ALICE

PROGRAMMATION EN ASSEMBLEUR

ALICE

PROGRAMMATION EN ASSEMBLEUR

FAGOT-BARRALY

SYBEX

Paris • Berkeley • Düsseldorf • Londres

ISBN : 2-7361-0120-1

INTRODUCTION

Ce livre est destiné à tous ceux qui connaissent les éléments de base de la programmation BASIC d'Alice et qui souhaitent maintenant franchir l'étape suivante, l'étape du langage machine et de sa forme plus compréhensible : l'assembleur.

Il existe des ouvrages très bien faits qui traitent de ce sujet mais la plupart d'entre eux s'adressent à des lecteurs déjà initiés et ne s'intéressent que très peu à ceux qui font leurs premiers pas dans ce domaine. Aussi avons-nous considéré qu'il y avait place dans le monde du livre pour un ouvrage qui prendrait les débutants par la main et guiderait, avec mille égards, leur entrée dans l'univers fascinant des microprocesseurs.

Tous les exemples que nous proposerons pourront être testés immédiatement sur n'importe quelle version d'Alice possédant l'*éditeur-assembleur*. Et ainsi le lecteur pourra, au fur et à mesure, mettre en pratique les connaissances qu'il viendra d'acquérir. Très vite il sera à même de comprendre comment obtenir de son ordinateur des sons, un graphisme et des couleurs d'une richesse exceptionnelle.

Le Chapitre 1 donnera les bases indispensables de l'arithmétique binaire car, ne l'oublions pas, un ordinateur ne connaît en réalité pas autre chose que les chiffres 0 et 1.

Le Chapitre 2 rappellera comment est conçue la mémoire écran d'Alice. Cette étude est rendue nécessaire par le fait que la majorité des programmes écrits en langage machine sont des animations de type vidéo. On trouvera aussi dans ce chapitre un programme de démonstration qui permettra de voir la différence flagrante dans les vitesses d'exécution d'un programme BASIC et de son équivalent assembleur.

Le Chapitre 3 nous fera pénétrer au cœur du microprocesseur : c'est un chapitre consacré aux différents registres, registres dont la connaissance est obligatoire pour aborder la suite de ce livre. Nous trouverons aussi dans ce chapitre l'étude des différentes façons d'utiliser une instruction assembleur suivant le mode d'adressage choisi.

Le Chapitre 4 sera consacré à l'étude de notre premier programme écrit en assembleur : les moindres détails seront expliqués.

Le Chapitre 5 analysera les principales instructions nécessaires à la programmation du microprocesseur d'Alice. De nombreux exemples,

toujours avec les explications correspondantes, seront fournis. Lorsque cela présentera un intérêt, nous montrerons comment l'on peut relier l'un à l'autre le BASIC et le langage machine. Nous avons enchaîné l'étude des diverses instructions sans nous soucier d'un quelconque ordre logique ou alphabétique : c'est la seule notion de progressivité qui nous a guidés.

Notre principal souhait est d'avoir fait un livre accessible facilement, un livre que l'on ne referme pas au bout de quelques pages devant la supposée trop grande ampleur de la difficulté.

1

L'ARITHMÉTIQUE BINAIRE

LES SYSTÈMES DE NUMÉRATION

1. La base dix

Le système de numération à base 10 est le système que nous utilisons dans la vie de tous les jours. On le connaît sous le nom de système décimal et c'est le nombre 10 qui y joue le rôle primordial.

Pour commencer notre étude rappelons ce que valent les premières puissances de 10 :

$10^0 = 1$
$10^1 = 10$
$10^2 = 10 \times 10 = 100$
$10^3 = 10 \times 10 \times 10 = 1000$

Par convention, n'importe quel nombre avec l'exposant 0 est égal à 1, et 10 n'échappe pas à la règle : $10^0 = 1$.

A l'aide de ces puissances, il est possible d'écrire un quelconque nombre entier.

$2548 = 2000 + 500 + 40 + 8$
Or $2000 = 2 \times 1000 = 2 \times 10^3$
$500 = 5 \times 100 = 5 \times 10^2$
$40 = 4 \times 10 = 4 \times 10^1$
$8 = 8 \times 1 = 8 \times 10^0$
ce qui donne : $2548 = 2 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 8 \times 10^0$.

D'une manière analogue, on aura :

$4706 = 4000 + 700 + 6$
soit : $4706 = 4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 6 \times 10^0$.

$10^3 = 1000$	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
2	5	4	8
4	7	0	6

Naturellement, il nous est loisible de choisir des nombres plus grands car il suffira de prendre des puissances de 10 avec un exposant supérieur.

Rien de bien compliqué dans tout cela. Passons à l'étude d'une autre base mais, auparavant, notons bien quelque chose que nous retrouverons dans tout ce chapitre : les chiffres utilisés en base 10 vont de 0 à 9 ; ils sont tous inférieurs à cette base.

2. La base cinq

Les puissances de 5 se calculent facilement : $5^0 = 1$; $5^1 = 5$; $5^2 = 25$; $5^3 = 125$. Pour écrire un nombre en base 5, il va falloir constituer un tableau analogue au précédent mais, bien entendu, sa première ligne sera écrite avec les puissances de 5. Soit par exemple à traduire 138 dans le système à base 5 :

$5^3 = 125$	$5^2 = 25$	$5^1 = 5$	$5^0 = 1$
1	0	2	3

On a cherché combien de multiples de 125 (5^3) étaient contenus dans 138 :

1 fois et il reste 13 : $138 = 1 \times 125 + 13$.

Puis on a cherché combien de fois on pouvait faire rentrer 25 (5^2) dans 13 :

0 fois et il reste toujours 13 : $138 = 1 \times 125 + 0 \times 25 + 13$.

Il a fallu alors chercher combien de fois allait rentrer 5 (5^1) dans 13 :

2 fois et il reste 3 : $138 = 1 \times 125 + 0 \times 25 + 2 \times 5 + 3$.

Dernière phase de l'opération : dans le reste qui vaut à ce moment-là 3, combien de fois peut-on faire rentrer 1 (5^0) ?

3 fois et il ne reste rien : $138 = 1 \times 125 + 0 \times 25 + 2 \times 5 + 3 \times 1$.

On a donc en résumé :

$138 = 1 \times 5^3 + 0 \times 5^2 + 2 \times 5^1 + 3 \times 5^0$
et on en déduit que 138 s'écrit 1023 en base 5.

Prenons un deuxième exemple : quelle est la valeur de 279 en base 5 ?

$5^3 = 125$	$5^2 = 25$	$5^1 = 5$	$5^0 = 1$
2	1	0	4

$279 = 2 \times 125 + 1 \times 25 + 0 \times 5 + 4 \times 1$
ou $279 = 2 \times 5^3 + 1 \times 5^2 + 0 \times 5^1 + 4 \times 5^0$

Par suite 279 s'écrit 2104 en base 5.

En pratique, pour écrire un nombre décimal dans une autre base, on utilise le plus souvent la méthode dite des *divisions successives*.

```
279 | 5
  4 | 55 | 5
    0 | 11 | 5
      1 | 2 | 5
        2 | 0
```

Elle consiste à diviser le nombre par 5 puis le quotient par 5 puis le nouveau quotient obtenu par 5 et ceci jusqu'à ce que le dernier quotient soit nul. Il ne reste plus alors qu'à écrire la liste des différents restes en prenant la précaution essentielle de les copier dans l'ordre inverse. Les restes, dans notre exemple, étant 4,0,1,2 on écrit alors : 279 = 2104 (base 5).

Si nous avons maintenant à traduire en décimal un nombre déjà écrit en base 5, il faudra inscrire ce nombre dans un tableau conçu comme les précédents et ensuite le calculer.

Soit à écrire 3421 (base 5) en base 10.

$5^3 = 125$	$5^2 = 25$	$5^1 = 5$	$5^0 = 1$
3	4	2	1

On en déduit que 3421 (base 5) = $3\times5^3 + 4\times5^2 + 2\times5^1 + 1\times5^0$. Et l'on obtient : 3421 (base 5) = $3\times125 + 4\times25 + 2\times5 + 1\times1$ = 486.

Remarquons, pour terminer, que les seuls chiffres utilisés en base 5 sont 0, 1, 2, 3, 4.

Il est conseillé au lecteur de s'assurer, avec quelques exercices dont il aura pris les nombres au hasard, que tout ce qui a été vu est bien assimilé. Non pas que la base 5 ait une quelconque importance en informatique, mais elle permet de comprendre sans peine les mécanismes des systèmes de numération.

3. La base deux

Nous arrivons maintenant au cœur du problème : voici le système de numération (dit système binaire) qu'utilisent les ordinateurs.

Tout d'abord, les puissances de 2 : $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$.

Puis maintenant, un exemple : on décide d'écrire 23 en binaire :

$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
1	0	1	1	1

La plus grande puissance de deux qui rentre dans 23 est 16 (2^4) : le reste est 7. Peut-on ensuite faire rentrer 8 (2^3) dans 7 : la réponse est non et le chiffre 0 a été placé dans la case correspondante.

Par contre 4 (2^2) est contenu dans 7 : on écrit le chiffre 1 dans la troisième case et on note le nouveau reste : 3.

2 (2^1) étant plus petit que 3, on écrit le chiffre 1 dans la quatrième case et puisque le reste vaut alors 1, il nous faut encore écrire 1 mais cette fois-ci dans la dernière colonne.

23 = 10111 (base 2).

Heureusement pour nous, la méthode des divisions successives par 2 va nous donner la réponse d'une manière plus sûre et plus rapide :

```
23 | 2
 1 | 11 | 2
     1 | 5 | 2
       1 | 2 | 2
         0 | 1 | 2
           1 | 0
```

23 = 10111 (2)

Voici d'autres exemples dont les calculs intermédiaires seront laissés à la charge du lecteur :

34 = 100010 (2)
150 = 10010110 (2)
255 = 11111111 (2)

Il reste à voir comment passer de la base 2 à la base 10.

Admettons que l'on veuille écrire 1111011 en décimal. On reconstitue le tableau dans lequel sont indiquées les puissances de 2 et on y écrit notre nombre :

$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
1	1	1	1	0	1	1

On passe plus de temps à faire le tableau qu'à obtenir la réponse !

1111011 = 64 + 32 + 16 + 8 + 2 + 1 = 123 (décimal)

Il est nécessaire de remarquer que, dans ce que nous venons de voir, les seuls chiffres utilisés sont le 0 et le 1, à savoir les chiffres inférieurs à la base.

4. La base seize

C'est le système (appelé hexadécimal ou hexa) dont les informaticiens ne peuvent se passer, alors qu'au premier abord on pourrait se demander ce que vient faire son étude dans ce livre.

Les 16 chiffres nécessaires à l'écriture dans cette base sont tout d'abord 0,1,2,3,4,5,6,7,8,9 ...

Mais après le 9, le 10 peut-être ? Mais non, puisque c'est un nombre. Comme il nous manque 6 chiffres, on les a remplacés par les premières lettres de l'alphabet.

Chiffres	A	B	C	D	E	F
Valeurs	10	11	12	13	14	15

Ainsi 12 s'écrit C, 14 s'écrit E.

Là encore, les méthodes de conversion étudiées dans les paragraphes précédents vont s'appliquer.

Soit à écrire 300 en base 16 : les divisions successives doivent se faire par 16.

```
300 | 16
  C | 18 | 16
      2 |  1 | 16
          1 |  0
```

300 = 12C (hexa)

Passons à un autre exemple après avoir remarqué que le reste de la première division, qui valait 12, a été remplacé par C.

```
5032 | 16
   8 | 314 | 16
       A |  19 | 16
           3 |  1 | 16
               1 |  0
```

5032 = 13A8 (hexa)

Si l'on souhaite traduire en décimal un nombre déjà écrit en base 16, on utilise les puissances de 16 :

$16^0 = 1$ $16^1 = 16$ $16^2 = 256$ $16^3 = 4096$

3D4F (hexa) s'écrit $3\times16^3 + D\times16^2 + 4\times16^1 + F\times16^0$

donc $3D4F = 3\times4096 + 13\times256 + 4\times16 + 15$

soit 3D4F = 15695 (base décimale).

Les utilisateurs d'Alice ne disposent pas de la fonction HEX$ qui donne la valeur hexadécimale d'un nombre décimal. Voici donc un court programme qui remédiera à cette lacune :

```
10 H$ = "" : INPUT "NOMBRE A CONVERTIR" ; N
20 D = INT(N/16) : R=N-16*D : H$ = CHR$(R+48-7*(R>9))+H$
30 IFD THENN=D : GOTO 20
40 PRINT "REPONSE " ; H$ : GOTO 10
```

Il nous faut maintenant comprendre où réside l'intérêt en informatique du système hexadécimal et pour cela comparer les représentations d'un même nombre décimal suivant que l'on veuille l'écrire en base 2 ou en base 16.

183 (décimal) = 1011 0111 (binaire)

183 (décimal) = B 7 (hexa)

On sépare les huit chiffres binaires en deux groupes de quatre :

1011 et 0111

Or 1011 = $1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 8+2+1 = 11$ (décimal)

Et 0111 = $0\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 4+2+1 = 7$ (décimal)

En remarquant que 11 décimal s'écrit B en hexadécimal, on voit de façon immédiate la correspondance entre les bases 2 et 16. Il est tout à fait possible de passer directement de la base 2 à la base 16 sans avoir à connaître précisément le nombre décimal dont il s'agit.

Essayons encore en partant du nombre décimal 143 qui s'écrit 10001111 en base 2 :

1000 1111 = 8F en hexadécimal

8 F

Bien entendu, on passera tout aussi facilement de la base 16 à la base 2 en ayant bien à l'esprit le tableau suivant.

Décimal	Binaire	Hexadécimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Que peut bien valoir par exemple en binaire le nombre hexadécimal 4A ?

Réponse : $\underbrace{0100}_{4}\ \underbrace{1010}_{A}$

Et le nombre hexadécimal 37E ?

Réponse : $\underbrace{0011}_{3}\ \underbrace{0111}_{7}\ \underbrace{1110}_{E}$

Dans ce dernier exemple, les deux premiers chiffres 0 sont inutiles et ne servent qu'à la compréhension de la règle qu'il faudra toujours respecter : le partage du nombre binaire doit se faire par groupes de quatre et ceci toujours en partant de la droite.

En utilisant le programme qui remplace la fonction HEX$, il nous sera possible d'éviter la tâche fastidieuse de conversion d'un nombre en binaire et cela grâce à l'utilisation intermédiaire de la base 16.

Admettons que l'on veuille convertir 1000 (décimal) en binaire :

```
NOMBRE A CONVERTIR ? 1000
Réponse : 3E8
```


On en déduit sans peine le résultat recherché :

$\underbrace{0011}_{3}\ \underbrace{1110}_{E}\ \underbrace{1000}_{8}$

OPÉRATIONS DANS LES BASES 2 ET 16

Nous nous limiterons dans ce paragraphe à l'addition et à la soustraction.

1. Addition

On considère le mécanisme de l'addition dans notre système décimal et on l'applique à la base 2.

```
  207
+ 321
------
= 528
```

Dans cet exemple, les chiffres de chaque colonne s'ajoutent : comme on n'atteint jamais 10 (la base, ne pas l'oublier), il n'y a aucun problème. Il va en être de même dans les additions binaires suivantes car les totaux ne dépasseront jamais 2 (valeur de la base binaire) :

```
  101100          100011
+ 010001        + 000100
--------        --------
= 111101        = 100111
```

Reste à voir le cas de la retenue :

```
  1
  447
+ 223
-----
= 670
```

Dans cette addition décimale, 7 et 3 donnent 10, c'est-à-dire très précisément la valeur de la base. On écrit alors 0 au-dessous des chiffres

7 et 3 puis on reporte 1 dans la colonne suivante. Nous procéderons exactement de la même façon avec le système binaire.

```
    1
  10001
+ 01001
-------
= 11010
```

La somme des deux chiffres de droite donne 2 (valeur de la base). Le dernier chiffre du résultat sera donc un 0 et la retenue 1 sera écrite en haut de la colonne suivante. Le reste des calculs s'effectue ensuite sans difficulté.

Essayons encore :

```
    1 1
  100101
+ 000101
--------
= 101010
```

Il n'y a rien à redire, passons à un autre exemple :

```
    11
  101011
+ 010011
--------
= 111110
```

La somme des deux chiffres de droite donnant 2, on a écrit 0 comme dernier chiffre pour la réponse et on a retenu 1. L'addition de cette retenue avec les deux chiffres 1 de la deuxième colonne donne alors 3, ce qui se traduit par l'écriture du chiffre 1 dans la réponse et la pose d'une retenue en haut de la troisième colonne.

Il faut bien reconnaître que le risque d'erreur n'est pas négligeable lorsque l'on a à effectuer des calculs en binaire. Aussi, une méthode souvent utilisée consiste à traduire les nombres en hexadécimal, à les ajouter alors, puis à reconvertir si nécessaire le résultat en base 2.

Décidons de faire en hexadécimal les additions suivantes :

```
  34 B5
+ 66 1 4
--------
= 9AC9
```

```
     1
  5264
+ A32E
-------
= F592
```


Si l'on se souvient de la correspondance :

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

on comprend directement comment la première opération a été faite.

5 + 4 = 9
B + 1 = 11 + 1 = C
4 + 6 = 10 = A
3 + 6 = 9

Pour ce qui concerne la deuxième addition, les choses se décomposent de la manière suivante :

4 + E = 4 + 14 = 18.

La retenue qui correspond à 10 dans notre système habituel est égale à 16 dans le système hexadécimal. Ce qui fait qu'après avoir posé la retenue en haut de la deuxième colonne, il restera 2 à écrire comme chiffre de droite de la réponse, réponse qui se complète ensuite par :

1 + 6 + 2 = 9
2 + 3 = 5
5 + A = 5 + 10 = 15 = F

Autres exemples :

```
    1 1
  4BC3
+ 2A2F
-------
= 75F2
```

```
    111
  FFFF
+ FFFF
-------
= 1FFFE
```

Nous sommes bien d'accord, n'est-ce pas, dans le système décimal il n'y a de retenue qu'à partir de 16.

2. Soustraction

Gardons le système précédent de numération et intéressons-nous au calcul d'une différence :

```
  9AE7
- 49B3
-------
= 5134
```

C'est, somme toute, plutôt facile à comprendre :

7 – 3 = 4

E − B = 14 − 11 = 3
A − 9 = 10 − 9 = 1
9 − 4 = 5

Alors, essayons les retenues :

```
  9B54
− 6A29
-------
= 312B
```

On a tendance à dire 9 ôté de 14, la force de l'habitude nous faisant rajouter une dizaine à 4. En réalité, puisque nous sommes en hexadécimal, ce n'est pas dix que l'on doit ajouter à 4 mais seize. Il s'agit, du coup, de faire 9 ôté de 20 : reste 11 c'est-à-dire B. Naturellement, la retenue ne doit pas être perdue dans la suite des calculs.

