Chapitre 4
Introduction à l'arithmétique sur ordinateur

Lorsqu'on pense aux opérations arithmétiques, on utilise les connaissances acquises à l'école primaire et à l'école secondaire qui permettent d'effectuer les quatre opérations sur des nombres entiers ou des nombres réels. Pour une addition, par exemple, on sait que l'on doit toujours procéder de droite à gauche en reportant les retenues, s'il y en a, vers la gauche. Toutes les opérations sont ainsi effectuées mécaniquement sans qu'il y ait besoin de réflexion.

Un ordinateur doit pouvoir manipuler des valeurs numériques semblables et leur appliquer les quatre opérations arithmétiques de base. Comme on le faisait déjà, on distinguera les nombres entiers des nombres réels, ce qui nous conduira à avoir deux sortes d'arithmétiques. Certains ordinateurs en possèdent même une troisième sorte basée sur une représentation décimale des nombres.

Une première différence entre l'arithmétique des ordinateurs et l'arithmétique apprise à l'école est que l'ordinateur manipule des valeurs binaires; ceci ne pose pas de difficultés supplémentaires car, bien que nous soyons habitués au système décimal, nous sommes en mesure d'effectuer les opérations arithmétiques dans le système binaire sans grand effort. Les opérations que l'on effectue de façon habituelle utilisent des nombres possédant un signe: ils sont formés d'une valeur absolue précédée du signe + ou du signe -. Pour faire l'addition de deux nombres on doit considérer le signe des nombres: si les deux signes sont identiques le résultat aura le même signe et une valeur absolue qui sera la somme des valeurs absolues des deux nombres. Cependant, si les deux nombres ont des signes différents il va falloir calculer la différence des valeurs absolues; il faut alors déterminer lequel des deux nombres a la plus grande valeur absolue afin de pouvoir effectuer la soustraction et obtenir le signe du résultat. Cet algorithme d'addition, simple pour nous, est cependant trop complexe pour être réalisé de façon satisfaisante par l'ordinateur; il est en effet plus simple de traiter la soustraction comme un cas particulier de l'addition. Pour ce faire on doit utiliser une représentation des nombres un peu différente de ce à quoi nous sommes habitués.

4.1 Arithmétique en complément

Comme nous venons de le voir, l'addition de nombres avec signes est un processus compliqué, difficile à réaliser électroniquement. On utilise en fait des algorithmes d'addition bien plus simples utilisant une autre sorte d'arithmétique, dite en complément.

Pour illustrer les idées de base de cette sorte d'arithmétique prenons un compteur d'automobile et examinons les nombres qu'il montre lorsqu'on le fait marcher à l'envers:

```
.........
00005
00004
00003
00002
00001
00000
99999
99998
.........
```

Le nombre précédant zéro est donc 99999 au lieu du nombre auquel on s'attendait en arithmétique: -1. Qu'arrive-t-il si l'on ajoute 00005 et 99999? Comme le compteur n'a que 5 chiffres la réponse n'aura que 5 chiffres:

```
  00005
+ 99999
  00004
```

la dernière retenue a été perdue, faute de place, et 99999 a eu le même effet que -1. De même:

```
  00005
+ 99998
  00003
```

99998 a agi comme -2. Nous avons là un *système en complément*. La valeur 99999 est le complément de 1, la valeur 99998 est le complément de 2, etc... Dans un tel système, il n'y a pas de signe pour les nombres négatifs, ceux-ci sont caractérisés par leur appartenance à un domaine donné. Si l'on ne tient compte que de 5 chiffres le complément d'un nombre est équivalent au négatif de ce nombre.
Exemple :

```
(15-6) 00015     (244-41) 00244    (12-16)   00012
     + 99994              + 99959            + 99984
       00009                00203              99996
```

Dans le dernier exemple la réponse est donnée sous forme de complément; on doit définir une méthode permettant de distinguer les nombres négatifs des nombres positifs.

Le domaine des nombres allant de 00000 à 99999, on le divise arbitrairement en deux parties:

- de 00000 à 49999 nombres positifs
- de 50000 à 99999 nombres négatifs

Les nombres négatifs auront donc un premier chiffre supérieur ou égal à 5.
Exemples:

```
  00543
+ 99418    soit 543 - 582 = -39
  99961

  49999
+ 50001    soit 49999 - 49999 = 0
  00000
```

Si l'on ajoute deux grandes valeurs on obtient un résultat inexact:

```
  40000
+ 40000
  80000    soit -20000
```

Dans un cas comme celui-là, la réponse 80000 ne se trouve pas dans le domaine des nombres positifs 00000 - 49999. Il s'est produit un *débordement*, le résultat étant trop grand pour le domaine prévu. Si le domaine choisi est suffisamment grand le problème du débordement n'est pas trop sérieux.

