duplication du bit de poids fort

- Extension de signe:
  Si l'on veut passer un entier signé **x** d'un format **n** bits vers un format **n+k** bits, en gardant la même valeur, il suffit de faire une extension de signe: le bit de signe est répété sur les nouveaux **k** bits de poids fort



Exemple : soit $\overline{10001000}^{8}$ un entier signé représenté en complément à 2 sur 8 bits. En complément à 2 sur 12 bits, on obtient : $\overline{111110001000}^{12}$

Justification

- soit T la représentation en CA2N de X (pour $N > 2$).

Justification

- soit T la représentation en CA2N de X (pour $N > 2$).
- Si $X \geq 0$, son codage en CA2N est son codage en binaire (il tient forcément sur moins de $N-1$ bits) auquel on ajoute des zéros à gauche pour remplir N bits. Pour remplir $N+1$ bits, on ajoute juste un zéro de plus à gauche.
  Et donc, une extension de format de CA2N à CA2(N+1) consiste juste à dupliquer le bit de poids fort.

Justification

- soit T la représentation en CA2N de X (pour $N > 2$).
- Si $X \geq 0$,
  Et donc, une extension de format de CA2N à CA2(N+1) consiste juste à dupliquer le bit de poids fort.
- Si $X < 0$, $2^{N-1} \leq X \leq -1$ donc $-2^{N-1} \leq X + 2^N \leq 2^N - 1$.
  Ainsi $X + 2^N$ est de la forme $2^{N-1} + \sum_{i=0}^{N-2} x_{N-2-i} 2^{N-2-i}$
  ($\overline{1x_{N-2} \dots x_0}^N$ est le codage de X).
  Codons X en CA2(N+1) :

Justification

- soit T la représentation en CA2N de X (pour $N > 2$).
- Si $X \geq 0$,
  Et donc, une extension de format de CA2N à CA2(N+1) consiste juste à dupliquer le bit de poids fort.
- Si $X < 0$, $2^{N-1} \leq X \leq -1$ donc $-2^{N-1} \leq X + 2^N \leq 2^N - 1$.
  Ainsi $X + 2^N$ est de la forme $2^{N-1} + \sum_{i=0}^{N-2} x_{N-2-i} 2^{N-2-i}$
  ($\overline{1x_{N-2} \dots x_0}^N$ est le codage de X).
  Codons X en CA2(N+1) :
  - $X+2^{N+1} = 2^N+(X+2^N) = 2^N+2^{N-1}+\sum_{i=0}^{N-2} x_{N-2-i}2^{N-2-i}$.

Justification

- soit T la représentation en CA2N de X (pour $N > 2$).
- Si $X \geq 0$,
  Et donc, une extension de format de CA2N à CA2(N+1) consiste juste à dupliquer le bit de poids fort.
- Si $X < 0$, $2^{N-1} \leq X \leq -1$ donc $-2^{N-1} \leq X + 2^N \leq 2^N - 1$.
  Ainsi $X + 2^N$ est de la forme $2^{N-1} + \sum_{i=0}^{N-2} x_{N-2-i} 2^{N-2-i}$
  ($\overline{1x_{N-2} \dots x_0}^N$ est le codage de X).
  Codons X en CA2(N+1) :
  - $X+2^{N+1} = 2^N+(X+2^N) = 2^N+2^{N-1}+\sum_{i=0}^{N-2} x_{N-2-i}2^{N-2-i}$.
  - Ainsi le codage en CA2(N+1) de X est $\overline{11x_{N-2} \dots x_0}^{N+1}$.

Overflow

- Il y a *overflow* (dépassement de capacité) lorsque le nombre à représenter ne tient pas dans le format choisi.

Overflow

- Il y a *overflow* (dépassement de capacité) lorsque le nombre à représenter ne tient pas dans le format choisi.
- Pour reconnaître un overflow dans une addition en complément à 2 :

Overflow

- Il y a *overflow* (dépassement de capacité) lorsque le nombre à représenter ne tient pas dans le format choisi.
- Pour reconnaître un overflow dans une addition en complément à 2 :
  1. si les deux opérandes sont du même signe : dépassement si le résultat est de signe opposé,

Overflow

- Il y a *overflow* (dépassement de capacité) lorsque le nombre à représenter ne tient pas dans le format choisi.
- Pour reconnaître un overflow dans une addition en complément à 2 :
  1. si les deux opérandes sont du même signe : dépassement si le résultat est de signe opposé,
  2. Si les deux opérandes sont de signes opposés, il n'y a jamais dépassement.