5 − 3 (dont 1 de retenue) = 2
B − A = 11 − 10 = 1
9 − 6 = 3

Deux autres exemples :

```
  4A85          ABCD
− 1F2E        − 2FFF
-------       -------
= 2B57        = 7BCE
```

On en arrive maintenant au calcul de la différence entre deux nombres écrits dans le système binaire. La méthode de soustraction directe peut être employée :

```
  101011
− 001001
--------
= 100010
```

Mais les programmeurs lui préfèrent une autre méthode, celle dite du *complément à deux*, car on comprend bien que la soustraction qui vient d'être effectuée aurait été plus compliquée si des retenues étaient apparues.

3. Le complément à deux

Considérons le nombre décimal 17.

Sa conversion en binaire donne 10001. Pour obtenir le complément à 2 de ce nombre, on respecte les trois étapes suivantes :

- on écrit notre nombre sur huit chiffres en rajoutant des 0 devant :
  00010001
- on remplace chaque 0 par 1 et chaque 1 par 0 :
  11101110
- on ajoute 00000001 à ce résultat :
  11101111

Le nombre que l'on obtient est appelé le complément à 2, sur huit chiffres, de 17, et l'ordinateur considérera que c'est l'opposé de 17, c'est-à-dire le nombre −17. Oui, vous avez bien lu, dans le mode complément à 2, le nombre binaire 11101111 est égal à −17 !

Comment s'en assurer ? En partant de l'idée toute simple qui consiste à dire : puisque, en ajoutant 17 et son opposé −17, on obtient 0, on doit normalement, en ajoutant 00010001 et 11101111, obtenir aussi 0.
Voyons cela :

```
  11111111
  00010001
+ 11101111
----------
= (1)00000000
```

Les deux chiffres 1 de la droite font apparaître une retenue que l'on retrouve ensuite de colonne en colonne. Il faut tout de même noter qu'il ne doit pas être tenu compte de la dernière retenue et que nous prendrons l'habitude de la négliger. Nous verrons bientôt que l'ordinateur ne procède pas autrement : pour lui aussi, la dernière retenue de gauche «tombe à l'eau».
Un autre exemple : essayons d'écrire −50 en binaire sous la forme complément à 2 :

```
00110010    50 décimal
11001101    chiffres inversés
11001110    ajout de 1
```

Donc −50 s'écrit 11001110 en binaire
ou C E en hexadécimal

Voici, tels quels, quelques résultats qui doivent permettre au lec-

teur d'assimiler parfaitement la façon dont l'ordinateur écrit les nombres négatifs :

−5 (décimal) = 11111011 (binaire) = FB (hexa)
−20 (décimal) = 11101100 (binaire) = EC (hexa)
−100 (décimal) = 10011100 (binaire) = 9C (hexa)

Avant de passer à autre chose, revenons quelques minutes sur la façon dont on s'y prendra pour faire une différence binaire maintenant que nous savons utiliser la technique du complément à 2.

Soit à calculer 101000 − 10111.

On cherche l'opposé du deuxième terme de la soustraction en mode complément à 2 : on obtient 11101001.

Il reste alors à ajouter le premier terme avec l'opposé du deuxième :

```
   111 1
    00101000
+   11101001
------------
= (1)00010001
```

La réponse est la suivante : 101000 − 10111 = 10001.

OPÉRATEURS LOGIQUES

En dehors des calculs arithmétiques habituels, on peut effectuer sur les nombres binaires des opérations d'un type spécial que l'on appelle les opérations logiques. Elles ne présentent aucune difficulté car en aucun cas ne se pose le problème des retenues.

1. Le OU logique

Cette opération respecte les règles suivantes :

```
  0      0      1      1
OU 0   OU 1   OU 0   OU 1
---    ---    ---    ---
= 0    = 1    = 1    = 1
```

C'est la même chose avec des nombres binaires plus grands :

```
  101101          101000
OU 110101       OU 001100
---------       ---------
= 111101        = 101100
```


Alice dispose d'une instruction qui effectue ce type de calculs : c'est le mot clé OR. Demandons-lui quelques résultats :

```
PRINT 46 OR 100 ; réponse : 110
   101110 <------ 46
OR 1100100 <----- 100
---------
=  1101110 <----- 110

PRINT 50 OR 0 ; réponse : 50
   110010 <------ 50
OR 000000 <------ 0
---------
=  110010 <------ 50
```

L'opérateur OR va nous servir en assembleur car il permet de forcer l'un des chiffres binaires à passer à 1. Voyons comment :

```
PRINT 82 OR 1 ; réponse : 83
   1010010 <------ 82
OR 0000001 <------ 1
---------
=  1010011 <------ 83

PRINT 91 OR 1 ; réponse : 91
   1011011 <------ 91
OR 0000001 <------ 1
---------
=  1011011 <------ 91
```

Dans le premier exemple, on part d'un nombre dont le dernier chiffre binaire (bit 0) est égal à 0. Après utilisation de OR 1, ce dernier chiffre a été porté à 1 sans qu'aucun des autres chiffres ait été modifié.

Dans le deuxième cas, on est parti d'un nombre qui se terminait déjà par un 1. OR 1 n'a modifié ni ce chiffre ni naturellement aucun des autres. On en conclut donc que si l'on effectue OR 1 avec n'importe quel nombre, on aura un résultat dont le dernier chiffre (bit 0) vaudra obligatoirement 1.

D'une façon analogue, en calculant OR 4 avec n'importe quel nombre, on sera certain que le troisième chiffre en partant de la droite est un 1 (bit 2) :

PRINT 19 OR 4 ; réponse : 23

```
  10011 <------ 19
OR 00100 <------ 4
-------
=  10111 <------ 23
```

Le troisième chiffre est bien passé à 1.

PRINT 52 OR 4 ; réponse : 52

```
  110100 <------ 52
OR 000100 <------ 4
--------
=  110100 <------ 52
```

Le troisième chiffre est resté à 1.

2. Le ET logique

Le ET logique est défini par les règles suivantes :

```
  0      0      1      1
ET 0   ET 1   ET 0   ET 1
---    ---    ---    ---
= 0    = 0    = 0    = 1
```

Quelques exemples :

```
   101100          101000
ET 011001       ET 110111
--------        --------
= 001000        = 100000
```

On peut faire faire ces calculs par l'ordinateur et cette fois, c'est le mot réservé AND qui va nous servir.

PRINT 30 AND 40 ; réponse : 8

```
    11110 <------ 30
AND 101000 <------ 40
-------
=  001000 <------ 8
```

On retrouve l'instruction AND en assembleur car, grâce à elle, nous pouvons mettre à 0 n'importe quel chiffre binaire. Supposons que nous

ayons un nombre et que nous voulions forcer à 0 son chiffre de droite (bit 0). On utilisera AND 254 et voici pourquoi :

PRINT 201 AND 254 ; réponse : 200

```
   11001001 <------ 201
AND 11111110 <------ 254
--------
=  11001000 <------ 200
```

Seul le dernier chiffre a été mis à 0, les autres sont restés les mêmes. 254 a en effet la particularité d'être constitué de sept chiffres 1 suivis d'un seul 0.

Si nous étions partis d'un nombre se terminant déjà par 0, AND 254 n'aurait rien modifié, ce qui nous permet de donner la conclusion suivante : quel que soit le nombre considéré, en le combinant avec 254 on pourra être assuré qu'il se terminera par 0.

Il est possible d'annuler n'importe quel chiffre d'un nombre avec l'opérateur AND. AND 124, par exemple, annulera le chiffre de gauche (bit 7) mais en même temps les deux chiffres de droite (bits 0 et 1) de n'importe quel nombre de huit chiffres.

PRINT 245 AND 124 ; réponse : 116

```
   11110101 <------ 245
AND 01111100 <------ 124
--------
=  01110100 <------ 116
```

3. Le OU exclusif logique

Noté XOR, le OU exclusif obéit aux mêmes règles que le OU déjà défini sauf pour la quatrième partie :

```
  0      0      1      1
XOR 0  XOR 1  XOR 0  XOR 1
---    ---    ---    ---
= 0    = 1    = 1    = 0
```

Le résultat n'est égal à 1 que lorsqu'un des chiffres et un seulement est égal à 1.

30 XOR 40 = 54

```
    11110 <------ 30
XOR 101000 <------ 40
--------
=  110110 <------ 54
```

```
25 XOR 100 = 125
    11001 <------ 25
XOR 1100100 <---- 100
=   1111101 <---- 125
```

L'opérateur XOR est mis en œuvre à chaque fois que l'on veut faire passer à 1 les chiffres 0 et à 0 les chiffres 1. Supposons que l'on ait un nombre dont on veuille faire changer d'état le dernier chiffre (bit 0) : on le combinera avec XOR 1. Si le nombre se terminait par 0, il se terminera alors par 1 mais par contre, si son dernier chiffre était 1, ce sera du coup 0. On essaie :

```
28 XOR 1 = 29
   11100 <------ 28
XOR 00001 <----- 1
=  11101 <------ 29

31 XOR 1 = 30
   11111 <------ 31
XOR 00001 <----- 1
=  11110 <------ 30
```

Bien entendu, XOR peut être utilisé pour faire basculer d'un état à l'autre n'importe lequel des chiffres sans modifier les autres. Par exemple, XOR 5 ne changera les états que du premier et du troisième chiffre en partant de la droite (5 est égal à 101 en binaire).

2

LA MÉMOIRE ÉCRAN D'ALICE

Partons à la découverte des ressources que possède Alice dans le domaine du graphisme et des couleurs, ressources qui sont, nous allons le voir, loin d'être toutes exploitées par le BASIC. Notez bien que tout ce qui suit concerne le mode CLS40.

GÉNÉRALITÉS

Tapez sur le clavier de votre ordinateur le programme suivant et faites-le exécuter :

```
10 POKE 48929 , 65 : REM    LETTRE A
20 POKE 48930 , 1  : REM    ALPHANUMERIQUE
30 POKE 48931 , 20 : REM    ROUGE/BLEU
40 POKE 48934 , 10 : REM    LIGNE
50 POKE 48935 , 35 : REM    COLONNE
60 POKE 48936 , 1  : REM    EXECUTION
```

Vous voyez apparaître en haut et à droite de l'écran un caractère — la lettre A majuscule — qui s'affiche en rouge sur fond bleu. Il est donc possible, nous en avons la preuve devant les yeux, de colorier un caractère alphanumérique avec des teintes différentes de celles qui sont employées habituellement (vert et noir). Nous verrons même dans un moment qu'Alice a à sa disposition une palette de couleurs dont le nombre est supérieur à 8.

Analysons les grandes lignes du programme :

10 POKE 48929 , 65

L'octet 48929 doit contenir le code ASCII de la lettre que nous souhaitons voir apparaître. Puisque 65 correspond à la lettre A, nous pouvons déjà comprendre pourquoi c'est elle et pas une autre que l'ordinateur a affichée.

20 POKE 48930 , 1

Disons, pour l'instant, que l'octet 48930 définit le type du caractère qui se verra sur l'écran ; il s'agit, pour ce qui nous concerne, d'un caractère alphabétique et non pas, par exemple, d'un élément semi-graphique.

30 POKE 48931 , 20

L'écriture du nombre 20 dans l'octet 48931 a pour effet de colorier la lettre en rouge sur fond bleu. Pour rester dans le domaine des généralités, retenons simplement que l'octet 48931 va nous servir essentiellement à choisir nos couleurs.

40 POKE 48934 , 10 et 50 POKE 48935 , 35

Il nous faut indiquer à la machine à quel endroit de l'écran doit s'effectuer l'affichage : les octets 48934 et 48935 sont là pour cela.

60 POKE 48936 , 1

Le rôle de l'octet 48936 est de commander, quand son contenu vaut 1, l'apparition de la lettre sur notre téléviseur. Il donne donc l'ordre de visualiser le caractère dont les références sont définies par les octets précédents.

AFFICHAGE ALPHANUMÉRIQUE

Reprenons, avec tous les détails cette fois-ci, l'étude des octets rencontrés dans le paragraphe précédent.

Octet 48929

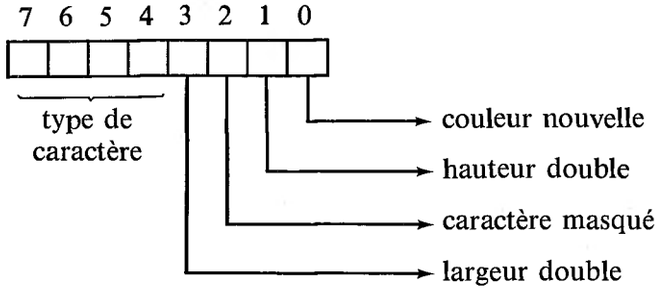
Il contient le code ASCII du caractère à afficher. Rappelons que ce code est inférieur à 128.

Exemple :

10 POKE 48929 , 90 : c'est la lettre Z qui pourra se voir à la place de la lettre A.

Octet 48930

Voici sa structure :



Les bits 7, 6, 5 et 4 sont toujours à 0 lors de l'affichage alphanumérique.

Le bit 0, s'il vaut 1, n'a aucune influence sur le reste du programme et, s'il vaut 0, va nous permettre d'obtenir 7 nouvelles couleurs dérivées des couleurs standard. Nous allons y revenir.

Le bit 2, pour sa part, n'interviendra pas dans notre étude. Son intérêt doit être considéré comme nul.

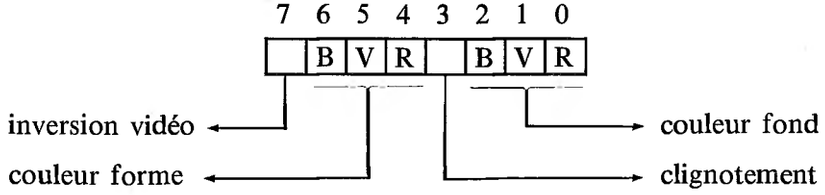
Les bits 1 et 3 autorisent, s'ils sont égaux à 1, l'impression sur écran de caractères dont la grandeur est modifiée.

Exemple :

20 POKE 48930 , 9 : la lettre s'affiche en double largeur (9 est égal à 00001001 en binaire). Notons toutefois que la partie droite du caractère est coloriée en vert et noir ; ceci s'explique par le fait que notre programme, tel qu'il est, ne concerne le coloriage en bleu et rouge que d'une seule case de l'écran et non pas de deux.

Octet 48931

Il est constitué de 8 bits dont les rôles sont définis ainsi :

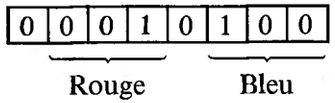


Les bits 4, 5 et 6 permettent de choisir la couleur du caractère et les bits 2, 1 et 0 la couleur du fond. Le choix de ces couleurs est déterminé par le tableau donné ici.

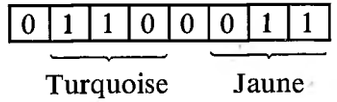
Couleurs	B	V	R
Noir	0	0	0
Rouge	0	0	1
Vert	0	1	0
Jaune	0	1	1
Bleu	1	0	0
Rose	1	0	1
Turquoise	1	1	0
Blanc	1	1	1

Exemples :

30 POKE 48931 , 20 : rouge sur bleu (20 = 00010100 binaire)



30 POKE 48931 , 99 : turquoise sur jaune (99 = 01100011 binaire)

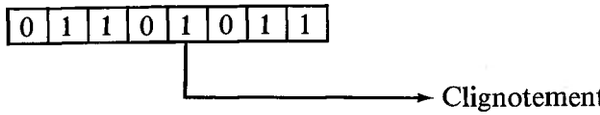


Le lecteur pourra maintenant mettre à 0 le bit de droite de l'octet 48930 et dessiner des caractères dans des couleurs nouvelles, couleurs qu'Alice n'avait pas dévoilées jusqu'alors.

Nous n'en avons pas fini avec les différents bits de l'octet 48931. Continuons avec le numéro 3 : s'il est égal à 1, le caractère va se mettre à clignoter sur l'écran. Voici quelque chose qui fait beaucoup d'effet dans un programme et, vraiment, à peu de frais.

Exemple :

30 POKE 48931 , 107 : la lettre se met à clignoter ; elle est de couleur turquoise sur jaune.



Le dernier bit enfin de cet octet miracle : celui de gauche. Il réalise l'inversion vidéo du caractère affiché.

Exemple :

30 POKE 48931 , 227 : apparition de la lettre en jaune sur fond turquoise.

Octet 48934

C'est dans cet octet que se trouve l'ordonnée de la case affichée.

Exemple :

40 POKE 48934 , 0 : la lettre apparaît sur la première ligne de l'écran (ligne 0).
40 POKE 48934 , 8 : la lettre apparaît sur la deuxième ligne (ligne 1). Oui, sur la ligne 1 et non pas sur la ligne 8. Il faut savoir en effet que le numérotage des lignes n'utilise pas les chiffres allant de 1 à 7 : ils sont en quelque sorte sautés.
40 POKE 48934 , 9 : la lettre apparaît sur la troisième ligne (ligne 2).
40 POKE 48934 , 31 : la lettre apparaît sur la dernière ligne (ligne 24).

Octet 48935

Nous savons déterminer le numéro de la ligne d'affichage, voici maintenant comment on choisit la colonne : on écrit dans l'octet 48935 l'abscisse de la case, et d'une façon tout à fait logique (aucun chiffre n'est *oublié*).

Exemples :

```
50 POKE 48935 , 0
50 POKE 48935 , 39
50 POKE 48935 , 7
```


Le caractère s'affiche respectivement dans la première, la dernière et la huitième colonne (colonne numéro 7).

Octet 48936

N'ajoutons rien à ce qui a déjà été dit : cet octet donne à l'ordinateur l'ordre de dessiner sur l'écran le caractère que l'on vient de définir.

PROGRAMME DE DÉMONSTRATION

Mettons en application nos connaissances toutes neuves et voyons-en de toutes les couleurs.

