Complément à un : addition, signes opposés

- Soient p et q deux entiers de signes opposés.
- Leur somme est toujours représentable pour la taille de mot mémoire fixée car

$$\min(p,q) < p+q < \max(p,q)$$

Exemple :
On a $(6)_{10} + (-6)_{10}$ qui s'écrit en $CA1_4$: $0110 + 1001 = 1111$

1. Il faut deux tests pour zéro.
2. Écrit en décimal, on a calculé $6 + \left((2^4 - 1) - 6\right)$

Complément à un : addition, signes opposés

- On travaille avec $k = 8$, *i.e.* des octets ;
- On veut calculer $28 - 63 = 28 + (-63)$;
- D'après ce qui précède :

$$\begin{array}{lll} +28 \text{ s'écrit} & 0001\ 1100 & \text{en } CA1_8 \\ +63 \text{ s'écrit} & 0011\ 1111 & \text{en } CA1_8 \\ -63 \text{ s'écrit} & 1100\ 0000 & \text{en } CA1_8 \end{array}$$

- On calcule $28 + (-63)$ en binaire habituel :

$$\begin{array}{cc} 0001 & 1100 \\ 1100 & 0000 \\ \hline 1101 & 1100 \end{array}$$

- Le résultat représente bien -35 en $CA1_8$.
  Note : il n'y a pas de retenue.

Complément à un : addition, signes opposés

- On travaille avec $k = 8$, *i.e.* des octets ;
- On veut déterminer $63 - 28 = 63 + (-28)$;
- On a $+63$ qui s'écrit 0011 1111 en $CA1_8$;
- Pour -28, on note que $+28$ s'écrit 0001 1100 ,
  le complément à un donne 1110 0011 ;
- On calcule $63 + (-28)$:

$$\begin{array}{rcc} & 0011 & 1111 \\ & 1110 & 0011 \\ \hline 1 & 0010 & 0010 \end{array}$$

- On a une retenue qui vaut $2^8 = 256$
  or on dépasse les k bits alloués, la retenue est négligée ;
- Écrit en décimal, on a calculé $63 + (255 - 28) = 35 + 255 = 290$
  Pour obtenir le bon résultat, il faut retrancher $255 = 2^8 - 1$;
- On ajoute donc 1 et <u>on dit</u> qu'on ajoute la retenue ;
- D'où 0010 0010 $+ 1 =$ 0010 0011 $= (35)_{10}$.

Complément à un : addition, même signe

- L'addition de deux nombres de même signe peut donner lieu à un **dépassement de capacité** !
- Cas de deux entiers de signe positif.
  On a un dépassement de capacité quand le bit de signe du résultat vaut 1.

  Exemples : sur 8 bits on code les nombres de -127 à $+127$

$$\begin{array}{rcll} +35 & : & 0010 & 0011 \\ +65 & : & 0100 & 0001 \\ \hline +100 & : & 0110 & 0100 \end{array} \qquad \begin{array}{rcll} +103 & : & 0110 & 0111 \\ +65 & : & 0100 & 0001 \\ \hline +168 & \neq & 1010 & 1000 \end{array}$$

À droite dépassement de capacité,
on obtient un nombre qui représente -87 en complément à un.

Complément à un : addition, même signe

- Cas de deux entiers de signe négatif.
- Il y a toujours une retenue puisque les bits de signe valent 1.
- On doit ajouter la retenue :
  ⋆ un 0 pour le bit de signe indique un dépassement de capacité ;
  ⋆ un 1 pour le bit de signe indique un résultat négatif, écrit en $CA1$.

Exemples : sur 8 bits, nombres de -127 à $+127$

$$
\begin{array}{rccc}
-35 & : & & 1101\ \ 1100 \\
-65 & : & & 1011\ \ 1110 \\
\hline
 & & 1 & 1001\ \ 1010 \\
-100 & : & & 1001\ \ 1011
\end{array}
\qquad
\begin{array}{rccc}
-103 & : & & 1001\ \ 1000 \\
-65 & : & & 1011\ \ 1110 \\
\hline
 & & 1 & 0101\ \ 0111 \\
-168 & \neq & & 0101\ \ 1000
\end{array}
$$

Dépassement de capacité.