L'arithmétique en complément convient bien aux ordinateurs car comme pour les compteurs, la taille des mots est fixe.

Dans un système de numération en base b le complément d'un nombre X est défini par: $b^n - X$, où n représente le nombre de chiffres et b la base. En revenant à notre exemple nous avions: b = 10 et n = 5, le complément de X était donc $10^5 - X$, soit si X vaut 34: 100000 - 34 = 99966.
Dans le système binaire: b = 2, n = 7

- de 0000000 à 0111111 nombres positifs
- de 1000000 à 1111111 nombres négatifs

On trouve le complément par 2^7 - X; si X vaut 0011010 son complément est:
10000000 - 0011010 = 1100110 .

Plutôt que de faire les soustractions on utilise généralement un truc pour trouver les compléments. On peut remarquer que dans toutes les bases b^n - 1 est une suite du plus grand chiffre, par exemple, 10^5 - 1 = 99999 et 2^7 - 1 = 1111111. Le complément en base diminuée (b^n - 1) - X est facile à trouver car il suffit de soustraire chaque chiffre du plus grand chiffre dans le système de numération en base b:

0011010 → 1100101
00402 → 99597

Pour obtenir le complément d'un nombre il suffit d'ajouter 1 à son complément en base diminuée.

0011010 → 1100101 → 1100110
00402 → 99597 → 99598

En base 2 les compléments sont appelés compléments à 2 et les compléments en base diminuée sont appelés compléments à 1. L'ordinateur PEP 8 effectue son arithmétique avec des compléments à 2. Toutes les valeurs négatives sont représentées par les compléments à 2 des valeurs positives correspondantes.

Exemples:

```
(16+8)      0010000     (16-8)  0010000     (8-16)  0001000
          + 0001000           + 1111000           + 1110000
   24       0011000      8      0001000     -8      1111000
```

Sur l'ordinateur PEP 8 la taille des mots est 16 bits; les valeurs dont le bit le plus à gauche est un zéro sont positives, les autres sont négatives. La valeur positive la plus grande est:

0111111111111111 $7FFF_{16}$ ou $2^{15} - 1$ soit 32767_{10}.

Le plus grand complément est:

1000000000000000 8000_{16} ou -2^{15} soit -32768_{10}

Pourquoi y a-t-il une valeur de plus dans les négatifs que dans les positifs? La valeur

1111111111111111 $FFFF_{16}$

représente - 1.

Le bit de gauche permet de déterminer le signe du nombre bien que ce ne soit pas réellement un signe mais un bit comme les autres lors des opérations arithmétiques.

En utilisant le système hexadécimal pour représenter les valeurs nous avons sur 16 bits:

valeur maximum	7FFF
valeur minimum	8000
-1	FFFF

les compléments ont pour premier chiffre 8, 9, A, B, C, D, E ou F; les compléments sont obtenus en soustrayant chaque chiffre de 15 (F) et en ajoutant 1 au résultat final. Par exemple, sur 32 bits :

Complément(12345678) => EDCBA987 + 1 => EDCBA988

Exemples:

```
(32 + 16)    00000020        (32-16)     00000020
           + 00000010                  + FFFFFFF0
             00000030                    00000010

(16-32)      00000010
           + FFFFFFE0
             FFFFFFF0
```

4.2 Exercices

4.2.1 Avec des mots d'une longueur de 16 bits et en utilisant la représentation par compléments à deux, donnez les nombres décimaux équivalents aux nombres hexadécimaux suivants:

a. 1234 b. FADE c. BABA d. D2BC e. 7AB3 f. 564E

4.2.2 Donner l'équivalent hexadécimal (sur 16 bits) des nombres décimaux suivants:

a. 1234 b. -1048 c. 1492 d. 111 e. -777 f. -10

4.2.3 Effectuer les opérations suivantes (sur 16 bits):

```
a.   OA94    b.    B747   c.  2BC3   d.    6789    e.  0522    f.  CAFE
   + 6421       + CAFE     + 1234       + 9876      - 3502      - FADE
```

4.2.4 Effectuer les soustractions suivantes en utilisant la méthode des compléments. (sur 32 bits)

```
a.        54321768     b.     00F9A920        c.      7FFF0123
        - 12345678           - 32145678              - 00012345
```