Programme BASIC

```
 10 CLS : POKE 48930 , 1 : POKE 48931 , 0
 20 FOR Y = 0 TO 31 : FOR X = 0 TO 39
 30 POKE 48934 , Y : POKE 48935 , X
 40 READ I : POKE 48929 , I : POKE 48936 , 1
 50 IF I < >42 THEN 70
 60 RESTORE : POKE 48931 , PEEK(48931) + 1
 70 NEXT X : IF Y = 0 THEN Y = 7
 80 IF Y = 17 THEN POKE 48930 , 0
 90 NEXT Y : DATA 32 , 65 , 76 , 73 , 67 , 69 , 32 , 42
100 GOTO 100
```

Commentaires

Au démarrage du programme, les nombres X et Y sont nuls tous les deux. Puisqu'ils sont écrits dans les octets 48934 et 48935, le premier caractère est affiché en haut et à gauche de l'écran. Ce caractère (ASCII 32) est une case vide dont le code est écrit dans l'octet 48929. Sa couleur nous est donnée par la valeur de l'octet 48931 ; comme cette valeur est nulle pour l'instant, le premier caractère apparaît en noir sur noir.

A la deuxième exécution de la boucle FOR NEXT X, les octets 48934 et 48935 définissent la case située immédiatement à droite de la précédente. Dans cette case est écrite la lettre A (ASCII 65) et, puisque nous ne sommes pas intervenus sur l'octet 48931, toujours en noir sur noir.

Au huitième passage de la boucle, nous pouvons voir en haut de

l'écran huit cases disposées les unes à côté des autres. Elles sont toutes entièrement coloriées en noir.

A ce moment-là, une unité est ajoutée à l'octet 48931 et l'instruction RESTORE oblige l'ordinateur à relire les huit codes ASCII écrits en DATA. Les lettres correspondantes sont alors affichées sur le téléviseur et, cette fois-ci, elles sont bien visibles car de couleur noire sur fond rouge. On aura reconnu dans les nombres écrits en DATA les équivalents des caractères espace, A , L , I , C , E , espace et * ; c'est donc cette chaîne qui, naturellement, est apparue.

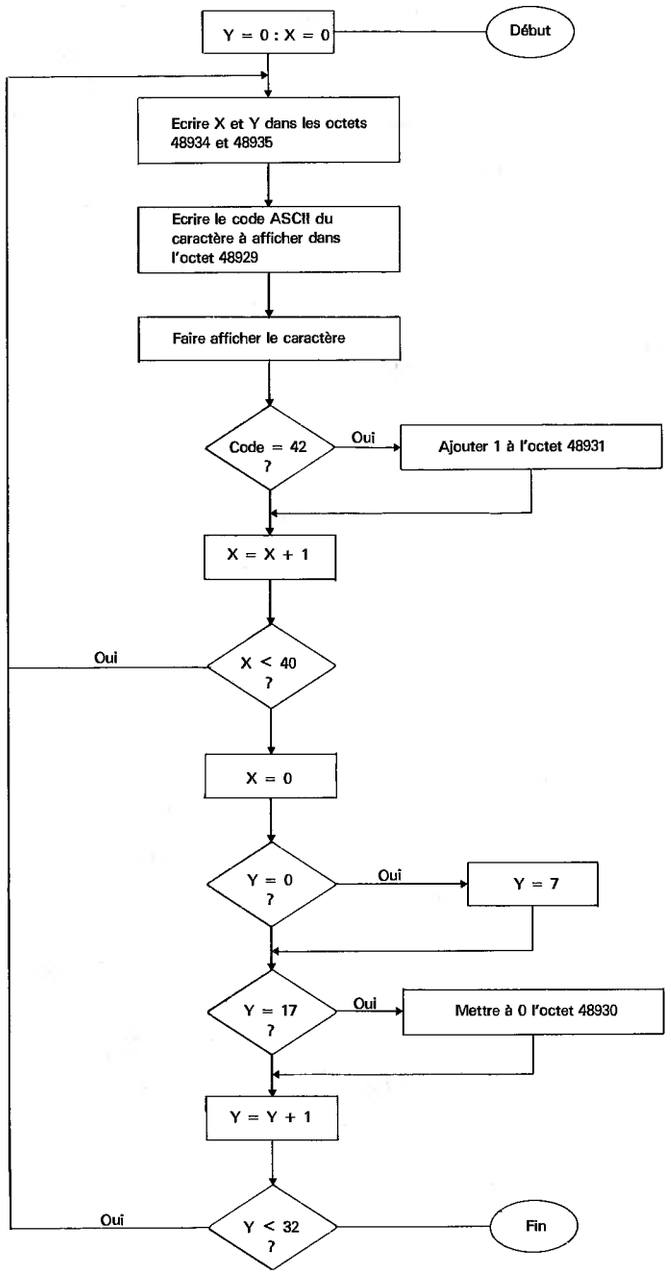
Faisons les comptes : quand le programme arrive à son terme, le mot ALICE a été affiché 125 fois. A chaque fois le contenu de l'octet 48931 est modifié (augmenté de 1). De ce fait, les couleurs changent constamment et certaines d'entre elles se mettent à clignoter (quand le bit 3 de notre octet est à 1). Remarquons que nous nous servons à partir d'un certain moment des nouvelles couleurs (lignes BASIC 80). Nous avons devant les yeux une idée de ce que notre ordinateur est capable de réaliser dans le domaine des couleurs. Etonnant, non ?

Programme assembleur

Tapez au clavier le programme suivant et faites-le exécuter :

```
10 CLEAR 100 , 18000
20 FOR I = 18944 TO 19025 : READ J
30 POKE I , J : NEXT : EXEC 18944
40 GOTO 40
50 DATA183,191,34,127,191,35,206,74,74,204,0,0,247,191,38,55,
   183,191,39
60 DATA54,166,0,183,191,33,198,1,247,191,40,246,191,32,193,128,
   36,249,8
70 DATA128,42,38,6,124,191,35,206,74,74,50,76,129,40,38,218,79,
   51,93,38,2
80 DATA198,7,193,17,38,3,127,191,34,92,193,32,38,195,57,32,65,
   76,73,67
90 DATA69,32,42
```

Inutile de faire un effort pour essayer de comprendre ce programme. Laissez de côté la fin de ce paragraphe et passez à la suite. Lorsque vous y reviendrez dans quelques jours (quelques heures ?), il vous paraîtra limpide. Retenez seulement que le programme assembleur a réalisé le même travail que le programme BASIC, l'affichage du mot



ALICE 125 fois de suite, mais avec une rapidité époustouflante : tout l'écran s'allume d'un seul coup !

```
1                ORG     $4A00
2                EXC     DEBUT
3   DEBUT        LDAA    #$1
4                STAA    $BF22     ; 48930 DECIMAL
5                CLR     $BF23     ; 48931 DECIMAL
6                LDX     #ALICE
7                LDD     #$0       ; A ET B A ZERO
8   LIGNE        STAB    $BF26     ; 48934 ORDONNEE
9                PSHB
10  COL.         STAA    $BF27     ; 48935 ABSCISSE
11               PSHA
12               LDAA    $0,X
13               STAA    $BF21     ; 48929 CARACTERE
14               LDAB    #$1
15               STAB    $BF28     ; 48936 EXECUTION
16  TEMPO        LDAB    $BF20     ; ATTENTE
17               CMPB    #$80
18               BHS     TEMPO
19               INX
20               CMPA    #$2A      ; 42 ASCII *
21               BNE     ETIQ1
22               INC     $BF23     ; 48931 COULEUR
23               LDX     #ALICE
24  ETIQ1        PULA
25               INCA
26               CMPA    #$28      ; 40 FIN DE LIGNE
27               BNE     COL.
28               CLRA
29               PULB
30               TSTB
31               BNE     ETIQ2
32               LDAB    #$7       ; NUMEROS SAUTES
33  ETIQ2        CMPB    #$11      ; 17 DECIMAL
34               BNE     ETIQ3
35               CLR     $BF22     ; 48930 NVELLES COUL.
36  ETIQ3        INCB
37               CMPB    #$20      ; 32 DERNIERE LIGNE
38               BNE     LIGNE
39               RTS
40  ALICE        ' ALICE *
```


Commentaires

Lignes 3, 4, 5 et 6 : c'est l'initialisation du programme. On met à 1 l'octet 48930 (l'affichage va concerner des caractères alphanumériques). On annule l'octet 48931 (la première lettre sera masquée : couleur noire sur fond noir). Puis on place dans le registre X l'adresse du premier caractère de la chaîne " ALICE *".

Lignes 8 à 11 : on précise à l'ordinateur à quel endroit de l'écran doit apparaître la première lettre.

Lignes 12 et 13 : on écrit dans l'octet 48929 le code ASCII du premier caractère. Il s'agit donc pour l'instant du nombre 32 (code espace).

Lignes 14 et 15 : l'exécution de l'affichage est commandée par l'écriture de la valeur 1 dans l'octet 48936. A ce moment-là, le premier caractère est visible en haut et à gauche de l'écran.

Lignes 16 à 18 : voici la seule chose qui change par rapport au programme BASIC. Etant donné que l'affichage d'un symbole sur l'écran prend un certain temps, on est obligé de ralentir la course du microprocesseur. Expliquons-nous : si l'on supprime ces trois lignes, le programme risque, au passage suivant dans la boucle COL., de donner à l'octet 48936 l'ordre de faire apparaître le deuxième caractère sur le téléviseur alors que l'affichage du premier n'a pas été terminé. Du coup cet ordre ne peut être pris en compte et notre programme ne fait pas exactement ce que l'on attend de lui. Nous engageons le lecteur à regarder l'effet produit par la suppression des lignes 16 à 18. Reste à savoir maintenant comment on empêche le 6803 de *marcher plus vite que la musique*. On lui fait attendre dans la boucle TEMPO que le bit 7 de l'octet 48928 ($BF20) passe à 0. Dès que cela arrivera, le contenu de l'octet en question sera inférieur à 128 ($80) et le programme pourra reprendre son exécution normale. Retenons de ceci que le bit gauche de l'octet 48928 est toujours à 1 quand une commande d'affichage est lancée et qu'il est abaissé à 0 dès que cet affichage est entièrement réalisé.

Lignes 19 à 23 : le registre X est incrémenté et pointe alors sur le deuxième caractère de la chaîne, c'est-à-dire sur la lettre A. On vérifie alors que la fin de cette chaîne (caractère *) n'est pas atteinte. Si tel était le cas, donc si le mot ALICE * était affiché entièrement sur

l'écran, on changerait la couleur en augmentant d'une unité la valeur de l'octet 48931.

Lignes 24 à 27 : le programme tourne dans la boucle COL. tant que la première ligne n'est pas complètement dessinée sur le téléviseur.

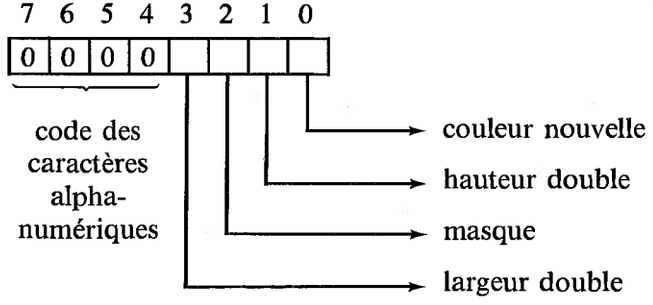
Lignes 28 et suivantes : quand une ligne est affichée, on remet à zéro, par l'intermédiaire du registre A, l'octet 48935 et on incrémente l'octet 48934. Comme ce dernier contient les numéros de ligne, on prend soin de lui faire sauter les nombres 1, 2, ... 7. D'autre part, quand sa valeur atteint 17, on intervient sur l'octet 48930 de manière que l'affichage se poursuive dans les nouvelles teintes.

DIFFÉRENTS JEUX DE CARACTÈRES

Repartons du programme rencontré au début de ce chapitre.

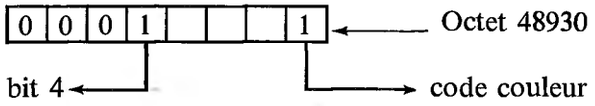
```
10 POKE 48929 , 65 : REM    LETTRE A
20 POKE 48930 , 1  : REM    ALPHANUMERIQUE
30 POKE 48931 , 20 : REM    ROUGE/BLEU
40 POKE 48934 , 10 : REM    LIGNE
50 POKE 48935 , 35 : REM    COLONNE
60 POKE 48936 , 1  : REM    EXECUTION
```

Il a été dit que la structure de l'octet 48930 était la suivante :



Notre préoccupation, jusqu'à maintenant, s'est limitée à l'affichage des caractères alphanumériques standard. Mais Alice a beaucoup d'autres ressources et ce paragraphe nous emmène découvrir lesquelles.

Affichage alphanumérique souligné



Remplaçons la ligne 20 de notre programme par :

```
20 POKE 48930 , 17
```

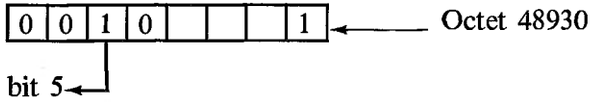
puis faisons RUN. La lettre majuscule A apparaît en rouge sur fond bleu à l'intersection de la colonne n° 35 et de la ligne n° 3. Ce qui a changé par rapport à notre étude précédente, c'est qu'un trait fin est tracé sous le caractère : la lettre est maintenant soulignée.

Un autre exemple :

```
10 POKE 48929 , 101
```

Puisque 101 est le code ASCII de la lettre minuscule *e*, c'est cette lettre, soulignée, qui se dessine sur l'écran.

Affichage de 128 caractères mosaïques



Remplacez la ligne 20 par :

```
20 POKE 48930 , 33
```

puis faites plusieurs exécutions en changeant les nombres écrits dans l'octet 48929. Vous verrez apparaître des caractères semi-graphiques ayant des formes de mosaïques.

Exemples :

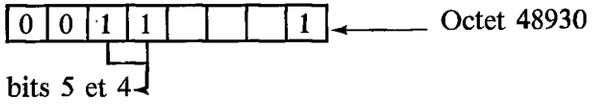
```
10 POKE 48929 , 123
```

Apparition d'un graphisme ressemblant à la lettre C à l'envers.

```
10 POKE 48929 , 42
```

Dessin de trois petits points les uns en dessous des autres. Le lecteur intéressé pourra écrire tous les nombres inférieurs à 128 dans l'octet 48929 et dresser une table de tous les symboles semi-graphiques dont on peut disposer.

Affichage de 32 caractères mosaïques complémentaires



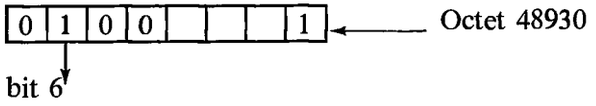
Si les 128 graphismes précédents ne vous suffisent pas, vous aurez la possibilité d'en utiliser 32 autres (numérotés de 0 à 31).

Exemple :

```
10 POKE 48929 , 9
20 POKE 48930 , 49
```

Quelque chose qui ressemble à un V à l'envers apparaît sur le téléviseur.

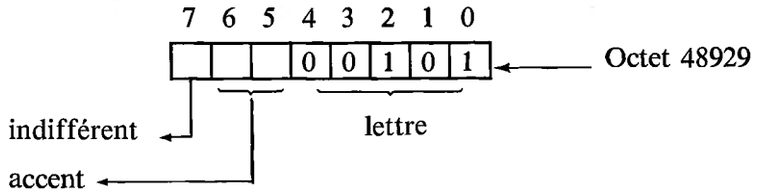
Affichage des minuscules avec accent



Avant d'essayer de nous y retrouver, modifions la ligne 20 :

```
20 POKE 48930 , 65
```

Reste à savoir comment est codé l'octet 48929 :



Les cinq bits de droite permettent de déterminer la lettre minuscule sur laquelle on va tracer un accent : *a* est la première, et *z* est la vingt-sixième. Pour notre exemple, c'est la lettre *e*, la cinquième de l'alphabet, donc.

Les bits 6 et 5 servent à choisir l'accent :

bit 6 — 0 0 — bit 5 Pas d'accent.

Exemple :

```
10 POKE 48929 , 5
```

La lettre *e* apparaît sans aucun ajout.

bit 6 — 0 1 — bit 5 Surlignage.

Exemple :

```
10 POKE 48929 , 37
```

La même lettre est surmontée d'un petit trait.

bit 6 — 1 0 — bit 5 Accent aigu.

Exemple :

```
10 POKE 48929 , 69
```

La lettre est cette fois visible avec un accent aigu.

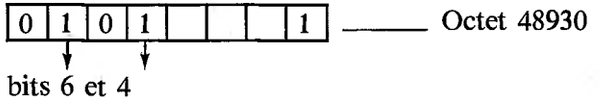
bit 6 — 1 1 — bit 5 Accent grave.

Exemple :

```
10 POKE 48929 , 101
```

Et voici la lettre *è*.

Affichage des minuscules accentuées soulignées



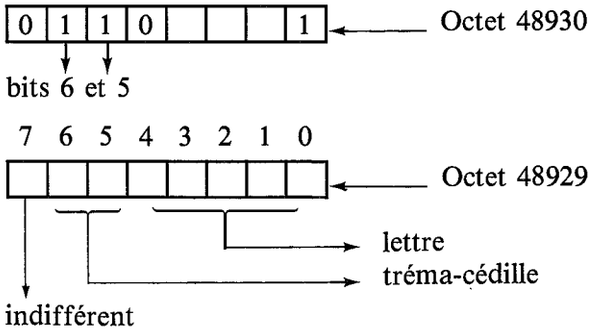
Nous avons ici accès aux mêmes caractères que précédemment mais, de plus, un trait fin les souligne.

Exemple :

```
10 POKE 48929 , 101
20 POKE 48930 , 81
```

Le programme affiche la lettre *è* soulignée.

Affichage des minuscules avec tréma ou cédille



Ici encore les bits 6 et 5 de l'octet 48929 servent à déterminer ce qui sera ajouté à la lettre.

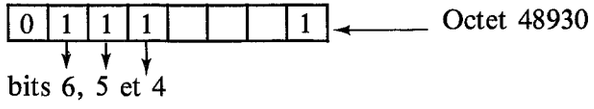
bit 6	0 0	bit 5	Aucun ajout.
bit 6	0 1	bit 5	Lettre surmontée d'un point.
bit 6	1 0	bit 5	Lettre surmontée d'un tréma.
bit 6	1 1	bit 5	Lettre avec cédille.

Exemple :

```
10 POKE 48929 , 99
20 POKE 48930 , 97
```

Nous avons devant les yeux le caractère *ç*.