Résumé pour l'addition en «complément à un»

- 2 nombres de signes opposés
  - ⋆ le résultat est représentable avec le nombre de bits fixés, *pas de dépassement de capacité* ;
  - ⋆ s'il n'y a pas de retenue, le résultat est un nombre négatif dont la *valeur absolue est obtenue en inversant les bits* ;
  - ⋆ s'il y a une retenue, le résultat est un nombre positif, *on ajoute la retenue pour avoir sa valeur* ;
- 2 nombres de signe positif
  - ⋆ *dépassement de capacité* quand le bit de signe du résultat est 1 ;
- 2 nombres de signe négatif
  - ⋆ il y a une retenue puisque les deux bits de signe sont 1 ;
  - ⋆ on ajoute la retenue et
    - un 0 pour le bit de signe signifie un *dépassement de capacité* ;
    - un 1 pour le bit de signe, on a le bon résultat .

Complément à deux

Définition

- La taille des mots k est fixée.
- Le bit de signe est placé en tête (bit de poids fort).
- Les entiers positifs $n \in \mathbb{N}$ sont codés en binaire naturel signé.
- Pour un entier négatif $n \in \mathbb{Z}_-$:
  on code la valeur absolue $|n|$ en binaire naturel,
  ensuite on *inverse les bits un à un* ($CA1_k$) et l'on *ajoute 1*.

Exemple : sur $k = 4$ bits, représenter $(-6)_{10}$ en complément à deux.

la valeur absolue est 6	:	0110
inversion bit à bit	:	1001
on ajoute 1	:	1010

Représentation de $(-6)_{10}$ en complément à deux sur 4 bits 1010

Complément à deux

- On travaille sur k bits et on représente $\{-2^{k-1}, \ldots, 2^{k-1} - 1\}$.
- Le complément à deux d'un entier n est obtenu par
  le calcul de la quantité $2^k - n$.
  En effet, "inversion" et " + 1" revient à faire $((2^k - 1) - n) + 1$
  Attention à l'ordre : " + 1" et ensuite "inversion" revient à faire
  $(2^k - 1) - (n + 1) = 2^k - (n + 2)$
- Le complément à deux du complément à deux d'un entier n
  est l'entier lui même.
  En effet, $2^k - (2^k - n) = n$
- Le complément à deux de zéro est zéro.
  En effet, $2^k - 0 = 2^k$*, on néglige la retenue qui dépasse la taille fixée.*
- Le nom complet de cette opération est «complément à 2^k»
  qui est en général tronqué en «complément à 2».

Complément à deux

Si l'entier n est codé sur k bits en complément à deux :

n :	s_{k-1}	s_{k-2}	$\cdots$	s_3	s_2	s_1	s_0

alors $$(n)_{10} = -s_{k-1}2^{k-1} + \sum_{i=0}^{k-2} s_i\, 2^i \,.$$

En complément à deux, le bit de signe s_{k-1} a comme poids -2^{k-1} .

Pour $k = 8$ bits, on a les entiers signés :

-128	=	1	0	0	0	0	0	0	0
	$\vdots$								
-1	=	1	1	1	1	1	1	1	1
0	=	0	0	0	0	0	0	0	0
1	=	0	0	0	0	0	0	0	1
	$\vdots$								
127	=	0	1	1	1	1	1	1	1

Complément à deux

Pour un codage sur 4 bits on a :

-8	=	1	0	0	0	0	=	0	0	0	0
-7	=	1	0	0	1	1	=	0	0	0	1
-6	=	1	0	1	0	2	=	0	0	1	0
-5	=	1	0	1	1	3	=	0	0	1	1
-4	=	1	1	0	0	4	=	0	1	0	0
-3	=	1	1	0	1	5	=	0	1	0	1
-2	=	1	1	1	0	6	=	0	1	1	0
-1	=	1	1	1	1	7	=	0	1	1	1

On a 2^4 valeurs distinctes de $-8 = -2^{4-1}$ à $+7 = 2^{4-1} - 1$.
Vérifier que :

- Si $n \in \mathbb{N}^*$, l'entier négatif $(-n)_{10}$ est codé par $(2^4 - n)_2$;
- si le code *CA*2, $m = (s_3 s_2 s_1 s_0)$ commence par $s_3 = 1$, alors m représente la valeur $(m)_{10} - 2^4$.