4.3 Addition

L'organe arithmétique utilise un opérande placé dans un accumulateur (registres A ou X) et un second opérande placé soit dans l'instruction, soit dans la mémoire centrale. Le résultat de l'opération est rangé dans l'accumulateur, détruisant ainsi la valeur de l'opérande qui s'y trouvait. Soit à calculer:

```
Somme = Z + Y;
```

Pour pouvoir effectuer une opération arithmétique (ici une addition), l'un des opérandes doit être nécessairement placé dans l'accumulateur; il faut donc, avant de pouvoir effectuer l'opération, aller chercher la donnée correspondante en mémoire centrale et la placer dans l'accumulateur. L'instruction Assembleur qui permet de faire ceci est l'instruction *LOAD* qui copie dans l'accumulateur une valeur prise dans la mémoire; la valeur en mémoire demeure inchangée tandis que la valeur antérieure de l'accumulateur est détruite.

```
LDA     Z,d  ; Accumulateur = Z;
```

Une fois la valeur ainsi placée dans l'accumulateur, on peut exécuter l'addition au moyen de l'instruction suivante:

```
ADDA    Y,d
```

qui ajoute la valeur de la variable Y à la valeur de l'accumulateur et range le résultat dans l'accumulateur. Il reste alors à ranger ce résultat en mémoire, ce qui peut être fait en utilisant l'instruction *STORE* qui copie la valeur de l'accumulateur dans le mot mémoire indiqué en détruisant la valeur antérieure, l'accumulateur demeurant inchangé.

```
STA     Somme,d
```

On aurait pu faire le calcul en utilisant le registre X ; les instructions seraient alors `LDX`, `ADDX` et `STX`.

4.4 Résultats intermédiaires

Lorsqu'on effectue des calculs, il arrive souvent que le résultat d'une étape de calcul soit utilisé directement dans la phase suivante du calcul. Par exemple pour effectuer le calcul:

```
Somme = W + X + Y + Z;
```

on fait la somme des deux premiers nombres à laquelle on ajoute le troisième nombre puis le quatrième (d'où le nom *accumulateur*!). On peut faire ceci comme l'indique la Figure 4.1.

On notera l'utilisation de la pseudo-instruction (ou directive) .WORD pour réserver des mots mémoire.

```
        LDA     W,d       ; W
        ADDA    X,d       ; W + X
        ADDA    Y,d       ; W + X + Y
        ADDA    Z,d       ; W + X + Y + Z
        STA     Somme,d   ; Ranger dans Somme
                …
W:      .WORD   10      ; int W = 10;
X:      .WORD   20      ; int X = 20;
Y:      .WORD   30      ; int Y = 30;
Z:      .WORD   40      ; int Z = 40;
Somme:  .WORD   0       ; int Somme = 0;
```

Figure 4.1 Calcul d'une somme

Il se peut que l'on désire conserver les résultats intermédiaires pour un traitement ultérieur; dans ce cas, une fois la somme intermédiaire calculée elle sera rangée dans un mot mémoire supplémentaire.

Ainsi, si l'on désire conserver pour utilisation ultérieure (calcul, impression) les sommes des deux premières données et les sommes des deux dernières on aura un programme légèrement différent, illustré par la figure 4.2.

```
        LDA     W,d       ; W
        ADDA    X,d       ; W + X
        STA     Som1,d    ; Som1 = W + Y
        LDA     Y,d       ; Y
        ADDA    Z,d       ; Y + Z
        STA     Som2,d    ; Som2 = Y + Z
        ADDA    Som1,d    ; Som1 + Som2
        STA     Somme,d   ; Ranger dans Somme
        …
W:      .WORD   10        ; int W = 10;
X:      .WORD   20        ; int X = 20;
Y:      .WORD   30        ; int Y = 30;
Z:      .WORD   40        ; int Z = 40;
Somme:  .WORD   0         ; int Somme = 0;
Som1:   .WORD   0         ; int Som1 = 0;
Som2:   .WORD   0         ; int Som2 = 0;
```

Figure 4.2 Calcul de sommes

4.5 Exercices

4.5.1 Lesquelles de ces instructions modifient l'accumulateur? Lesquelles modifient la mémoire? Lesquelles modifient les deux?

```
LDA     X,d
STA     X,d
ADDA    X,d
```

4.5.2 Étant donné les instructions suivantes, indiquer quels seront les contenus de l'accumulateur et du mot mémoire Z après ou avant exécution de chaque instruction.