Affichage souligné des minuscules avec tréma ou cédille



Exemple :

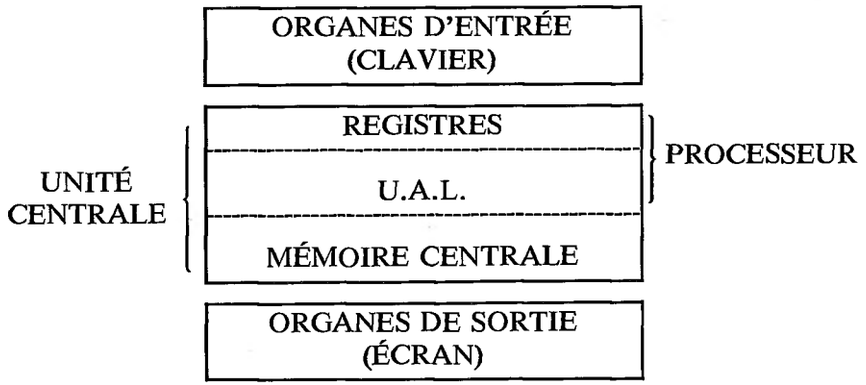
```
10 POKE 48929 , 73
20 POKE 48930 , 113
```

L'exécution du programme entraîne le dessin de la lettre *ï* soulignée.

3

L'ARCHITECTURE INTERNE DU MICROPROCESSEUR 6803

Avant de commencer l'étude du microprocesseur, il est nécessaire de le situer dans son environnement : aussi ce chapitre débutera par quelques rappels sur la structure générale d'un micro-ordinateur.



C'est essentiellement la mémoire centrale qui sera utile à notre étude et c'est pour cette raison que les mémoires auxiliaires n'apparaissent pas sur le schéma.

LA MÉMOIRE CENTRALE

La mémoire centrale permet d'enregistrer, de conserver, et de restituer à la demande les informations qui lui ont été communiquées. Ces informations sont de deux sortes : ce sont soit des données soit des programmes. A priori, rien ne distingue en mémoire ces deux types d'informations et c'est la seule logique du programme qui empêchera la confusion.

Pour des raisons technologiques, l'information rangée en mémoire se trouve sous la forme de combinaisons des chiffres binaires 0 et 1. Le bit (ou chiffre binaire) est l'unité élémentaire d'information et ne peut prendre que l'une de ces valeurs. Un octet est constitué de huit bits. Le nombre le plus grand que l'on puisse avoir dans un octet est le nombre 11111111 (soit 255 en décimal). Le nombre le plus petit est obtenu quand les huit bits valent 0 : c'est le nombre 00000000 (ou 0 en décimal).

La mémoire centrale de la plupart des micro-ordinateurs est divisée en 65536 octets et l'ordinateur est capable de retrouver le contenu de

n'importe quel octet grâce à son adresse. Les octets de la mémoire sont numérotés de 0 à 65535 et c'est ce numéro que l'on appelle l'adresse. Il est possible de savoir le nombre que contient un octet avec la fonction BASIC PEEK. Essayons :

```
PRINT PEEK (50390) ; réponse : 155
```

Puisque 155 = 10011011 en binaire, on peut retrouver la configuration exacte de l'octet numéro 50390 :

1	0	0	1	1	0	1	1
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

On peut dire aussi que cet octet contient le nombre hexadécimal 9B.

Puisqu'on est capable de connaître la valeur qui se trouve dans un octet, on doit être capable de modifier cette valeur : l'instruction POKE est à notre disposition pour cela.

```
PRINT PEEK (20000) ; réponse : 0
POKE 20000, 100
PRINT PEEK (20000) ; réponse : 100
```

Avant notre intervention, l'octet numéro 20000 contenait le nombre 0. Tous les octets contiennent une valeur et il est difficile de dire, dans l'absolu, à quoi elle correspond. Nous avons inscrit par POKE la valeur 100 dans l'octet et il ne nous est plus resté qu'à en demander la confirmation avec PRINT PEEK (20000).

Un nouvel essai :

```
PRINT PEEK (50000) ; réponse : 39
POKE 50000, 100
PRINT PEEK (50000) ; réponse : 39
```

Eh oui, la commande POKE ne nous permet pas de modifier le contenu de n'importe quel octet de la mémoire ! Nous allons savoir pourquoi dès que nous aurons distingué les deux grands types de mémoire.

La ROM (*Read Only Memory*) est la partie de la mémoire centrale que l'on peut seulement lire. Il n'est pas question de modifier un octet qui se trouve dans cette zone-là. L'octet 50000 par exemple se trouve

en ROM et c'est pourquoi POKE n'a eu aucun effet sur lui. Pour ce qui concerne l'ordinateur qui est étudié dans ce livre, Alice, il faut savoir que nous ne pourrons en aucun cas modifier un octet dont l'adresse est supérieure à 49152. C'est en effet la partie de la mémoire qui contient l'interpréteur, c'est-à-dire le programme qui va traduire le BASIC que l'on tape au clavier en un langage compréhensible par la machine. Le contenu de la ROM a l'avantage de ne pas s'effacer quand on coupe le courant : c'est une mémoire permanente.

La RAM (*Random Access Memory*) ou mémoire vive, contient tous les octets que l'on peut lire — avec PEEK — et aussi modifier — avec POKE. L'octet 20000, dont le contenu a pu être modifié par nous, se trouve dans la zone vive de la mémoire. C'est en RAM que le programme BASIC que nous tapons est stocké, et nous savons que dès la mise hors tension, l'ordinateur *oublie tout*. Ceci met en avant une propriété de la mémoire vive : elle est volatile, son contenu est perdu dès que l'unité centrale n'est plus alimentée.

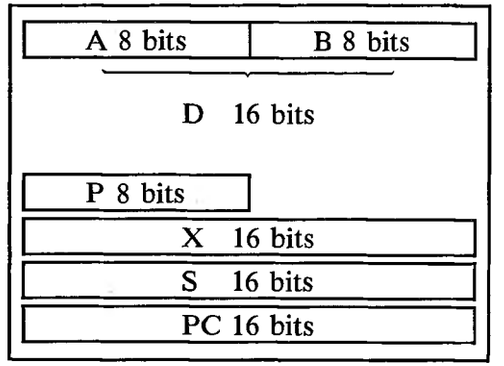
Nous nous servirons tout au long de ce livre des mots PEEK et POKE. Ces instructions réservent parfois quelques surprises, mais elles sont considérées comme indispensables par tous les esprits curieux qui veulent aller un peu plus loin que le BASIC.

Attention toutefois à ne pas chercher à inscrire dans un octet un nombre supérieur à 255 ; cela aurait pour effet de faire afficher le message FC ERROR. La raison en est qu'avec huit bits, on ne peut constituer un nombre supérieur à 11111111 soit 255 en décimal.

L'UNITÉ ARITHMÉTIQUE ET LOGIQUE (U.A.L.)

L'UAL est constituée de circuits électroniques câblés et est capable d'effectuer des calculs arithmétiques (addition, soustraction et multiplication) et des choix logiques (comparaison de deux nombres). Elle permet aussi de faire des opérations de décalage et de rotation, opérations auxquelles quelques pages de ce livre seront consacrées plus loin.

En écrivant nos programmes, nous n'aurons pas à intervenir directement sur l'UAL mais, même s'il n'en est plus du tout question explicitement, il ne faudra pas oublier le rôle capital de cette unité dans un ordinateur : l'UAL est le calculateur du microprocesseur.



Les registres sont identifiables à des cases mémoire par lesquelles l'information va transiter. Le microprocesseur 6803 contient des registres 8 bits et des registres 16 bits. Les instructions BASIC ne permettent pas l'accès à ces registres et nous ne pourrons jamais, avec la fonction PEEK par exemple, savoir ce que contient tel ou tel registre.

1. Les registres A et B

Ce sont les plus utilisés en assembleur et on leur donne souvent le nom d'accumulateurs. Nous serons par exemple obligés de passer par l'un d'eux dès qu'il s'agira de faire un calcul. Ils sont constitués tous deux de 8 bits, ce qui fait que le plus grand nombre qu'ils peuvent contenir est 255 décimal (FF hexa ou 11111111 binaire).

Une particularité très importante de A et B est qu'ils peuvent être utilisés ensemble pour constituer le registre D, appelé lui aussi accumulateur. Naturellement, cela fera 16 bits pour D.

Voyons ceci de plus près : supposons que les bits des registres A et B soient disposés comme suit :

	128	64	32	16	8	4	2	1
A	0	1	0	1	0	0	0	1

Dans A se trouve donc le nombre décimal 81 ou hexadécimal 51.

	128	64	32	16	8	4	2	1
B	1	0	0	0	1	1	0	0

Dans B c'est le nombre 140 (décimal) ou 8C (hexa) qui se trouve.

Pour former D, on associe A et B :

A								B							
0	1	0	1	0	0	0	1	1	0	0	0	1	1	0	0

La valeur de D est alors 0101000110001100 c'est-à-dire 20876 en décimal. Il faudra toujours se souvenir que D est un registre 16 bits et qu'il correspond donc à deux octets (soit un nombre compris entre 0 et 65535).

2. Le registre X

C'est un registre 16 bits et donc on peut y écrire n'importe quel nombre de 0 à 1111111111111111 (16 fois le chiffre 1). Si nous prenons la peine de traduire cette valeur binaire en décimal, on obtient 65535. Ceci nous permet de comprendre le rôle que jouera X : il contiendra généralement une adresse mémoire et cette adresse pourra être celle de n'importe quel octet de l'intervalle 0 – 65535.

X est souvent appelé registre d'adresses eu égard à ce que nous venons de dire, mais on l'appelle aussi registre d'index car on peut l'utiliser dans le mode d'adressage indexé (nous verrons cela bientôt).

3. Le registre SP ou S

Il porte le nom de pointeur de pile. Une pile est un endroit de la mémoire où seront stockés — empilés — des nombres les uns à la suite des autres. La structure de la pile d'un ordinateur correspond tout à fait à celle d'une pile d'assiettes : on peut toujours rajouter une assiette sur la pile, mais si l'on veut en reprendre une, ce sera toujours la dernière posée que l'on devra récupérer (dernier entré, premier sorti).

L'utilisation de la pile n'étant pas évidente pour le programmeur qui fait ses premiers pas en assembleur, passons un peu de temps sur un exemple. Supposons que dans le processeur, les registres A, B, X et S aient les valeurs suivantes :

A	0	0	0	1	0	1	0	0

A contient donc 20 (décimal) ou 14 (hexadécimal)

B	0	0	1	0	0	0	1	1

B contient donc 35 (décimal) ou 23 (hexadécimal)

X	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	1

Dans le registre X est écrit le nombre décimal 3969 (0F81 hexadécimal)

S	0	1	0	0	0	1	0	1	1	1	1	0	1	1	0	0

Dans S se trouve le nombre 17900 (45EC en hexadécimal).

La valeur 17900 indique dans cet exemple que nous allons empiler nos nombres dans une série d'octets de la mémoire à partir justement de l'octet 17900. Cet octet en lui-même ne sera pas concerné par notre travail car c'est dans l'octet juste à côté que va démarrer notre empilement.

PSHA

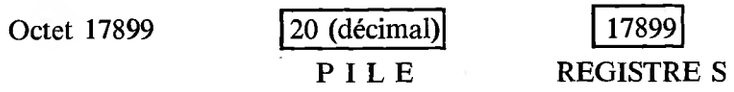
Nous voilà devant notre première instruction assembleur. Elle utilise une abréviation du mot anglais PUSH (pousser). La lettre A précise que c'est le contenu de l'accumulateur A qui va être placé sur la pile. Lorsque cette instruction aura été exécutée par le microprocesseur, voici ce qui se sera passé :

Le registre A n'aura subi aucune modification. Son contenu, le nombre 20, sera allé s'écrire dans la pile mais ce registre en aura conservé la trace. Il en sera toujours ainsi quand nous donnerons l'ordre au processeur de transférer un nombre d'un registre vers une case mémoire : le registre ne sera pas modifié, et tout se passera comme si le registre n'avait envoyé qu'un double, un duplicata à la mémoire.

Les registres B et X ne sont pas intervenus dans l'instruction et gardent donc la même valeur.

Le registre S est en fin de compte le seul qui sera modifié. Il va passer à 17899 indiquant de ce fait que la pile se trouve maintenant à cette adresse. Voici donc défini le rôle de S : il contient un nombre de 16 bits et ce nombre est l'adresse de l'octet où se trouve la pile.

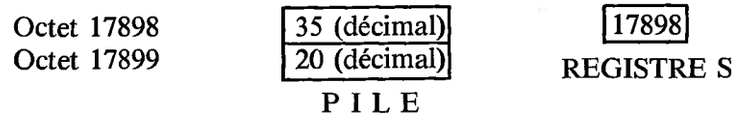
Pour l'instant, notre pile n'est formée que d'un seul octet (à l'adresse 17899) et ne contient qu'un seul nombre (20).



Continuons avec **PSHB**

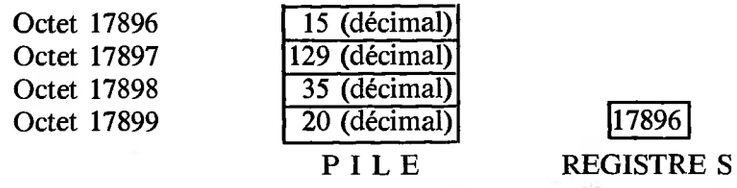
Cette fois, c'est le contenu du registre B qui va aller se placer sur la pile — c'est-à-dire dans l'octet 17898. Si nous avions la possibilité de mélanger à loisir l'assembleur et le BASIC, nous taperions au clavier PRINT PEEK (17898) : la réponse serait 35.

Naturellement, le registre S est encore modifié, n'oublions pas que c'est lui qui tient les comptes de la pile.



Et si on essayait **PSHX** ?

Il va falloir que le microprocesseur aille ranger dans la pile la valeur de X. Or X est un registre 16 bits et il ne peut être question d'en placer le contenu sur un seul octet. Par contre, en nous servant des deux octets 17896 et 17897, nous aurons la réponse à notre problème : les huit bits de gauche de X (bits de poids fort) seront recopiés dans l'octet 17896 et les huit derniers bits (bits de poids faible) seront inscrits dans l'octet 17897.

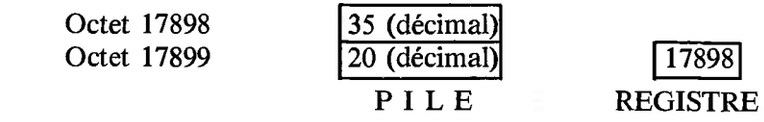


15 est la traduction décimale de la partie gauche de X et 129 est la traduction, toujours décimale, de sa partie droite.

Avant de passer à l'opération de dépilement, notons bien que les nombres sont stockés dans la pile sur des octets dont les adresses décroissent à chaque fois. Il n'y a rien à y faire, c'est dans la logique sans doute de notre ordinateur : lui, il empile par en dessous. Le tout est de le savoir.

PULX

C'est notre deuxième instruction assembleur : elle effectue exactement le travail inverse de la première. Elle va rechercher un nombre dans la pile et elle l'écrit dans le registre X. Elle a donc dépilé les octets 17896 et 17897 et a été les replacer dans X. Par là même, notre pile ne contient plus que deux octets et le registre S repasse à 17898.

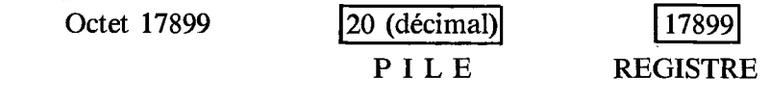


L'intérêt des instructions PSH et PUL n'apparaît pas immédiatement aux programmeurs qui débutent avec l'assembleur car ils ne voient pas à quoi peuvent bien servir deux choses qui ne font rien d'autre que nous ramener à notre point de départ. Et pourtant c'est là justement qu'en réside tout l'intérêt : la principale difficulté de l'assembleur vient de ce que nous ne disposons que d'un nombre très réduit de registres. Nous verrons très vite le très gros avantage qu'il y a à placer la valeur d'un registre dans la pile (PUSH), à utiliser ce registre pour faire autre chose, et à retrouver (PULL) la valeur initiale de ce même registre.

Finissons-en avec notre exemple :

PULA

La machine va aller écrire 35 dans l'accumulateur. Elle exécute très exactement l'ordre qu'on lui donne en prenant le nombre écrit au sommet de la pile — donc 35 — et en le plaçant dans A. A ce moment-là les deux registres A et B contiennent précisément la même valeur, 35.



PULB

La pile que nous avions constituée est réduite à zéro. Le registre B va prendre la valeur 20 et le registre S reprendra sa valeur initiale 17900.

En fin de compte, dans notre exercice, les accumulateurs auront été

échangés et les registres X et S auront retrouvé les contenus qu'ils avaient au départ.

Un autre exemple maintenant. On repart du même énoncé :

A contient le nombre décimal 20
B contient le nombre décimal 35
X contient le nombre décimal 3969
S contient le nombre décimal 17900.

Puis, on exécute les instructions suivantes :

```
PSHA  —  PSHB  —  PSHX
PULA  —  PULB  —  PULX
```

Nous laisserons au lecteur le soin de déterminer ce que les registres A, B et X contiendront à la fin de ces six opérations.

(Réponses : A (15) ; B (129) ; X (8980))

Il faudra se souvenir que la pile S est utilisée aussi par le microprocesseur pour tenir des comptes internes. Les ennuis nous seront garantis si, par mégarde, nous ne remettons pas la pile système dans l'état où nous l'avons trouvée. Si on a eu besoin par exemple d'empiler trois octets dans S, il faudra être sûr qu'à la fin de notre programme les trois octets en question ont été dépilés.

4. Le registre PC

C'est un registre 16 bits que l'on appelle le compteur de programme ou le compteur ordinal. Le nombre qui est écrit dans ce registre est l'adresse de la prochaine instruction à exécuter. Il permet au microprocesseur de toujours savoir à quel octet du programme il va devoir s'intéresser. On ne l'utilise en programmation que dans des cas bien particuliers : il n'est pas directement accessible.

5. Le registre P

Ce registre 8 bits est utilisé pour faire ressortir certaines conditions particulières qui sont apparues dans le déroulement d'un programme. Ses bits prennent des valeurs différentes suivant les instructions utilisées. On ne se servira (directement) qu'assez peu de ce registre dans ce livre : sachez toutefois qu'il permettra de tester, de façon souvent transparente pour nous, si un résultat est négatif, positif ou nul ou si une retenue est apparue dans une opération par exemple.

LES MODES D'ADRESSAGE

Un mode d'adressage est un moyen qui permet au microprocesseur d'avoir accès à une donnée. Cette donnée peut être un nombre quelconque dont on aura besoin dans le programme, un nombre qui se trouve déjà dans un registre, ou encore un nombre qui se trouve écrit quelque part en mémoire.

La connaissance des principaux modes d'adressage est obligatoire : elle permet d'écrire les programmes de la façon la plus courte, la plus simple et la plus lisible possible.