Exemple

- On additionne $\overline{01111000}^{8}$ (120) et $\overline{01110011}^{8}$ (115) en CA2_8.

Exemple

- On additionne $\overline{01111000}^8$ (120) et $\overline{01110011}^8$ (115) en CA2_8.
  - ```
    1    ****
    2    01110011
    3   +01111000
    4   ----------
    5    11101011 (changement de signe donc overflow)
    ```
    ```

Exemple

- On additionne $\overline{01111000}^8$ (120) et $\overline{01110011}^8$ (115) en CA2_8.
  - ```
    1    ****
    2    01110011
    3   +01111000
    4   ----------
    5    11101011 (changement de signe donc overflow)
    ```
- On constate un overflow.
    ```

Exemple

- On additionne $\overline{01111000}^8$ (120) et $\overline{01110011}^8$ (115) en CA2_8.
  - ```
    1    ****
    2    01110011
    3   +01111000
    4   ----------
    5    11101011 (changement de signe donc overflow)
    ```
- On constate un overflow.
- On passe au CA2_9. 120 : $\overline{001111000}^9$ et 115 : $\overline{001110011}^9$
    ```

Exemple

- On additionne $\overline{01111000}^8$ (120) et $\overline{01110011}^8$ (115) en CA2_8.
  - ```
    1    ****
    2    01110011
    3   +01111000
    4   ----------
    5    11101011 (changement de signe donc overflow)
    ```
- On constate un overflow.
- On passe au CA2_9. 120 : $\overline{001111000}^9$ et 115 : $\overline{001110011}^9$

```
1     ****
2    001110011
3   +001111000
4   -----------
5    011101011
```
```

Exemple

- On additionne $\overline{01111000}^{8}$ (120) et $\overline{01110011}^{8}$ (115) en CA2_8.
  - ```
    1    ****
    2    01110011
    3   +01111000
    4   ----------
    5    11101011 (changement de signe donc overflow)
    ```
- On constate un overflow.
- On passe au CA2_9. 120 : $\overline{001111000}^{9}$ et 115 : $\overline{001110011}^{9}$

```
1     ****
2    001110011
3   +001111000
4   -----------
5    011101011
```

- Le résultat, $\overline{011101011}^{9}$ représente 235 en CA2_9.
```

Valeurs maximales

- Le fichier `limits.h` (voir ici) contient toutes les réponses à nos questions.

Valeurs maximales

- Le fichier `limits.h` (voir ici) contient toutes les réponses à nos questions.
- Compilons :

```c
# include <stdio.h>
# include <limits.h>
int main (void) {
   printf("%d\n",INT_MAX);
   int x = INT_MAX + 1;// produit un Warning !!
   printf("%d\n",x);
   return 0;
}
```

Sur ma machine, on obtient

```
2147483647
-2147483648
```

Les entiers sont donc implémentés en CA2_32.

Contraindre la taille des entiers

- Les entiers `int` sont souvent sur 4 octets mais pas toujours : cela dépend des implémentations.

Contraindre la taille des entiers

- Les entiers `int` sont souvent sur 4 octets mais pas toujours : cela dépend des implémentations.
- Lorsque la taille des entiers est élément important pour le programmeur, le fichier d'en-tête `<stdint.h>` est une aide précieuse. Il décrit des types de tailles précises `intN_t` (entiers signés de N bits) et `uintN_t` (non signés sur N bits).

Contraindre la taille des entiers

- Les entiers `int` sont souvent sur 4 octets mais pas toujours : cela dépend des implémentations.
- Lorsque la taille des entiers est élément important pour le programmeur, le fichier d'en-tête `<stdint.h>` est une aide précieuse. Il décrit des types de tailles précises `intN_t` (entiers signés de N bits) et `uintN_t` (non signés sur N bits).
- Par exemple : `uint16_t` représente les entiers non signés sur 16 bits (2 octets).

Contraindre la taille des entiers

```c
#include <stdint.h>
int main(){
  int8_t x = 127;//max valeur signée sur 8 bits
  printf("%4d\n", x);
  x = x + 1;
  printf("%4d", x);  // dépassement
```

On obtient :