Complément à deux

Avantages et inconvénients :

- (+) Codage/décodage facile.
- (+) Représentation unique de zéro.
- (+) Opérations arithmétiques faciles (*cf.* addition).
- (-) Taille mémoire fixée.

Faire attention au langage et distinguer :

1. la **représentation/notation** d'un nombre en complément à deux ;
2. l'**opération mathématique** de prendre le complément à deux d'un nombre.

Exemple : Sur 4 bits, 0111 représente 7 en complément à deux.
Le complément à deux de 7 est $9 = (1001)_2$
et la représentation en complément à deux de -7 est 1001.

Complément à deux : addition, signes opposés

- Le résultat est toujours représentable.
- Exemple sur un octet, $k = 8$:

$+63$	:		0011	1111
-63	:		1100	0001
report		1	1 1 1 1	1 1 1
		1	0000	0000
0	:		0000	0000

- On ne tient pas compte de la retenue.
  On effectue $+63 \; + \; (256 - 63) \; - \; 2^8$.

Complément à deux : addition, signes opposés

- Exemples sur un octet, $k = 8$:

-63	:	1100	0001	$+63$	:	0011	1111
$+28$	:	0001	1100	-28	:	1110	0100
report	0	0		report	1	1 1 1 1	1
					1	0010	0011
-35	:	1101	1101	$+35$	:	0010	0011

- S'il n' y a pas de retenue, on lit le résultat directement.

  S'il y a une retenue, on la néglige, *i.e.* on retranche 2^8.

Complément deux : addition, même signe

- L'addition de deux nombres de même signe peut donner lieu à un **dépassement de capacité** !
- Cas de deux entiers de signe positif.
  On a un dépassement de capacité quand la retenue est distincte du dernier bit de report (*i.e.* celui sur le bit de signe).

  Exemples : sur 8 bits, nombres de -128 à $+127$

$+35$	:	0010	0011	$+103$	:	0110	0111
$+65$	:	0100	0001	$+65$	:	0100	0001
report	0	0 0 0 0	0 1 1	report	$0\neq$	1 0 0 0	1 1 1
$+100$	:	0110	0100	$+168$	$\neq$	1010	1000

À droite dépassement de capacité, on obtient un nombre qui représente -88 en complément à deux.

Complément à deux : addition, même signe

- Cas de deux entiers de signe négatif.

  On a toujours une retenue, que l'on oublie.
  En effet, on calcule $(2^k - n_1) + (2^k - n_2) - 2^k$.

  On a un dépassement de capacité quand la retenue est distincte du dernier bit de report (*i.e.* celui sur le bit de signe).

  Exemples : sur 8 bits, nombres de -128 à $+127$

-35	:		1101	1101
-65	:		1011	1111
report		1	1 0 0 0	0 1 1
		1	1001	1100
-100	:		1001	1100

-103	:		1001	1001
-65	:		1011	1111
report		$1 \neq$	0 1 1 1	1 1 1
		1	0101	1000
-168	$\neq$		0101	1000

Dépassement de capacité.