1. L'adressage inhérent

L'adressage inhérent est habituellement réservé aux instructions qui agissent directement sur les valeurs contenues par les registres. Ces instructions se comprennent d'elles-mêmes et n'ont aucunement besoin qu'on leur ajoute des indications.

Exemple : INCA

Tous les microprocesseurs comprennent ce genre d'instruction : elle signifie que le registre A se verra incrémenté, c'est-à-dire que la valeur qu'il contenait se retrouvera augmentée d'une unité.

A contenait 35 (par exemple)

```
INCA
```

A contient 36.

2. L'adressage immédiat

Dans ce mode, une valeur apparaît après l'instruction assembleur. Prenons par exemple : LDAA #$5

La formule LDAA, qui sera retrouvée tout au long de ce livre, signifie que l'on va placer (charger) un nombre dans le registre A. Il est facile de voir qu'ici l'instruction LDAA n'aurait pas pu être écrite toute seule, comme dans l'adressage inhérent. Il nous faut absolument rajouter des indications à la suite : et, si l'on doit mettre un nombre dans l'accumulateur A, il faut bien dire lequel.

Dans le mode d'adressage immédiat, c'est la valeur marquée après l'instruction (ici 5) qui sera écrite dans A.

A contenait par exemple 50

```
LDAA #$5
```

A contient alors 5.
Le signe # est réservé à l'adressage immédiat et permet de ne pas avoir, après chaque instruction, à écrire sous quel mode elle doit être comprise. Quant au signe $, il est là pour indiquer que la valeur qui suit est un nombre hexadécimal. C'est bien ennuyeux mais c'est comme ça : tous nos programmes assembleur utiliseront l'hexadécimal.
Un contre-exemple :

```
LDAA #$300
```

Cette ligne devrait, en principe, écrire dans A le nombre hexadécimal 300 (768 décimal). Vous l'aviez deviné ; ceci n'a aucun sens puisque A est un registre 8 bits et que le nombre maximum qu'il peut contenir est 255.

3. L'adressage étendu

On l'appelle souvent adressage absolu car c'est un mode qui va concerner le contenu de n'importe quel octet de la mémoire.

```
LDAA $80
```

L'accumulateur sera chargé non par le nombre 128 (ou 80 hexa) comme il l'aurait été dans l'adressage immédiat, mais avec la valeur écrite dans l'octet numéro 128.

```
PRINT PEEK (128) ; réponse : 34
```

En fin de compte, le registre A contiendra 34. Aucun signe n'est utilisé quand l'instruction est écrite avec le mode étendu.
Un deuxième exemple :

```
LDAA $FFFF
```

Un piège, comme dans le paragraphe précédent ? 65535 (ou FFFF hexa) paraît bien trop important pour notre accumulateur 8 bits. Mais

non, cette ligne n'est pas un non-sens : elle signifie que le registre A va contenir non pas le nombre 65535, mais le nombre qui se trouve écrit dans l'octet ayant 65535 pour adresse.

```
PRINT PEEK (65535) ; réponse : 46
```

Ce qui fait que A sera chargé, une fois l'instruction assembleur exécutée, par la valeur 46.

4. L'adressage indexé

C'est un mode qui concerne le contenu du registre X.
Un peu délicat à utiliser au départ, il s'avère ensuite offrir de multiples possibilités au programmeur.
Voici quelques exemples :

```
LDAA $0, X
```

L'accumulateur va être chargé avec la valeur qui se trouve dans l'octet ayant pour adresse le nombre écrit dans X.
Supposons que dans le registre 16 bits X il y ait le nombre 50000.

```
PRINT PEEK (50000) ; réponse : 39.
```

Après LDAA $0, X il y aura dans le registre A la valeur 39. Cette instruction est donc tout à fait équivalente à LDAA $C350 (adressage étendu avec C350 = 50000 décimal).
Deuxième exemple :

```
LDAA $3, X
```

Cette fois, le processeur va aller chercher, pour l'écrire dans A, la valeur qui se trouve en mémoire dans l'octet ayant pour adresse le nombre contenu par X auquel on ajoute 3.
Admettons ici aussi que X contienne la valeur 50000.
On ajoute 3 à 50000 et l'octet concerné est alors le numéro 50003.

```
PRINT PEEK (50003) ; réponse : 11.
```

Ainsi, dans cet exemple, A va être chargé avec le nombre 11.

5. L'adressage direct

Il concerne, tout comme l'adressage étendu, le contenu d'un octet de la mémoire mais il ne peut être utilisé que s'il s'agit d'un octet dont l'adresse est inférieure à 256. Un exemple :

```
LDAA <$80
```

Après exécution de cette instruction, il y aura dans le registre A le même nombre que celui qui est écrit à l'adresse $80 (128 décimal).

```
PRINT PEEK (128) ; réponse : 34.
```

C'est donc la valeur 34 qui a été placée dans A.

Il n'y a rien d'autre à ajouter sur ce mode d'adressage que d'ailleurs l'on n'emploie qu'épisodiquement. Remarquons tout de même que l'adresse de l'octet est donnée en hexadécimal et qu'elle est précédée du signe <.

Nous en avons terminé avec ce chapitre, le cap difficile de la théorie est passé. Asseyons-nous devant Alice et voyons comment nous allons lui faire comprendre puis exécuter un programme écrit en assembleur.

4

ÉTUDE D'UN EXEMPLE

Retrouvons le programme BASIC qui nous a servi, dans le Chapitre 2, à mettre en évidence les qualités graphiques de notre ordinateur.

```
10 POKE 48929 , 65 : REM    LETTRE A
20 POKE 48930 , 1  : REM    MODE ALPHA
30 POKE 48931 , 23 : REM    ROUGE/BLANC
40 POKE 48934 , 27 : REM    LIGNE N° 20
50 POKE 48935 , 27 : REM    COL. N° 27
60 POKE 48936 , 1  : REM    EXECUTION
```

Inutile de le retaper, il n'est présenté ici que pour nous remettre en tête la marche à suivre quand on veut afficher un caractère sur l'écran. Cette marche à suivre, nous allons le voir, est rigoureusement la même en BASIC et en assembleur.

LE MODE ÉDITEUR-ASSEMBLEUR

Respectons les consignes du manuel d'utilisation :

```
CLEAR 100 , 18000      puis ENTER
&                      puis ENTER
```

Dès que ces deux ordres ont été tapés, l'écran s'efface et le fond devient bleu. Nous sommes maintenant sous le contrôle de l'éditeur-assembleur. Le microprocesseur 6803 attend nos ordres. Allons-y sans crainte.

```
1          ORG    $4A00    ; 18944 DECIMAL
2          EXC    DEBUT
3   DEBUT  LDAA   #$41     ; CODE ASCII A
4          STAA   $BF21    ; 48929 DECIMAL
5          LDAA   #$1
6          STAA   $BF22    ; 48930 DECIMAL
7          LDAA   #$17     ; 23 DECIMAL
8          STAA   $BF23    ; 48931 DECIMAL
9          LDAA   #$19     ; 27 DECIMAL
10         STAA   $BF26    ; 48934 DECIMAL
```



```
11         STAA   $BF27    ; 48935 DECIMAL
12         LDAA   #$1
13         STAA   $BF28    ; 48936 EXECUTION
14  FIN    BRA    FIN
```

Voici quelques indications sur les précautions à prendre pour entrer ce programme :

- N'indiquez surtout pas, comme en BASIC, les numéros des lignes que vous voyez ici écrits à gauche. Nous ne les avons fait apparaître que parce que cela vous permettra de vous y retrouver plus facilement dans les explications.
- Sachez que tout ce qui suit un point-virgule est ignoré par l'ordinateur ; le point-virgule est en assembleur l'équivalent de l'instruction BASIC REM.
- Vous devez écrire les mots DEBUT et FIN (que l'on appelle des étiquettes) à partir du bord gauche de l'écran.
- Enfin, n'oubliez jamais de laisser au moins deux espaces libres avant d'écrire une instruction et d'en laisser au moins un entre les différents constituants d'une ligne.

ANALYSE DU PROGRAMME

```
ORG    $4A00    (ligne 1)
```

On s'est bien compris, n'est-ce pas ? Vous avez laissé au moins deux espaces avant de taper les lettres O, R et G et vous avez ensuite laissé au moins un espace avant d'afficher $4A00 ? Et vous n'avez mentionné aucun numéro de ligne ? Bien, alors voici à quoi correspond cette ligne :

La directive ORG apparaît toujours à la première ligne des programmes. Elle donne à l'ordinateur des indications sur la façon dont il devra ranger, dans sa mémoire, les codes machine. Ne cherchons pas, pour l'instant, à y comprendre quoi que ce soit. D'une part parce que nous aurons l'occasion d'y revenir et d'autre part parce que cela n'a pas de rapport direct avec ce que nous sommes en train d'analyser.

```
EXC    DEBUT    (ligne 2)
```

Le programme, à proprement parler, n'a pas encore commencé ; EXC est une directive qui va préciser à la machine à quelle ligne se trouve la première instruction réelle du programme. En l'occurrence, c'est de la ligne 3 qu'il est question, ligne à laquelle nous avons *collé une étiquette*. Au lieu de l'appeler "ligne 3", nous l'appelons "ligne DEBUT". Et, du coup, EXC ligne 3 devient EXC DEBUT. Naturellement, nous aurions pu remplacer le mot DEBUT par ABCDE (ou n'importe quel assemblage de cinq lettres maximum). Il aurait fallu alors faire débuter la ligne 3 par ABCDE.

```
DEBUT   LDAA    #$41    (ligne 3)
```

A partir de là, nous intervenons sur le microprocesseur lui-même. LDAA est une instruction que nous avons déjà rencontrée. Elle signifie que le registre A va contenir la valeur hexadécimale $41 (65 décimal). Le symbole # est là pour indiquer le mode d'adressage immédiat. On ignore quel nombre se trouvait dans l'accumulateur avant cette instruction, mais maintenant on est sûr de la valeur de A : c'est 65 en décimal ($41).

```
STAA    $BF21   (ligne 4)
```

STAA est une instruction très fréquemment utilisée en assembleur : elle signifie que la valeur contenue dans A va devoir être inscrite dans un octet de la mémoire. STAA est l'abréviation de STORE A qui, en anglais, veut dire RANGER A. Le mode d'adressage choisi, l'étendu, nous laisse entendre que $BF21 (48929 décimal) est l'adresse d'un octet de la mémoire. En définitive, c'est dans cet octet que sera rangé le contenu de A. Cette instruction est donc l'équivalent assembleur de la ligne BASIC 10 POKE 48929,65 car, ne le perdons pas de vue, le registre A contient le nombre 65.

Voilà reconstituée, sous sa forme assembleur, la ligne BASIC 10. Il y a donc, pour l'instant, le code ASCII de la lettre A dans l'octet 48929. Avant de poursuivre notre étude, tirons la leçon de ce que l'on vient de faire : pour écrire une valeur dans un octet, on place cette valeur dans le registre A (LDAA en mode immédiat) puis on la range en mémoire (STAA en mode étendu).

```
LDAA    #$1     (ligne 5)
STAA    $BF22   (ligne 6)
```


Nous pouvons aller plus vite maintenant que le principe est compris. On écrit dans l'accumulateur le nombre 1 et on le transfère dans l'octet $BF22 (48930 décimal). On vient, par là-même, de porter son choix sur le type d'affichage alphanumérique standard.

```
LDAA    #$17    (ligne 7)
STAA    $BF23   (ligne 8)
```

Puisque $17 vaut 23 en décimal, ces deux lignes ont pour but d'écrire la valeur 23 dans l'octet $BF23 (48931 décimal). La lettre majuscule A apparaîtra donc en rouge sur fond blanc.

```
LDAA    #$19    (ligne 9)
STAA    $BF26   (ligne 10)
```

Nous continuons à respecter l'enchaînement des lignes BASIC : voici les deux instructions qui correspondent à POKE 48934,27. Le caractère sera visible sur la ligne numéro 20 car, mais vous ne l'avez pas oublié, les lignes 1, 2 ... 7 sont sautées par l'ordinateur.

```
STAA    $BF27   (ligne 11)
```

Aucun ordre LDAA n'ayant été programmé, la valeur contenue par A est restée la même. Cette ligne fournit à l'ordinateur la dernière chose qui lui manquait : le numéro de la colonne sur laquelle se verra la lettre A.

```
LDAA    #$1     (ligne 12)
STAA    $BF28   (ligne 13)
```

Il ne reste plus qu'à rendre effectif l'affichage : ceci se fait en écrivant le nombre 1 dans l'octet $BF28 (48936 décimal).

```
FIN     BRA     FIN     (ligne 14)
```

Nous utilisons tous dans nos programmes BASIC des lignes du genre :

```
100 GOTO 100
```

Nous avons sous les yeux la forme équivalente en assembleur. BRA est une instruction de branchement inconditionnel et comme ce branchement s'effectue à la ligne courante, le programme boucle sans fin. Bien entendu, l'intérêt de cet état de choses sera de nous laisser le temps de voir ce qui va apparaître sur le téléviseur : la lettre A coloriée en rouge sur fond blanc.

EXÉCUTION DU PROGRAMME

Le programme est entièrement tapé, il ne reste plus qu'à commander à l'ordinateur de l'exécuter.

Pressez à la fois les touches CTRL et 1. Le message suivant apparaît en bas de l'écran :

```
LISTING (ECRAN , IMP , ENTER) ?
```

Appuyez sur E puis sur la touche ENTER. Vous devez voir apparaître la totalité de votre programme avec, en plus, divers renseignements. Nous avons, devant nous, ce que l'on appelle le listing d'assemblage.

```
1                                       ORG     $4A00
2                                       EXC     DEBUT
3   4A 00   86 41      DEBUT   LDAA    #$41
4   4A 02   B7 BF 21            STAA    $BF21
5   4A 05   86 01               LDAA    #$1
6   4A 07   B7 BF 22            STAA    $BF22
7   4A 0A   86 17               LDAA    #$17
8   4A 0C   B7 BF 23            STAA    $BF23
9   4A 0F   86 19               LDAA    #$19
10  4A 11   B7 BF 26            STAA    $BF26
11  4A 14   B7 BF 27            STAA    $BF27
12  4A 17   86 01               LDAA    #$1
13  4A 19   B7 BF 28            STAA    $BF28
14  4A 1C   20 F2       FIN     BRA     FIN

0 ERREUR(S) PASSE 1
0 ERREUR(S) PASSE 2
```



```
SYMBOLES :
DEBUT = 4A00        FIN = 4A1C

FICHIER OBJET ?
```

A la dernière ligne, l'ordinateur demande si l'on désire sauvegarder le programme objet sur une cassette. Appuyez sur BREAK. Apparaît alors le message qui nous intéresse :

```
EXECUTION IMMEDIATE ?
```

Répondez en enfonçant la touche 0 (puis ENTER). Le programme est enfin exécuté et cela se traduit par le dessin de la lettre majuscule A, en rouge sur blanc, en bas et à droite de l'écran.

Voilà, nous sommes arrivés à nos fins ; le premier pas est franchi. En fin de compte, programmer en assembleur n'est pas plus compliqué qu'en BASIC. C'est simplement un peu plus long car le microprocesseur ne comprend que des actions élémentaires ; il faut lui *mâcher le travail.*

N'oubliez pas que le programme boucle indéfiniment dans la ligne 14. La touche BREAK ne vous sera d'aucune utilité pour reprendre Alice en main. Il n'y a qu'un seul moyen pour sortir d'un programme assembleur qui boucle : enfoncer la touche INIT placée sur la face arrière de la machine.

Laissez la fin de ce chapitre de côté. Dans un premier temps tout au moins, ce qui suit peut très bien être ignoré. Inutile donc de vous encombrer l'esprit avec des notions qui, en définitive, ne nous concernent pas directement. Vous reprendrez le livre à cette page quand vos connaissances se seront un peu stabilisées.

LE LANGAGE MACHINE

Un ordinateur ne comprend que des nombres et pour lui les expressions LDAA, STAA ou BRA ne veulent absolument rien dire. Il va donc falloir lui traduire le programme que nous avons écrit en assembleur sous la seule forme qui lui soit compréhensible : les codes machine.

Insistons bien sur la différence qu'il y a entre les langages assembleurs et machine : le premier est conçu pour l'esprit humain et il est

formé d'instructions qui ont un sens pour nous. Quand on écrit LDAA, on sait très bien ce qui se passera dans le processeur, on n'a pas besoin de faire un gros effort pour comprendre que le contenu du registre A sera modifié et remplacé par une nouvelle valeur. La forme assembleur permet d'écrire des programmes qui soient lisibles, des programmes qui soient constitués de mots ou d'abréviations dont on s'habituera très vite à connaître le sens.

Quant au langage machine, il est constitué d'une série de nombres que l'ordinateur, lui, est capable d'interpréter. Il y a naturellement une correspondance absolue entre les instructions assembleur et les codes machine qui leur sont relatifs.

Livrons-nous, pour la première et pour la dernière fois, à la traduction en langage machine du programme que nous avons écrit en assembleur. C'est Alice elle-même qui se charge normalement de ce travail. Servez-vous du tableau de l'Annexe C.

```
3    4A 00    86 41      DEBUT    LDAA        #$41
4    4A 02    B7 BF 21            STAA        $BF21
```

86 est l'équivalent pour la machine de LDAA. Vous constatez que LDAA se code de différentes façons suivant le mode d'adressage. Celui qui nous intéresse est l'immédiat, dans la première colonne donc. Il faudra toujours se souvenir que les codes machine sont écrits en hexadécimal ; 86, ainsi que tous les autres codes de ce tableau, respecte cette règle.

41 est le nombre qui suit 86. Le microprocesseur, après avoir interprété 86, s'attendra à ce qu'on lui dise avec quel nombre il doit charger A. Puisque 41 vient à la suite de 86, il comprendra que la valeur 41 (65 décimal) doit être placée dans l'accumulateur.

B7 est le code machine de l'instruction STAA. Il doit être choisi dans la bonne colonne, celle de l'adressage étendu. C'est en effet ce mode que nous avons décidé d'utiliser en écrivant le programme assembleur. Quand l'ordinateur va lire ce code, il saura qu'il lui faut alors s'intéresser aux deux valeurs suivantes.

BF et 21 forment le nombre hexadécimal BF21. La machine, ayant rencontré B7, comprendra qu'elle doit placer le contenu du registre A dans un octet de la mémoire ; dans le numéro 48929 ($BF21) bien sûr.

Arrêtons nos efforts de traduction, cela devient vite fastidieux. Retenons que notre programme assembleur correspond, pour le processeur, à la suite de nombres :

86, 41, B7, BF, 21, etc.