```
 127
-128
```

Cette section hors programme est laissée au lecteur curieux.

Découpage des entiers en Python

D'après Nicolas Pécheux.

- Les entiers en `Python` ont une précision non limitée. Ils ne fonctionnent donc pas en complément à 2.

Découpage des entiers en Python

D'après Nicolas Pécheux.

- Les entiers en `Python` ont une précision non limitée. Ils ne fonctionnent donc pas en complément à 2.
- Sur une machine 32 bits, la représentation binaire d'un entier `Python` x est découpée en paquets de 15 bits, stockés dans un tableau dont les éléments sont des entiers de 16 bits (pour que la taille soit un multiple de 8).

Découpage des entiers en Python

D'après Nicolas Pécheux.

- Les entiers en `Python` ont une précision non limitée. Ils ne fonctionnent donc pas en complément à 2.
- Sur une machine 32 bits, la représentation binaire d'un entier `Python` x est découpée en paquets de 15 bits, stockés dans un tableau dont les éléments sont des entiers de 16 bits (pour que la taille soit un multiple de 8).
- Les cases du tableau sont donc les coefficients de x dans sa représentation en base 2^{15}.

Découpage des entiers en Python

D'après Nicolas Pécheux.

- Les entiers en `Python` ont une précision non limitée. Ils ne fonctionnent donc pas en complément à 2.
- Sur une machine 32 bits, la représentation binaire d'un entier `Python` x est découpée en paquets de 15 bits, stockés dans un tableau dont les éléments sont des entiers de 16 bits (pour que la taille soit un multiple de 8).
- Les cases du tableau sont donc les coefficients de x dans sa représentation en base 2^{15}.
- A ce tableau est associée un (petit) entier indiquant le nombre de chiffres de x dans la base 2^{15}.

Exemple

D'après Nicolas Pécheux.

- Soit le nombre $x = 1234567891011$. Il s'écrit en binaire

$$\underbrace{000010001111101}_{1149_{10}} \mid \underbrace{110001111110110}_{25590_{10}} \mid \underbrace{000100001000011}_{2115_{10}}$$

Exemple

D'après Nicolas Pécheux.

- Soit le nombre $x = 1234567891011$. Il s'écrit en binaire

$$\underbrace{000010001111101}_{1149_{10}} \mid \underbrace{110001111110110}_{25590_{10}} \mid \underbrace{000100001000011}_{2115_{10}}$$

- Dans la base 2^{15}, il faut 32768 symboles pour les digits. On peut garder l'écriture en base 10 comme symbole. Ainsi x s'écrit

$$1149_{10}25590_{10}2115_{10}{}_{2^{15}} \text{ en base } 2^{15}$$

Exemple

D'après Nicolas Pécheux.

- Soit le nombre $x = 1234567891011$. Il s'écrit en binaire

$$\underbrace{000010001111101}_{1149_{10}} \mid \underbrace{110001111110110}_{25590_{10}} \mid \underbrace{000100001000011}_{2115_{10}}$$

- Dans la base 2^{15}, il faut 32768 symboles pour les digits. On peut garder l'écriture en base 10 comme symbole. Ainsi x s'écrit

$$1149_{10}25590_{10}2115_{10}{}_{2^{15}} \text{ en base } 2^{15}$$

- Alors x est représenté en PYTHON par le tableau `[1149, 25590, 2115]` et l'entier 3. Et $-x$ est représenté par `[1149, 25590, 2115]` et -3.

Opérations arithmétiques

- La base choisie (2^{15} pour une machine 32 bits, 2^{30} pour une 64 bits) ne doit pas être trop grande pour que le processeur puisse faire les opérations arithmétiques entre les digits.

Opérations arithmétiques

- La base choisie (2^{15} pour une machine 32 bits, 2^{30} pour une 64 bits) ne doit pas être trop grande pour que le processeur puisse faire les opérations arithmétiques entre les digits.
- En cas d'addition par exemple, les coefficients de mêmes exposants sont additionnés par le processeur. Un petit logiciel est ensuite chargé d'harmoniser les résultats (en cas de propagation de retenue par exemple).

Opérations arithmétiques

- La base choisie (2^{15} pour une machine 32 bits, 2^{30} pour une 64 bits) ne doit pas être trop grande pour que le processeur puisse faire les opérations arithmétiques entre les digits.
- En cas d'addition par exemple, les coefficients de mêmes exposants sont additionnés par le processeur. Un petit logiciel est ensuite chargé d'harmoniser les résultats (en cas de propagation de retenue par exemple).
- Ce traitement logiciel ralenti les opérations arithmétiques sur les entiers PYTHON.

- Pour connaître la base (en exposant de 2) et la taille d'occupation (en octets) de chaque digit dans cette base, il suffit d'entrer

```
1 import sys
2 sys.int_info
```

- Pour connaître la base (en exposant de 2) et la taille d'occupation (en octets) de chaque digit dans cette base, il suffit d'entrer

```
1 import sys
2 sys.int_info
```

- On obtient

```
sys.int_info(bits_per_digit=30,
       sizeof_digit=4)
```