		Avant		Après	
		(A)	(Z)	(A)	(Z)
LDA	Z,d	123	456		
ADDA	Z,d	123	321		
LDA	Z,d	123			456
LDA	Z,d	123		456	
ADDA	Z,d			456	123
STA	Z,d		123		456

4.5.3 Écrire les instructions permettant de calculer la somme de trois nombres et le triple de cette somme.

4.6 Soustraction

La soustraction est semblable à l'addition. Si l'on utilise la notation c(A) pour indiquer le contenu de l'accumulateur et le signe => pour indiquer l'affectation d'une valeur à une variable, l'instruction de soustraction aura la forme et la signification suivante:

```
SUBA  Y,d          ; c(A) - c(Y) => A
```

La valeur antérieure de l'accumulateur est perdue. Prenons comme exemple un calcul simple de paye comme:

```
Net = Paye + Sup - Impôt;
```

Ceci se traduira en assembleur par l'exemple de la figure 4.3.

```
        LDA     Paye,d
        ADDA    Sup,d
        SUBA    Impot,d
        STA     Net,d
         …
Paye:   .WORD   310     ; int Paye = 310;
Sup:    .WORD   50      ; int Sup = 50;
Impot:  .WORD   60      ; int Impot = 60;
Net:    .WORD   0       ; int Net = 0;
```

Figure 4.3 Calcul simple de paye

La pseudo-instruction ou directive .WORD permet de réserver l'espace pour un mot en mémoire et d'y placer des valeurs initiales. Lorsqu'on exécute les instructions ci-dessus l'accumulateur prend successivement les valeurs suivantes: 310, 360 et 300.

Les ordinateurs de deuxième génération (début des années soixante du siècle dernier) donnaient au programmeur deux accumulateurs (généralement appelés AC et MQ), ce qui était la caractéristique la plus restrictive de ces ordinateurs. Il fallait alors continuellement transférer de l'information entre accumulateurs et mémoire. Les ordinateurs de la troisième génération et les ordinateurs actuels possèdent un bien plus grand nombre d'accumulateurs, qui sont maintenant appelés registres, 16 étant le nombre le plus courant. PEP 8, cependant, ne nous offre que deux registres, allongeant ainsi les programmes en nécessitant plus d'instructions de rangement.

Les processeurs modernes possèdent ainsi un nombre de registres plus élevé, dont le rôle n'est pas limité aux opérations arithmétiques. Ces registres de 32 ou 64 bits sont généralement divisés en deux catégories: les registres de données et les registres d'adresse. Les opérations que l'on peut appliquer aux registres de données et aux registres d'adresse diffèrent quelque peu selon les machines.

4.7 Codes de condition

Nous avons déjà parlé au chapitre 2 des codes de condition. Bien que certaines instructions ne les affectent en rien, un grand nombre d'instructions leur donnent des valeurs indiquant le résultat de leur exécution. C'est le cas en particulier des instructions arithmétiques. Les instructions permettant de vérifier la valeur du code de condition seront vues ultérieurement.

Comme nous l'avons déjà vu, les codes de condition sont représentés par 4 bits qui sont identifiés par les lettres N, Z, V et C.

- N: négatif. Mis à un si résultat négatif, mis à zéro autrement.
- Z: zéro. Mis à 1 si résultat nul, mis à zéro autrement.
- V: débordement (« overflow »). Mis à 1 si débordement avec signe, à zéro autrement.
- C: retenue (« carry »). Mis à 1 si retenue, à zéro autrement.

Les instructions arithmétiques ADD et SUB affectent les codes de condition et leur donnent les valeurs correspondant au résultat de l'opération. De même, l'instruction LDA ne change jamais la valeur des indicateurs C et V, et donne des valeurs à Z et N qui décrivent la valeur déplacée.
La description détaillée des instructions de PEP 8 indique toujours l'effet de l'exécution de l'instruction sur les codes de condition (voir chapitre 7).

4.8 Limites

Toutes les opérations que l'on vient de voir ne concernent que l'arithmétique entière. Notre exemple de paye, pour être plus réaliste, devrait utiliser des nombres avec partie fractionnaire. Cependant les instructions ADD et SUB sont limitées à l'arithmétique entière.

Lorsque le résultat de toute opération arithmétique entière doit tenir dans un mot mémoire, toute valeur en dehors de l'intervalle $[-2^{15}, 2^{15}[$ aboutira à un résultat vide de sens: il y aura débordement.