L'ensemble de ces valeurs est appelé le code machine et il est directement exécutable. Il suffit, pour le faire exécuter, de brancher le processeur sur la première de ces valeurs. C'est exactement ce qui est réalisé quand nous répondons OUI à la question EXECUTION IMMEDIATE ?

Il reste pour clore ce chapitre, à indiquer comment l'ordinateur fait pour savoir à quel endroit de sa mémoire se trouve le premier des codes machine. Il utilise, pour cela, la première indication du programme :

```
ORG     $4A00
```

Cette directive lui donne l'adresse à partir de laquelle sera rangé le code machine. Puisque $4A00 est égal à 18944 en notation décimale, on en déduit que les nombres 86, 41, B7, etc., seront écrits dans les octets 18944, 18945, 18946, etc.

Vous pourrez vérifier ceci en étudiant les nombres qui, sur le listing d'assemblage, suivent les numéros de ligne :

```
3    4A 00    86 41
4    4A 02    B7 BF 21
```

86 est écrit dans l'octet $4A00, B7 est écrit dans l'octet $4A02, etc.

5

ÉLÉMENTS DE PROGRAMMATION DU 6803

LDAA

Abréviation de LOAD A (charger A), cette instruction permet de placer une valeur 8 bits dans le registre A.

Les modes d'adressage possibles sont l'immédiat, le direct, l'indexé et l'étendu.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (25 octets)

Programme assembleur

```
1            ORG     $4A00
2            EXC     DEBUT
3     DEBUT  LDX     #$BF21   ; 48929 DECIMAL
4            LDAA    #$64     ; 100 : ASCII d
5            STAA    $0,X     ; 48929
6            LDAA    #$11     ; 17 : ALPHA SOULIGNE
7            STAA    $1,X     ; 48930
8            LDAA    #$1E     ; 30 : ROUGE/TURQUOISE
9            STAA    $2,X     ; 48931
10           STAA    $5,X     ; 48934 LIGNE
11           LDAA    #$1      ; 1
12           STAA    $6,X     ; 48935 COLONNE
13           STAA    $7,X     ; 48936 EXECUTION
14    FIN    BRA     FIN
```

Commentaires

Voici un programme dont le rôle est de faire apparaître un caractère sur l'écran. Il est donc identique, quant à son but, à celui que nous avons rencontré dans le chapitre précédent. Par contre, sa présentation est nouvelle par le fait que nous allons utiliser le mode d'adressage indexé.

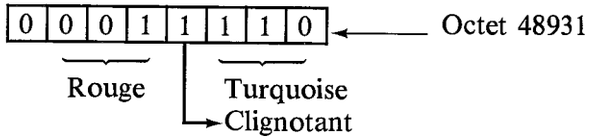
Ligne 3 : le registre X va contenir le nombre $BF21 (48929 décimal). Ceci est tout à fait possible puisque X est un registre 16 bits et que son contenu peut atteindre la valeur 65535 (FFFF hexadécimal).

Ligne 4 : on écrit dans l'accumulateur le nombre 100 c'est-à-dire le code ASCII de la lettre *d* minuscule.

Ligne 5 : STAA est l'instruction qui recopie le contenu du registre A dans un octet de la mémoire. Pour savoir lequel, il suffit d'interpréter la notation $0,X. Celle-ci indique que l'octet en question a pour adresse le nombre contenu dans le registre X (soit 48929) auquel on ajoute $0 (auquel on n'ajoute rien du tout, donc). Conclusion : le nombre décimal 100 est rangé dans l'octet 48929.

Lignes 6 et 7 : une nouvelle valeur est écrite dans A et se voit transférée dans l'octet 48930 (soit 1 + X). Rappelons que lorsque le nombre 17 ($11) est inscrit dans cet octet, le type de l'affichage qui va avoir lieu est l'alphanumérique souligné.

Lignes 8 et 9 : l'octet 48931 (2 + X) est chargé avec le nombre 30 ($1E).



La lettre que l'on se propose de faire apparaître sera donc de couleur rouge sur fond bleu clair et elle clignotera.

Ligne 10 : A n'a pas changé de valeur et l'instruction STAA copie dans l'octet 48934 (5 + X) le nombre 30. Souvenons-nous que cet octet correspond à la ligne d'affichage et que les numéros allant de 1 à 7 ne sont pas employés ; notre caractère sera donc visible à l'avant-dernière ligne.

Lignes 11 et 12 : après le numéro de la ligne, voici le numéro de la colonne ; il s'agit de la deuxième colonne en partant de la gauche.

Ligne 13 : il ne nous reste plus qu'à donner l'ordre d'affichage ; c'est ce que nous réalisons en portant l'octet 48936 (7 + X) au niveau 1.

Ligne 14 : le programme boucle sans fin sur cette ligne ; il n'y a pas ainsi de retour prématuré au BASIC et nous avons le temps de

voir que notre programme a été exécuté correctement. N'oubliez toutefois pas que seul un appui sur la touche INIT pourra vous rendre le contrôle de votre ordinateur.

LDAB

Cette instruction est absolument identique à LDAA, mais elle concerne le chargement du registre B. Là encore, les modes d'adressage possibles sont l'immédiat, le direct, l'indexé et l'étendu.

LDD

Abréviation de LOAD D (charger D), cette instruction permet de placer une valeur 16 bits dans le registre X.

Les modes d'adressage possibles sont l'immédiat, le direct, l'étendu et l'indexé.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (19 octets)

Programme assembleur

```
1            ORG     $4A00
2            EXC     DEBUT
3   DEBUT    LDX     #$BF21    ; 48929 DECIMAL
4            LDD     #$6411    ; 100 DANS A et 17 DANS B
5            STD     $0,X      ; 48929 et 48930
6            LDD     #$1E01    ; 30 DANS A et 1 DANS B
7            STAA    $2,X      ; 48931
8            STD     $5,X      ; 48934 et 48935
9            STAB    $7,X      ; 48936
10  FIN      BRA     FIN
```

Commentaires

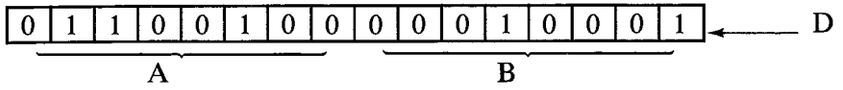
Voici un programme qui réalise, comme le précédent, l'affichage d'un caractère sur l'écran. Il a l'avantage de se présenter sous une forme plus compacte car il utilise les services du double accumulateur D.

Ligne 3 : le programme démarre avec le registre X pointant sur l'octet 48929.

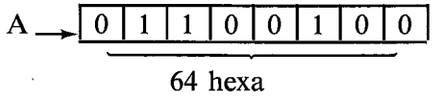
Ligne 4 : l'accumulateur D est chargé sur le mode immédiat, avec la valeur $6411. Faisons l'effort de voir à quoi ce nombre correspond au niveau du binaire.

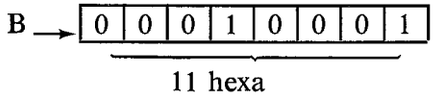
$$\$6411 = \underbrace{0110}_{6}\ \underbrace{0100}_{4}\ \underbrace{0001}_{1}\ \underbrace{0001}_{1}$$

Nous pouvons alors en déduire la configuration du registre D :



et, par la même occasion, celles des accumulateurs 8 bits A et B :





Ligne 5 : le contenu 16 bits de D est écrit en mémoire. Ceci ne peut, naturellement, s'effectuer sur le seul octet 48929. Son suivant, le numéro 48930 est lui aussi mis à contribution. Ainsi, quand l'instruction STD $0,X aura été exécutée, l'octet 48929 contiendra le nombre $64 et l'octet 48930 le nombre $11. Résumons ce qui vient d'être dit : les deux lignes 4 et 5 sont équivalentes à la série d'instructions suivantes :

```
LDAA #$64 ; STAA $0,X ; LDAB #$11 ; STAB $1,X
```

Le caractère qui sera affiché sera la lettre *d* soulignée.

Lignes 6, 7, 8 et 9 : on écrit dans D le nombre $1E01. Ce qui revient à dire que l'on charge les deux accumulateurs A et B avec les valeurs $1E et $1. Ensuite on écrit :

$1E dans l'octet 48931	(couleur du caractère)
$1E dans l'octet 48934	(n° de ligne)
$01 dans l'octet 48935	(n° de colonne)
$01 dans l'octet 48936	(exécution)

LDS LDX

Ces deux instructions s'utilisent de la même façon que LDD. Elles servent à placer dans les registres S et X des valeurs de 16 bits.

JSR

Cette instruction est l'abréviation de Jump to SubRoutine. *Elle indique au microprocesseur à quelle adresse le sous-programme doit démarrer. Le retour du sous-programme s'effectue à l'instruction qui suit JSR. On peut utiliser les modes d'adressage indexé, étendu et direct.*

Exemple : MODE D'ADRESSAGE ÉTENDU (8 octets)

Programme assembleur

```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  LDAA   #$66     ; FREQUENCE DU DO
4          LDAB   #$20     ; DUREE DU SON
5          JSR    SON
6          RTS
7   SON    =      $FFAB    ; ROUTINE DU SON
```

Commentaires

Voici un programme clé pour notre sujet. D'une part il va nous permettre d'approfondir nos connaissances sur la façon dont fonctionne notre ordinateur ; d'autre part il va nous faire comprendre comment nous pouvons, en assembleur, jouer des notes de musique.

Ligne 3 : le premier accumulateur est chargé avec le nombre $66 (102 décimal). Notons simplement, pour l'instant, que ce nombre correspond à la fréquence de la note DO du deuxième octave.

Ligne 4 : pour définir une note, il faut aussi des indications sur sa durée. Le second registre 8 bits est là pour cela.

Ligne 5 : c'est le cœur du programme. On peut comparer JSR à l'instruction BASIC GOSUB : le microprocesseur va partir exécuter les codes machine qui se trouvent à partir de l'adresse $FFAB (soit

65451 en décimal). Vous avez du mal à comprendre, vous ne voyez aucune signification à ces nombres, vous trouvez que ce n'est pas clair ? Disons-le tout net : dans l'état d'avancement de notre étude, ceci ne peut pas être clair. C'est même le trou noir. On ne sait pas du tout quel genre d'instructions l'ordinateur va aller exécuter ! Alors, que doit-on retenir de tout cela ?

- Premièrement que le nombre 65451 n'a pas été choisi au hasard ; il fait partie d'une zone mémoire de la machine que l'on appelle le moniteur.
- Deuxièmement, qu'à partir du moment où un programme se branche à cette adresse, une note de musique est émise par le haut-parleur.
- Troisièmement que la fréquence et la durée de cette note sont en relation directe avec les contenus des registres A et B.
- Quatrièmement, qu'après avoir joué une note, le processeur retrouve l'instruction qui suit immédiatement JSR. Dans notre cas, puisqu'il s'agit de RTS, le retour au BASIC est programmé.

Ligne 7 : c'est la première fois que l'on rencontre l'instruction d'affectation. Vous avez naturellement deviné à quoi sert cette directive : elle affecte la valeur $FFAB à l'étiquette SON. A chaque fois que l'assembleur butera sur le mot SON, il le remplacera par le nombre $FFAB. C'est d'ailleurs ce que l'on peut voir en analysant les codes machine relatifs à la ligne 5 : JSR SON a été traduit par BD FF AB (BD est le code de JSR).

STAA

Abréviation de STORE A (ranger A), STAA permet de placer dans un octet de la mémoire la valeur 8 bits qui a précédemment été chargée dans le registre A. Trois modes d'adressage : l'indexé, l'étendu et le direct.

Exemple : MODE D'ADRESSAGE ÉTENDU (7 octets)

Programme assembleur

```
1          ORG    $4A00     ; 18944 EN DECIMAL
2          EXC    DEBUT
3   DEBUT  LDAA   $4B00     ; 19200 EN DECIMAL
4          STAA   $4B01     ; 19201 EN DECIMAL
5          RTS
```

Programme BASIC

```
10 INPUT"DONNEZ UN NOMBRE";N : POKE 19200,N
20 EXEC 18944
30 PRINT"L'OCTET 19200 CONTIENT LA VALEUR";PEEK(19200)
40 PRINT"L'OCTET 19201 CONTIENT LA VALEUR";PEEK(19201)
50 GOTO 10
```

Commentaires

Le rôle du programme assembleur se réduit à peu de choses : l'accumulateur A est chargé avec la valeur écrite dans l'octet 19200 (ligne 3) puis cette valeur est elle-même transférée dans l'octet 19201 (ligne 4). Notre programme se contente donc de recopier à l'adresse 19201 le nombre qu'il a trouvé dans l'octet précédent. Nous allons nous assurer de cela grâce au BASIC, montrant en détail comment il est possible de mêler étroitement les deux langages compris par Alice. Après tout, si nous nous donnons la peine d'étudier l'assembleur, c'est dans l'espoir de l'utiliser un jour dans nos programmes BASIC, non ?

Commençons par le commencement : vous êtes passé sur le mode assembleur, vous avez tapé au clavier les cinq lignes de votre programme, vous vous êtes assuré qu'il ne contenait pas d'erreur puis vous en avez demandé l'exécution. Et vous n'avez rien vu de spécial ; vous vous êtes retrouvé devant un écran vide et sous le contrôle du BASIC. Pourtant il s'est passé quelque chose d'intéressant dont nous allons tirer parti. L'ordinateur a, préalablement à l'exécution du programme, traduit l'assembleur en codes machine et rangé ces codes sur sept octets à partir de l'adresse 18944 (4A00 hexa). On le vérifie ?

```
FOR I = 18944 TO 18950 : PRINT PEEK(I) ;: NEXT
```

Réponse : 182 75 0 183 75 1 57

C'est bien le résultat attendu : 182 (B6 hexa) est le code décimal de l'instruction LDAA en mode étendu, 75 est la traduction de 4B, ... et 57 (39 hexa) correspond à la commande RTS.

Ainsi donc, nous disposons en mémoire d'un programme écrit en langage machine. Nous allons pouvoir, autant de fois que nous le souhaitons, le faire exécuter en nous servant de la commande BASIC EXEC. Cette instruction, en effet, ne fait pas autre chose que de lancer l'exécution... d'un programme machine. C'est tout à fait ce qui nous convient, n'est-ce pas ?

Voyons, en pratique, comment il faut s'y prendre :

Ligne BASIC 10 : un nombre N est demandé et son écriture dans l'octet 19200 est réalisée par l'instruction POKE.

Ligne BASIC 20 : c'est l'exécution du programme machine. Le microprocesseur écrit dans l'octet 19201 le nombre qui se trouve à l'adresse précédente puis, rencontrant l'instruction RTS, *redonne la main au BASIC.*

Lignes BASIC 30 et 40 : voici la confirmation de ce que nous avions prévu ; on retrouve toujours le même nombre dans les octets 19200 et 19201.

Il n'y a rien d'autre à ajouter concernant le programme étudié. Toutefois, ne terminons pas notre étude sans faire deux remarques au sujet de la connexion, sur notre ordinateur, du BASIC et de l'assembleur.

La première a trait au fait que l'on peut passer sans aucune contrainte d'un langage à l'autre. On peut très bien taper un programme assembleur puis passer (touche BREAK appuyée deux fois) au BASIC. Mais on peut aussi, sans ennui, retrouver ensuite (touche %) le programme assembleur (qui n'aura donc pas été détruit). L'ordinateur a donc la possibilité de garder en mémoire un programme assembleur et un programme BASIC.

La seconde remarque est la suivante : si l'on enregistre sur une cassette le programme BASIC qui a servi à notre étude avec la ferme intention de s'en resservir un peu plus tard, on risque fort d'avoir quelques problèmes. Effectivement, dès que l'ordinateur est éteint, il oublie tout et en particulier les sept nombres que l'on a écrits dans les octets 18944 et suivants. Alors faudra-t-il à chaque fois retaper le programme assembleur ? Non, la solution est plus simple : il suffit d'inclure dans le programme BASIC la liste des codes machine et de les faire écrire directement dans les octets voulus :

```
4 CLEAR 100 , 18000
5 DATA 182 , 75 , 0 , 183 , 75 , 1 , 57
6 FOR I = 18944 TO 18950 : READ J
7 POKE I , J : NEXT
```

L'ajout de ces lignes au programme BASIC le rend totalement autonome. Il peut alors être enregistré.

STAB

Cette instruction, identique dans son principe d'utilisation à STAA, commande le passage d'une donnée 8 bits du registre B vers un octet de la mémoire.

STX

La valeur 16 bits contenue dans le registre X est rangée en mémoire. Le rangement s'effectue sur deux octets consécutifs. On peut utiliser les modes d'adressage étendu, indexé et direct.