Résumé pour l'addition en «complément à deux»

- 2 nombres de signes opposés
  - ⋆ Le résultat est représentable avec le nombre de bits fixés, *pas de dépassement de capacité* ;
  - ⋆ s'il y a une retenue, on l'oublie !
  - ⋆ On lit directement le résultat codé en *CA*2
- 2 nombres de même signe
  - ⋆ Il y a *dépassement de capacité* si la retenue est distincte du dernier bit de report (i.e. celui sur le bit de signe) ;
  - ⋆ s'il y a une retenue on l'oublie !
  - ⋆ On lit directement le résultat codé en *CA*2

Conclusion :

1. L'addition en codage complément à deux est simplement l'addition binaire. On ne garde jamais la retenue.
2. On détecte les dépassements de capacité grâce à un seul test pour tous les cas de figure.
3. Pour les autres opérations arithmétiques sur les sentiers signés, le codage complément à deux présente des avantages similaires. Ce codage est donc souvent choisi en pratique.

Application du codage complément à deux dans le langage C

Dans le fichier `/usr/include/bits/wordsize.h`
se trouve la taille des mots mémoire : `#define __WORDSIZE 32`

Dans le fichier `/usr/include/limits.h` sont précisés :

Type	Taille	Magnitude		
signed char	8 bits	- 128	à	+ 127
unsigned char	8 bits	0	à	255
signed short int	16 bits	- 32 768	à	+ 32 767
unsigned short int	16 bits	0	à	65 535
signed (long) int	32 bits	- 2 147 483 648	à	+ 2 147 483 647
unsigned (long) int	32 bits	0	à	4 294 967 295

Note : `gcc 4.5.1` et `ISO C99 Standard: 7.10/5.2.4.2.1`

Exemple de code C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

main()
{
  unsigned char    uc1, uc2, uc3;    signed char   sc1, sc2, sc3;
  unsigned short   ui1, ui2, ui3;    signed short  si1, si2, si3;

  printf("\n Taille de  char     : %d octets \n\n",sizeof(char));

  uc1 = 200 ; uc2 = 60 ; uc3 = uc1 + uc2 ;
  printf("(unsigned char) uc1 = %d, uc2 = %d, uc1+uc2 = %d \n",uc1, uc2, uc3) ;

  sc1 =  103 ; sc2 =  65 ; sc3 = sc1+sc2 ;
  printf("(signed char) sc1 = %d, sc2 = %d, sc1+sc2 = %d \n",sc1, sc2, sc3) ;

  sc1 = -103 ; sc2 = -65 ; sc3 = sc1+sc2 ;
  printf("(signed char) sc1 = %d, sc2 = %d, sc1+sc2 =  %d \n",sc1, sc2, sc3) ;

  printf("\n Taille de  short    : %d octets\n\n",sizeof(short));

  ui1 = 6000 ; ui2 = 60000 ; ui3 = ui1+ui2 ;
  printf("(unsigned short) ui1 = %d,  ui2 = %d, ui1+ui2 = %d \n",ui1, ui2, ui3) ;

  si1 = -10000 ; si2 = -30000 ; si3 = si1+si2 ;
  printf("(signed short) si1 = %d, si2 = %d, si1+si2 = %d \n",si1, si2, si3) ;
}
```

Exemple de code C : affichage des résultats

```
Taille de  char      : 1 octet(s)

(unsigned char ) uc1 =  200, uc2 = 60, uc1+uc2 = 4

(signed char) sc1 =  103, sc2 =  65, sc1+sc2 = -88
(signed char) sc1 = -103, sc2 = -65, sc1+sc2 =  88

Taille de  short     : 2 octet(s)

(unsigned short) ui1 =   6000, ui2 =  60000,
                                      ui1+ui2 =   464
(signed short)   si1 = -10000, si2 = -30000 ,
                                      si1+si2 = 25536
```

Ces faux résultats sont dus au dépassement de capacité des opérations effectuées en code complément à deux.

Conclusion

- La représentation des nombres entiers est limitée par la *taille du mot mémoire* qui leur est affectée.
- Le code le plus souvent utilisé est le *code complément à deux*.
  Il n'a qu'une seule représentation du zéro, les opérations arithmétiques et la détection de dépassement de capacité sont faciles à effectuer.
- Dans tous les cas et en fonction des architectures d'ordinateurs il y aura toujours des opérations dont le résultat n'est pas représentable.
  Sans précautions, elles engendrent des résultats aberrants ou empêchent la poursuite des calculs.