Exemples :	$(2^{15} - 1)$	7FFF	
	+ (1)	0001	
	(-2^{15})	8000	?
de même	(-2^{15})	8000	
	+ (- 1)	FFFF	
	$(2^{15} - 1)$	7FFF	?

4.9 Exemple de programme

Si nous reprenons le premier programme vu au chapitre 1 en le redonnant ici dans la figure 4.4, et si nous le traduisons nous mêmes en langage d'assemblage PEP 8, nous obtenons le programme bien plus court et compréhensible (si vous le comparez à la figure 1.2) de la figure 4.5.

```
//  Ce programme calcule et affiche le premier nombre de Fibonacci
//  supérieur à 500.
//    Philippe Gabrini     mai 2005
//
#include <iostream>
using namespace std;

int main()
{
  const int LIMITE = 500;
  int avantDernier = 0, dernier = 1, somme = 1;

  while(somme < LIMITE) {
    avantDernier = dernier;
    dernier = somme;
    somme = avantDernier + dernier;
  }
  cout << "Le premier nombre de la suite de Fibonacci supérieur à "
       << LIMITE << " est " << somme << endl;
  return 0;
}
```

Figure 4.4 Programme de la suite de Fibonacci en C++

Programme Assembleur PEP 8

Pour résoudre le même problème de la suite de Fibonacci, nous avons écrit un programme en assembleur PEP 8. La liste d'assemblage complète de ce programme produite par l'assembleur est donnée à la figure 4.5. Ce programme est plus simple que celui de la figure 1.2 car il n'est pas encombré des éléments et des détails de fonctionnement d'un système complet comme celui de C++. Les instructions nécessaires à la traduction de l'algorithme sont cependant plus claires puisqu'elles utilisent directement les variables `avant`, `dernier` et `somme`.
L'adresse de départ est l'adresse de la première instruction (étiquette Fibo); les instructions d'adresses relatives 3 à 1B correspondent à la boucle de calcul du premier nombre de la suite de Fibonacci supérieur à 500 , déjà vue à la figure 3.8. Les instructions d'adresses relatives 1E à 28 sont les instructions permettant d'afficher le résultat et de terminer l'exécution du programme.

Les variables sont déclarées à un endroit où elles sont séparées du programme et ne courent pas le risque d'être prises pour des instructions et exécutées puisqu'elles suivent l'instruction STOP, laquelle arrête effectivement l'exécution du programme. L'entrée-sortie n'est généralement pas quelque chose

de simple en langage d'assemblage; l'avantage de PEP 8 est de nous offrir des instructions d'entrée-sortie simples et on utilise ici les instructions STRO, DECO et CHARO pour afficher les résultats à l'écran.

```
Addr  Code   Symbol   Mnemon  Operand    Comment
             ;    Trouve et imprime le premier terme de la suite de Fibonacci
             ;    supérieur à 500.
             ;       Ph. Gabrini    septembre 2005
             ;
0000  C1002C Fibo:    LDA     somme,d    ;
0003  B001F4 Boucle:  CPA     500,i      ; while(somme < 500)
0006  0E001E          BRGE    Affiche    ; {
0009  C1002A          LDA     dernier,d  ;
000C  E10028          STA     avant,d    ;   avant = dernier;
000F  C1002C          LDA     somme,d    ;
0012  E1002A          STA     dernier,d  ;   dernier = somme;
0015  710028          ADDA    avant,d    ;
0018  E1002C          STA     somme,d    ;   somme = avant + dernier;
001B  040003          BR      Boucle     ; }//while
001E  41002E Affiche: STRO    msg1,d     ; cout << "Premier ... "
0021  39002C          DECO    somme,d    ;      << somme
0024  50000A          CHARO   '\n',i     ;      << endl;
0027  00              STOP               ;
0028  0000   avant:   .WORD   0          ; int avant = 0;
002A  0001   dernier: .WORD   1          ; int dernier = 1;
002C  0001   somme:   .WORD   1          ; int somme = 1;
002E  507265 msg1:    .ASCII  "Premier terme de la suite de Fibonacci > 500: \x00"
      6D6965
      722074
      65726D
      652064
      65206C
      612073
      756974
      652064
      652046
      69626F
      6E6163
      636920
      3E2035
      30303A
      2000
                    .END

Symbol    Value
Fibo       0000
Boucle     0003
Affiche    001E
avant      0028
dernier    002A
somme      002C
msg1       002E


No errors. Successful assembly.
```

Figure 4.5 Liste du programme Assembleur PEP 8 de la suite de Fibonacci