Exemple : MODE D'ADRESSAGE ÉTENDU (20 octets)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3                ;
4   DEBUT        JSR     EFF       ; CLS
5                ;
6                LDAA    #$0
7                STAA    $E8       ; MISE A 0 OCTET E8
8                LDX     $50A
9                STX     $3280     ; LIGNE ET COLONNE
10               LDAA    #$41      ; CODE LETTRE A
11               JSR     AFCAR     ; AFFICHAGE CARACTERE
12  FIN          BRA     FIN
13               ;
14               ;
15  EFF          =       $FBD4
16  AFCAR        =       $F9C6
```

Commentaires

Ligne 4 : vous vous souvenez de la routine $FFAB qui nous a permis de programmer un son en assembleur ? Eh bien, en voilà une autre tout aussi intéressante. A chaque fois qu'un programme se branche à l'adresse $FBD4, l'écran est entièrement effacé. JSR EFF reconstitue tout simplement la fonction CLS.

Ligne 11 : nous avons nettoyé l'écran pour y écrire quelque chose ; alors allons-y ! Envoyons, toujours avec JSR, le microprocesseur exécuter une nouvelle routine baptisée AFCAR. Ce sous-programme réalise pour nous l'affichage d'un caractère ; il suffit de brancher l'ordinateur à l'adresse $F9C6. Voyons les détails et appliquons à la lettre les renseignements fournis par le constructeur (cf. guide d'instructions assembleur).

Ligne 10 : avant l'appel de la routine, l'accumulateur A doit contenir le code ASCII du caractère à afficher. Mettons donc dans ce registre le nombre $41 (65 décimal) correspondant à la lettre majuscule A.

Lignes 6 et 7 : le registre A ne sert ici que d'intermédiaire. On y écrit la valeur 0 et on la recopie dans l'octet $E8 ; ceci pour indiquer à la machine que l'on veut voir notre caractère sur l'écran.

Ligne 8 : X, registre 16 bits ne l'oublions pas, est chargé avec le nombre hexadécimal $50A. Faisons un petit effort pour convertir ce nombre en binaire :

0	0	0	0	0	1	0	1	0	0	0	0	1	0	1	0

On a l'habitude de considérer les huit bits de gauche comme l'octet de poids fort du registre X et les huit bits de droite comme l'octet de poids faible. Ceci permet d'obtenir la valeur décimale contenue dans ce registre à l'aide de la relation suivante :

256*octet de poids fort (en décimal) + octet de poids faible (en décimal)

Pour ce qui concerne notre exemple, c'est le nombre 1290 (256*5 + 10) qui est maintenant placé dans X.

Ligne 9 : le contenu du registre X est rangé à l'adresse $3280. Comme il n'est pas question de décharger X sur le seul octet $3280, il sera nécessaire d'en utiliser un deuxième : la partie forte de X (5 en décimal) se retrouve dans l'octet $3280 et sa partie faible (10 en décimal) dans l'octet $3281.

Or, que lisons-nous sur le manuel livré avec Alice ? Qu'avant d'appeler la routine d'affichage, il est nécessaire d'avoir placé dans l'octet $3280 le numéro de la ligne et dans l'octet d'après le numéro de la colonne. Voici donc que s'explique pourquoi la lettre A est apparue à tel endroit du téléviseur plutôt qu'à tel autre.

Ligne 12 : on oblige le programme à boucler sur lui-même car, s'il se terminait, le retour immédiat au BASIC nous empêcherait de voir quoi que ce soit sur l'écran.

STD STS

On utilise ces deux instructions de la même façon que STX. Elles permettent de placer le contenu du registre spécifié dans un emplacement constitué par deux octets de la mémoire.

ADDA

Cette instruction va ajouter les contenus du registre A et d'un octet mémoire. Le résultat de l'addition sera alors placé dans A. On peut utiliser les modes d'adressage immédiat, indexé, étendu et direct.

Exemple : MODE D'ADRESSAGE INDEXÉ (10 octets)

Programme assembleur

```
1          ORG     $4A00
2          EXC     DEBUT
3   DEBUT  LDX     #$4B00    ; 19200 EN DECIMAL
4          LDAA    $0,X
5          ADDA    $1,X
6          STAA    $2,X
7          RTS
```

Programme BASIC

```
10 INPUT " PREMIER NOMBRE " ; N : POKE 19200,N
20 INPUT " DEUXIEME NOMBRE " ; M : POKE 19201,M
30 EXEC 18944
40 PRINT " LA SOMME VAUT " : PEEK (19202) : GOTO10
```

Commentaires

Ligne 4 : le registre A est chargé avec la valeur contenue dans l'octet pointé par X, c'est-à-dire l'octet 19200 (ou 4B00 hexa). La ligne BASIC 10 a fait entrer par POKE, dans cet octet, le premier terme de la somme. N ne doit pas, bien sûr, dépasser 255 sinon le message d'erreur FC apparaîtrait sur l'écran.

Ligne 5 : ADDA $1,X signifie que l'on ajoute la valeur de A et la valeur inscrite dans l'octet ayant pour adresse X + 1. De ce fait cette ligne additionne le premier et le second nombre de la somme, second nombre qui a été placé par POKE dans l'octet 19201 (ligne BASIC 20).

Ligne 6 : le résultat de l'opération étant dans le registre A, il ne reste plus qu'à ranger la somme obtenue dans l'octet 19202 puisque c'est là que le BASIC ira chercher la réponse.

Faisons tourner le programme :

Premier nombre :	Deuxième nombre :	Somme :
10	20	30
100	0	100
200	60	4
250	250	244

Il n'y a rien à redire pour les deux premiers cas : les résultats obtenus sont conformes à nos prévisions. Quand aux calculs suivants, il ne sera pas bien compliqué d'établir leur cohérence : puisque A est un registre 8 bits, le nombre le plus grand que l'on puisse y écrire est 11111111 (255 en décimal). Si, à ce moment-là, on essaie d'ajouter 1, le registre repassera à 00000000 (0 en décimal) ; et c'est ce qui explique que 260 soit devenu 4 dans notre troisième somme. D'une manière analogue, en ajoutant 250 et 250, on ne trouve pas 500 mais 500 – 256 soit 244.

ADDB

On procède avec cette instruction de la même façon qu'avec ADDA. ADDB permet d'ajouter les valeurs inscrites dans le registre B et dans un octet mémoire. Le résultat de l'addition se trouve dans B.

ABA ABX

Ces deux instructions ajoutent directement les contenus de deux registres. ABA additionne A et B (résultat dans A) et ABX additionne B et X (résultat dans X). On n'emploie que le mode d'adressage inhérent.

ADDD

Il s'agit là encore d'ajouter le contenu d'un registre à une valeur prise en mémoire. D étant un registre 16 bits, on pourra donc additionner deux valeurs elles-mêmes de 16 bits. Modes d'adressage possibles : immédiat, indexé, étendu et direct.

Exemple : MODE D'ADRESSAGE ÉTENDU (10 octets)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3     DEBUT      LDD     $4B00     ; 19200 EN DECIMAL
4                ADDD    $4B02     ; 19202 EN DECIMAL
5                STD     $4B04     ; 19204 EN DECIMAL
6                RTS
```

Programme BASIC

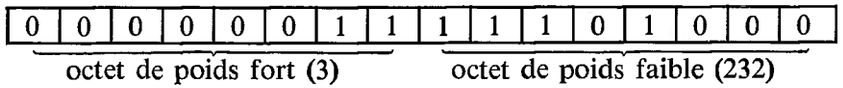
```
10 INPUT " PREMIER NOMBRE " ; N
20 N1 = INT(N/256) : POKE 19200 , N1
30 N2 = N-N1*256 : POKE 19201 , N2
40 INPUT " DEUXIEME NOMBRE " ; M
50 M1 = INT(M/256) : POKE 19202 , M1
60 M2 = M-M1*256 : POKE 19203 , M2
70 EXEC 18944
80 PRINT"LA SOMME VAUT";256*PEEK(19204)+PEEK(19205)
```

Commentaires

Ligne 3 : le registre D est chargé avec le premier nombre N. Tenant compte du fait que D est un registre 16 bits, l'instruction LDD va chercher en mémoire la valeur des deux octets 19200 et 19201. Regardons, en supposant N égal à 1000, comment cela se passe.

N1 = INT(1000/256) soit N1 = 3. L'octet 19200 contient le nombre 3.
N2 = 1000 − 3*256 soit N2 = 232. L'octet 19201 contient le nombre 232.

Après exécution de la ligne, D se présente sous la forme suivante :



En appliquant, à titre de vérification, la formule
256 * poids fort + poids faible

on obtient : 256*3 + 232 c'est-à-dire le nombre 1000.

Ligne 4 : l'opération est effectuée. Le premier nombre, N, se trouve déjà dans D et le second, M, doit être recherché en mémoire. Choisissons, par exemple, M égal à 2000. Les lignes BASIC 50 et 60 ont placé la valeur de M dans les octets 19202 et 19203 de la façon suivante : la partie de poids fort de 2000, c'est-à-dire 7 (M1) est inscrite à l'adresse 19202 et sa partie faible 208 (M2) à l'adresse 19203.

On peut résumer les lignes 3 et 4 en disant que tout s'est passé comme si l'on avait ajouté directement les nombres N et M écrits aux adresses 19200 et 19201 d'une part, 19202 et 19203 d'autre part. Il n'existe malheureusement pas d'instruction qui le fasse de façon directe et le passage par le registre D a été obligatoire.

Ligne 5 : le résultat de l'addition ayant été mis dans D, il ne reste plus qu'à ranger le contenu de ce registre dans un emplacement mémoire où la ligne BASIC 80 pourra aller le chercher. Puisque ce résultat est un nombre de 16 bits, l'instruction STD va le placer sur deux octets, à savoir partie forte à l'adresse 19204 et partie faible à l'adresse 19205.

En faisant exécuter le programme sur quelques exemples, vous remarquerez qu'à chaque fois que la somme dépasse 65535, le registre D la fait repartir à zéro. 65535 est en effet la valeur décimale la plus grande possible que l'on puisse écrire sur 16 bits.

PREMIER NOMBRE ? 50000
DEUXIEME NOMBRE ? 20000
LA SOMME VAUT 4464 soit 70000 − 65536.

SUBA

SUBA est une instruction qui permet de retrancher au contenu de l'accumulateur A la valeur inscrite dans un octet de la mémoire. Le résultat de la soustraction est écrit dans A. Les modes d'adressage immédiat, indexé, étendu et direct sont utilisables.

Exemple : MODE D'ADRESSAGE INDEXÉ (10 octets)

Programme assembleur

```
1          ORG     $4A00
2          EXC     DEBUT
3   DEBUT  LDX     OCTET
4          LDAA    $0,X
5          SUBA    $1,X
6          STAA    $2,X
7          RTS
8   OCTET  DFD     $4B00
```

Programme BASIC

```
10 INPUT " PREMIER NOMBRE " ; N : POKE 19200 , N
20 INPUT " DEUXIEME NOMBRE " ; M : POKE 19201 , M
30 EXEC 18944
40 PRINT " LA DIFFERENCE VAUT " ; PEEK(19202) : GOTO 10
```

Commentaires

Ligne 3 : le registre X est chargé avec le nombre hexadécimal 4B00 (19200) mais cette fois sans utiliser le mode d'adressage immédiat. Nous avons voulu mettre en avant une des directives de l'assembleur, DFD ; vous la voyez apparaître à la ligne 8. Elle a pour rôle de réserver deux octets en mémoire et d'y inscrire le nombre 16 bits qui est écrit à sa

suite. Peu nous importe de savoir où se trouvent précisément ces deux octets. La seule chose qui compte est qu'à chaque fois que l'assembleur rencontrera l'étiquette OCTET, il nous fournira la valeur contenue par les deux octets en question. Voilà comment s'explique que le registre X a pour contenu le nombre 4B00.

Ligne 4 : LDAA $0,X signifie que l'on place dans l'accumulateur le nombre qui se trouve dans l'octet 19200 (0 + X) : c'est la ligne BASIC 10 qui a auparavant inscrit dans cet octet la valeur du premier terme N de la différence.

Ligne 5 : on retranche à N le nombre M contenu dans l'octet 19201 (1 + X). Le nombre M est connu du programme dès que la ligne BASIC 20 est exécutée. Notons que la soustraction se fait toujours dans le même sens : c'est l'octet mémoire qui est retranché à l'accumulateur et non le contraire.

Ligne 6 : puisque le résultat est maintenant dans A, il ne reste plus qu'à stocker ce résultat dans un emplacement mémoire que le programme BASIC pourra retrouver. Il s'agit, cela se voit bien, de la case 19202.

Avant de passer aux instructions suivantes, ne manquez pas de faire tourner ce programme en lui proposant des calculs du genre 10 – 11 ou 0 – 255 et en analysant les réponses de l'ordinateur.

SUBB SUBD SBA

SUBB est utilisable de la même façon que SUBA mais concerne le registre B. L'instruction SUBD, pour sa part, effectue des soustractions 16 bits. Quant à l'instruction SBA, elle permet de soustraire B de A, le résultat étant dans A. On emploie cette dernière commande sur le mode d'adressage inhérent.

MUL

Cette instruction, n'acceptant que le seul mode d'adressage inhérent, effectue le produit de deux valeurs 8 bits contenues dans les accumulateurs A et B. Le résultat de la multiplication, sur 16 bits, est inscrit dans le registre D, faisant ainsi disparaître les valeurs initiales de A et B.

Exemple : MODE D'ADRESSAGE INHÉRENT (20 octets)

Programme assembleur

```
1            ORG     $4A00
2            EXC     DEBUT
3   DEBUT    LDAA    $4B00     ; 19200 EN DECIMAL
4            LDAB    #$2
5            MUL
6            STD     $4B01
7            LDAA    $4B00
8            LDAB    $4B00
9            MUL
10           STD     $4B03
11           RTS
```

Programme BASIC

```
10 INPUT "DONNEZ UN NOMBRE" ; N : POKE 19200,N
20 EXEC 18944
30 PRINT "VOICI SON DOUBLE"; 256*PEEK(19201)+PEEK(19202)
40 PRINT "VOICI SON CARRE"; 256*PEEK(19203)+PEEK(19204)
```

Commentaires

C'est la ligne BASIC 10 qui a chargé par POKE, dans l'octet 19200, le nombre N dont on va calculer le double puis le carré. N est un

nombre 8 bits et il n'est pas possible de proposer une valeur supérieure à 255 sans que l'ordinateur ne renvoie le message FC Error.

Lignes 3 et 4 : le registre A va contenir la valeur de N et le chiffre 2 est placé dans B.

Ligne 5 : les valeurs des deux accumulateurs sont multipliées, ce qui revient pour nous à calculer le double du nombre qui a été donné au départ. Le résultat, sur 16 bits, de cette opération se retrouve alors dans D et, naturellement, les valeurs qui étaient dans A et B sont perdues ; n'oublions pas que A et B constituent les registres de poids fort et de poids faible de D.

Ligne 6 : le résultat de la multiplication est rangé en mémoire dans les octets 19201 et 19202 : la ligne 50 retrouve pour nous la valeur décimale de cette réponse non sans avoir, mais c'est déjà de l'histoire ancienne, multiplié le contenu de l'octet fort par 256.

Lignes 7 et 8 : on réintroduit dans A et dans B les nombres nécessaires à la suite du programme : puisqu'il s'agira de calculer un carré, les deux registres doivent contenir la même valeur, en l'occurrence N.

Lignes 9 et 10 : le produit de N par lui-même est calculé, le résultat placé dans D et ce registre rangé dans les octets 19203 et 19204, octets que le programme BASIC retrouvera à la ligne 40.

Il n'y a rien d'autre à ajouter concernant l'instruction MUL.

Avant de passer au programme suivant, pourquoi ne pas réécrire le même que celui-ci mais en le débutant par LDX #$4B00 et en utilisant le plus souvent possible l'adressage indexé. Le lecteur ne devra jamais perdre de vue que si le mode indexé est un peu moins lisible que l'étendu, il a en revanche une qualité non négligeable : il est bien souvent moins gourmand en octets que ce dernier.

BEQ BNE BRA BSR

Tous les programmes que nous avons étudiés jusqu'à maintenant étaient conçus sur le type séquentiel, ce qui veut dire que les instructions étaient exécutées les unes à la suite des autres, dans l'ordre où elles avaient été écrites. Nous savons tous qu'en BASIC il est possible, avec des instructions comme GOTO par exemple, d'empêcher le programme de se dérouler normalement en l'obligeant à se brancher à un numéro de ligne choisi par nous. Voyons comment nous devrons nous y prendre pour obtenir le même effet. Nous retrouverons, après quelques explications théoriques, l'étude d'exemples bien concrets.

BEQ

Branch if Equal

Branchement si égal

Le branchement à l'une des parties du programme machine ne se fera que si l'une des deux conditions suivantes vient d'être réalisée :

1 — soustraction entre deux nombres égaux.

2 — comparaison entre deux nombres égaux.

C'est l'octet qui suit immédiatement l'instruction BEQ qui, en mode complément à 2, indiquera alors au processeur quelle autre partie du programme devra être exécutée.

Dans le cas d'une comparaison ou d'une soustraction entre deux nombres différents, BEQ n'aura aucun effet et c'est l'instruction écrite à la ligne d'après qui sera exécutée.

Pour résumer, disons que l'instruction BEQ s'utilise de la même façon que la phrase BASIC bien connue :

```
IF A = B THEN...
```

BNE

Branch if Not Equal
Branchement si non égal

Voici l'instruction contraire de la précédente. Cette fois le branchement ne sera effectué que dans le cas où l'une des deux situations suivantes se sera présentée :

1 — soustraction entre deux nombres différents.
2 — comparaison effectuée sur deux nombres différents.

Ici aussi l'endroit du programme où le branchement devra se faire sera indiqué par l'octet placé après l'instruction BNE. Le nombre écrit dans cet octet devra l'être sous la forme complément à deux.
On peut trouver l'équivalent BASIC de BNE en écrivant :

```
IF A < > B THEN...
```

BRA

Branch Always
Branchement dans tous les cas

Le branchement à la partie du programme indiquée par l'octet qui suit l'instruction BRA est un branchement inconditionnel. Ce type de branchement ne se préoccupe pas de savoir si telle ou telle condition a été réalisée : il s'effectue de toute manière.
Vous aurez reconnu en BRA l'équivalent assembleur de la commande BASIC GOTO.

BSR

Branch at Subroutine
Branchement vers un sous-programme

Après GOTO, voici GOSUB : BSR est en effet l'instruction de branchement qui permet de sauter jusqu'à un sous-programme. Il s'agit

comme pour BRA d'un branchement inconditionnel qui s'effectuera dans tous les cas.
Il est inconcevable, en BASIC, d'écrire un GOSUB sans prévoir le RETURN qui nous ramènera au programme principal.
Il en est de même en assembleur et il nous faudra toujours penser à terminer nos sous-programmes par une instruction que nous avons rencontrée dès nos premiers exemples : RTS (*ReTurn from Subroutine*).

INCA

Cette instruction incrémente le registre A, c'est-à-dire qu'elle lui ajoute une unité. Un seul mode d'adressage possible : l'inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (20 octets)

Programme assembleur

```
1            ORG    $4A00
2            EXC    DEBUT
3   DEBUT    LDAA   OCTET    ; FREQUENCE INITIALE
4   SUITE    LDAB   #$1      ; DUREE DE CHAQUE NOTE
5            STAA   OCTET
6            JSR    SON      ; MUSIQUE
7            LDAA   OCTET    ; ON RETROUVE A
8            INCA
9            SUBA   #$0
10           BNE    SUITE
11           RTS
12  SON      =      $FFAB
13  OCTET    DFO    $1
```

Commentaires

Nous en savons maintenant suffisamment pour obtenir de notre ordinateur des effets de son. Après l'étude de ces deux pages, le lecteur sera déjà assez bien armé pour programmer lui-même, en assembleur, l'émission de sons tous plus spéciaux les uns que les autres. Dans les jeux d'action, la commande SOUND aura disparu, remplacée par EXEC ...

Pour notre part, contentons-nous de reproduire un bruit de sirène.

Ligne 3 : le registre A contient la valeur de l'octet que nous avons fait réserver par la directive DFO (ligne 12). C'est donc avec le nombre 1 dans A que démarre le programme.

Ligne 4 : le second registre B contiendra tout au long du programme la même valeur, 1 ; c'est la durée des notes qui apparaît ici.

Ligne 5 : nous prenons la précaution de mettre de côté (dans l'octet appelé OCTET) le contenu de A. La raison en est simple : nous aurons besoin de cette valeur un petit peu plus tard et nous saurons donc précisément où la retrouver.

Ligne 6 : voici le branchement vers la routine de son, routine dont l'adresse a été communiquée au système par la ligne d'affectation

```
SON = $FFAB
```

L'ordinateur joue donc, pendant une durée extrêmement courte, la note la plus basse possible.

Ligne 7 : nous retrouvons, comme prévu, la valeur qu'avait l'accumulateur A avant l'appel de la routine. Cette façon de procéder est dictée par le fait que le registre est utilisé (à notre insu) dans le sous-programme d'émission de musique et que la valeur qu'il a en retour n'a rien de commun avec celle qu'il avait.

Ligne 8 : INCA ajoute 1 au contenu de l'accumulateur qui prend donc la valeur 2. Puisque notre souhait est de réaliser un bruit de sirène, il nous faudra à chaque fois augmenter la fréquence de la note qui sera jouée. C'est ce que nous venons de faire en incrémentant la valeur de A.

Ligne 9 : on retranche au registre A le nombre 0. Voici une instruction qui serait tout à fait inutile si elle n'allait nous servir à la ligne suivante.

Ligne 10 : le branchement à la ligne SUITE s'effectuera si la différence précédente a porté sur deux nombres non égaux ; ce qui est le cas, on a retranché 0 à 2.

On retrouve alors la ligne 4 qui va permettre cette fois de faire jouer par la machine la note de fréquence 2, note d'un ton légèrement au-dessus de la note déjà entendue. Puis — lignes 7 à 10 —, une unité sera ajoutée à A, et 0 sera retranché à la nouvelle valeur (3) de ce registre ; cette soustraction se faisant sur deux chiffres différents, un branchement sera à nouveau effectué à la ligne SUITE.

Le but de cette boucle est donc de faire entendre à chaque passage une note d'une fréquence à peine supérieure à la précédente et c'est pourquoi l'exécution du programme nous donne l'impression d'écouter une sirène.

Essayons, pour finir, de savoir à quel moment l'ordinateur sortira de la boucle. Le registre A étant augmenté de 1 à chaque fois, finira par arriver à 255 (ses huit bits à 1). L'incrémentation suivante ramènera tous ses bits à zéro et la soustraction de la ligne 9 portera alors sur deux nombres égaux. A ce moment-là, l'instruction BNE sera devenue sans effet et c'est l'ordre d'après, RTS, qui s'exécutera.

Alors, ce programme vous a-t-il donné des idées ? Vous allez accélérer le mouvement, n'est-ce pas ? Vous allez remplacer l'instruction INCA de la ligne 8 par ADDA #$5 ou ADDA #$10 ? Allez-y, mais n'ajoutez pas n'importe quel nombre au registre A, vous risqueriez de faire boucler indéfiniment votre programme !

INCB INX INS

INCB augmente d'une unité le contenu du registre B. INX et INS réalisent les mêmes opérations avec les registres doubles X et S. On ne peut utiliser ces instructions qu'avec le mode d'adressage inhérent.

INC

Avec INC, la possibilité est donnée d'incrémenter — ajouter 1 — au contenu d'un octet de la mémoire. L'adressage indexé et l'adressage étendu sont les deux modes possibles.

Exemple : MODE D'ADRESSAGE ÉTENDU (13 octets)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3   DEBUT        LDAB    $4B01      ; 19201 DECIMAL
4   SUITE        SUBB    #$1
5                BEQ     FIN
6                INC     $4B00      ; 19200 DECIMAL
7                BRA     SUITE
8   FIN          RTS
```

Programme BASIC

```
10 INPUT " PREMIER NOMBRE " ; N : POKE 19200 , N
20 INPUT " DEUXIEME NOMBRE " ; M : POKE 19201 , M + 1
30 EXEC 18944
40 PRINT " LA SOMME VAUT " ; PEEK (19200)
```

Commentaires

La lecture des lignes BASIC indique qu'il s'agit là d'un programme d'addition. L'instruction ADD n'apparaît pas dans la partie assembleur car elle a été remplacée par une boucle incrémentant le contenu d'un octet et ceci le nombre de fois voulu.

Ligne 3 : le contenu de l'octet 19201 est placé dans l'accumulateur B. Ce qui fait que B contient le nombre M + 1. Cette valeur est égale au second terme de la somme augmenté de 1.

Ligne 4 : on retranche 1 au registre B. Voici que s'explique la raison pour laquelle on est parti d'une valeur supérieure d'une unité pour B, ceci compense cela.

Ligne 5 : si la différence porte sur deux chiffres égaux, c'est-à-dire si B vaut 1, un branchement est effectué à la ligne 8, ligne dont l'étiquette est FIN.

Ligne 6 : on ajoute 1 au contenu de l'octet 19200, donc au premier terme de la somme.

Ligne 7 : BRA est l'instruction de branchement inconditionnel. Equivalente de GOTO, elle renvoie le processeur à la ligne 4 (SUITE) pour la suite du programme.

Nous nous retrouvons une fois encore devant une boucle qui, à chaque passage, procède aux deux opérations suivantes :

- 1 est enlevé au registre B, c'est-à-dire au deuxième nombre de la somme.
- 1 est ajouté à l'octet 19200, c'est-à-dire au premier terme de la somme.

B étant décrémenté à chaque fois, sa valeur arrivera forcément à 1. La ligne assembleur 4 (SUITE) effectuera alors une différence donnant un résultat nul, et le branchement sera réalisé. L'octet 19200 contiendra à la fin du programme la somme des deux nombres que nous avons proposés à l'ordinateur.

Reprenons rapidement ce que nous avons déjà dit à propos des additions sur huit bits (voir l'instruction ADDA) : lorsque le résultat d'une somme arrive à 256, il est ramené à 0.

Exemple : 200 + 100 = 256 + 44 = 44.

CLRA

Le registre A est mis à zéro, ce qui revient à dire que A est chargé avec la valeur 0. Il n'y a qu'une seule possibilité pour l'adressage : le mode inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (14 octets)

Programme assembleur

```
1           ORG     $4A00
2           EXC     DEBUT
3   DEBUT   CLRA
4           LDAB    #$0A        ; COMPTEUR A DIX
5           LDX     #$4B00      ; 19200 DECIMAL
6   SUITE   STAA    #$0,X
7           INX
8           SUBB    #$1
9           BNE     SUITE
10          RTS
```

Programme BASIC

```
10 FOR I = 19200 TO 19209 : POKE I , RND(10) : NEXT
20 EXEC 18944
30 FOR I = 19200 TO 19209 : PRINT PEEK(I) ; : NEXT
```

Commentaires

Le but de ce programme est d'écrire la valeur 0 dans chacun des 10 octets numérotés de 19200 à 19209. Le programme BASIC inscrit des valeurs quelconques dans ces octets, lance le programme machine et vérifie que le but a bien été atteint.

Ligne 3 : les explications concernant l'instruction CLRA tiennent en quelques mots : le chiffre 0 est écrit dans A et l'on aurait pu tout aussi bien, avec un octet de plus il est vrai, écrire LDAA #$0.

Ligne 4 : B est chargé avec 10. Le registre nous servira de compteur pour la boucle qui sera exécutée 10 fois.

Ligne 5 : X contient au début du programme la valeur 19200 et, de ce fait, X pointe le premier octet que nous aurons à rendre nul.

Ligne 6 : on range la valeur de A dans l'octet 19200. Comme A a été mis à zéro à la ligne 3, cet octet contient donc maintenant le nombre 0.

Ligne 7 : le registre X est incrémenté et pointe alors vers 19201.

Ligne 8 : on retranche 1 à B, celui-ci vaut alors 9.

Ligne 9 : puisque la soustraction a été effectuée entre deux valeurs différentes (10 et 1), l'instruction BNE branchera le programme à la ligne 6 (l'étiquette de cette ligne étant justement SUITE). L'octet 19201 sera alors mis à zéro, et le registre X, incrémenté, pointera vers 19202. La ligne 8 retranchera alors 1 de B qui, à ce moment-là, vaudra 8. Le test de branchement de la ligne 9 renverra à nouveau le programme en ligne 6.

Nous avons donc une boucle qui sera exécutée 10 fois. Après avoir rendu nul le dixième octet (19209), la ligne 8 effectuera la différence entre le contenu de B (qui vaudra alors 1) et le nombre 1. Puisque cette différence ne sera plus différente de zéro, la ligne 9 n'aura plus aucun effet et il ne restera plus qu'à retourner au programme BASIC.

CLRB

Cette instruction se programme de la même manière que CLRA. L'adressage inhérent est le seul accepté.

CLR

Le contenu de l'octet mémoire spécifié est rendu nul. On peut utiliser les modes d'adressage étendu et indexé.

Exemple : MODE D'ADRESSAGE ÉTENDU (27 octets)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3   DEBUT        JSR     $FBD4     ; CLS
4                CLR     $E8       ; SORTIE ECRAN
5                LDAA    #$40      ; ASCII 64
6                BSR     SPROG
7                BSR     SPROG
8                BSR     SPROG
9                BSR     SPROG
10  FIN          BRA     FIN
11               ;
12  SPROG        INCA               ; LETTRE SUIVANTE
13               INC     $3280      ; LIGNE SUIVANTE
14               CLR     $3281      ; COLONNE 0
15               JSR     AFCAR      ; AFFICHAGE
16               RTS
17               ;
18  AFCAR        =       $F9C6
```

Commentaires

Ligne 3 : nous connaissons l'adresse $FBD4 : c'est celle qui efface l'écran.

Ligne 4 : nous souhaitons faire apparaître les quatre premières lettres de l'alphabet en utilisant quatre fois de suite la routine d'affichage d'un caractère. Rappelons-nous qu'il faut mettre à zéro l'octet $E8.

Ligne 5 : il faut ensuite placer dans l'accumulateur le code ASCII de la lettre que l'on veut afficher.

Ligne 6 : un branchement vers la ligne 12 est réalisé. L'équivalent BASIC aurait été GOSUB 12.

Ligne 12 : on ajoute une unité au registre A. Ceci a pour effet de faire passer sa valeur à 65 (décimal). De ce fait, A contient le code ASCII de la première lettre de l'alphabet.

Ligne 13 : on commande l'incrémentation de l'octet $3280 ; on l'augmente donc de 1. Mais que pouvait-il contenir auparavant ? Eh bien tout simplement la valeur 0. En effet cet octet contient le numéro de la ligne sur laquelle l'affichage d'un caractère va avoir lieu. Or, où se trouve le curseur après l'exécution de l'instruction CLS ? Sur la ligne 0, bien sûr.

Ligne 14 : l'octet $3281 sera, dans tout le programme, forcé à 0 ; de cette manière, toutes les lettres seront visibles sur la première colonne de l'écran.

Ligne 15 : on lance l'ordinateur dans la routine $F9C6. Il y a donc, pour l'instant, la lettre A affichée au début de la ligne numérotée 1 (c'est-à-dire la deuxième).

Ligne 16 : BSR étant l'équivalent de GOSUB et RTS celui de RETURN, le microprocesseur retourne à l'instruction qui suit celle de laquelle il est parti...

Ligne 7 : ... pour être relancé dans le sous-programme SPROG ...

Lignes 12 à 16 : ... et retrouver une seconde fois la routine AFCAR. La lettre B (INCA) apparaîtra en dessous de la précédente : colonne 0 (CLR $3281) et ligne 2 (INC $3280).

Ligne 10 : quand le programme sera entièrement exécuté, on pourra voir sur notre écran les lettres A, B, C et D disposées les unes en dessous des autres.

DECA

Le contenu de l'accumulateur A est décrémenté, c'est-à-dire qu'une unité lui est retirée. Le mode d'adressage inhérent est le seul utilisable.

Exemple : MODE D'ADRESSAGE INHÉRENT (40 octets)

Programme assembleur

```
1          ORG     $4A00
2          EXC     DEBUT
3   DEBUT  JSR     $FBD4     ; CLS
4          LDD     #$4101    ; 65 DANS A et 1 DANS B
5          STD     $BF21     ; LETTRE A ALPHANUM.
6          LDAB    #$1E
7          STAB    $BF23     ; ROUGE/BLEU CLIGNOTANT
8          LDD     #$1000    ; 16 DANS A et 0 DANS B
9          STD     $BF26     ; LIGNE + COLONNE
10         LDD     #$1A01    ; 26 DANS A et 1 DANS B
11  SUITE  STAB    $BF28     ; EXECUTION
12         LDX     #$64
13  TEMPO  DEX
14         BNE     TEMPO
15         INC     $BF21     ; LETTRE SUIVANTE
16         DECA
17         BNE     SUITE
18  FIN    BRA     FIN
```

Commentaires

Lignes 4 et 5 : le contenu du double registre D est rangé dans les octets $BF21 et $BF22. Ces octets font partie du générateur vidéo, générateur que nous allons utiliser pour faire apparaître à l'écran les vingt-six lettres de l'alphabet. Pour l'instant, le premier de ces octets

(le n° 48929 en décimal) contient le code ASCII de la lettre A et le suivant (le n° 48930) la valeur correspondant à l'affichage de type alphanumérique.

Lignes 6 et 7 : l'écriture de $1E (30 décimal) dans l'octet $BF23 (48931) va provoquer le coloriage des lettres en rouge sur fond turquoise. Elles seront, de plus, clignotantes.

Lignes 8 et 9 : on indique au microprocesseur à quel endroit de l'écran il doit dessiner le premier caractère : vers le milieu et complètement à gauche (ligne n° 16 et colonne n° 0).

Ligne 10 : avant d'entrer dans la boucle SUITE, le premier accumulateur 8 bits est chargé avec le nombre de lettres de l'alphabet (26 donc, soit $1A hexa) et le second avec le nombre 1.

Ligne 11 : l'écriture de la valeur de B dans l'octet $BF28 provoque l'affichage de la première lettre de l'alphabet, en majuscule et sur le mode clignotant.

Lignes 12, 13 et 14 : avant de laisser l'ordinateur s'occuper du reste du programme, on l'oblige à perdre un peu de temps. Tout simplement parce que le système d'affichage d'un caractère sur l'écran possède une certaine inertie et que l'on risque, si l'on ne tempère pas le 6803, de voir celui-ci donner un ordre d'affichage à l'octet $BF28 alors que le précédent caractère n'est pas encore visible. Naturellement cet ordre ne sera alors d'aucune utilité et il manquera des lettres à notre alphabet. Voyons, pratiquement comment on doit s'y prendre pour faire, si l'on peut dire, *traîner le microprocesseur* : on écrit dans le registre X un nombre (100 décimal dans notre cas), on lui soustrait 1 (instruction DEX de la ligne 13), on regarde si l'on n'atteint pas 0 et on recommence la même opération 99 fois de suite. Au bout du compte, le registre X, à force d'être décrémenté, sera nul et le programme reprendra son cours normal.

Ligne 15 : 1 est ajouté à l'octet qui contient le code ASCII des caractères qui s'inscrivent sur le téléviseur. C'est donc la lettre B qui est cette fois concernée.

Lignes 16 et 17 : puisque A est décrémenté, sa valeur passe à 25 et l'instruction BNE rebranche le programme à la onzième ligne.

Quand la valeur du premier accumulateur sera arrivée à 0, la boucle SUITE aura été empruntée vingt-six fois, faisant apparaître à chaque

passage une nouvelle lettre de l'alphabet. Et toujours avec les mêmes caractéristiques : clignotement et couleurs rouge et bleu.

Ne terminons pas cette étude sans noter une particularité de l'octet $BF27 ; cet octet voit son contenu automatiquement augmenté d'une unité à chaque fois qu'un caractère est inscrit sur l'écran. C'est ce qui explique que nos lettres aient été écrites les unes à côté des autres, sans aucune intervention de notre part.

DECB DEX DES

Ces instructions procèdent avec les registres B, X et S de la même façon que DECA le fait avec A. Elles retranchent 1 à leur contenu. Le mode d'adressage utilisable est l'inhérent.

DEC

Cette instruction permet de retirer 1 de la valeur d'un octet en mémoire. On peut se servir des modes d'adressage indexé et étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (14 octets)

Programme assembleur

```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  LDAA   #$FF      ; 255 EN DECIMAL
4          STAA   $4B00     ; 19200 EN DECIMAL
5   SUITE  DECA
6          BNE    SUITE
7          DEC    $4B00
8          BNE    SUITE
9          RTS
```

Programme BASIC

```
10 PRINT @ 500 , " DEBUT " : SOUND 150 , 10 : EXEC 18944
20 PRINT @ 500 , "FIN " : SOUND 150 , 10
```

Commentaires

Voici le programme qui réalise une boucle de temporisation. Elle est équivalente en durée à la boucle BASIC :

```
FOR I = 1 TO 300 : NEXT
```

Mais si le BASIC n'a rien fait d'autre que de compter jusqu'à 300, l'assembleur, pendant la même durée, a eu le temps d'exécuter sa boucle vide plus de 60 000 fois. Ceci met en avant l'extraordinaire rapidité des programmes écrits en langage machine.

Lignes 3 et 4 : le registre A est chargé avec la valeur maximum et cette valeur est écrite dans l'octet 19200.

Lignes 5 et 6 : A valant 255, la décrémentation lui soustrait une unité et le fait passer à 254. On doit être habitué maintenant à l'instruction BNE qui va renvoyer le programme à la ligne 5. Ainsi le microprocesseur n'a effectué aucune action visible : il n'a fait que perdre du temps. La ligne 5 place dans A le nombre 253 et à nouveau le branchement BNE fait retourner à la ligne SUITE. On retrouve donc avec ces deux lignes, mais en beaucoup plus rapide, la ligne BASIC :

```
FOR I = 254 TO 0 STEP −1 : NEXT
```

Ligne 7 : l'accumulateur étant alors à zéro, c'est au tour de l'octet 19200 d'être décrémenté. Il avait été chargé au départ avec le nombre 255, il va donc contenir 254.

Ligne 8 : nouvelle utilisation de BNE qui concernera la dernière soustraction effectuée, en l'occurrence la décrémentation de la ligne 7. Puisque cette différence aura été faite entre les nombres différents 255 et 1, le programme se rebranchera à SUITE, donc 8 octets en arrière.

On retrouve alors la ligne 5. A sera une nouvelle fois décrémenté et, partant de 0, repassera à 255. N'oublions pas que −1 s'écrit pour le processeur 255 (ou FF hexadécimal). Nous voilà de nouveau dans une boucle qui va faire passer A de 255 à 0 (lignes 5 et 6), puis à terme, une décrémentation de l'octet 19200 (ligne 7) sera réalisée. Puisque cet octet en sera alors à la valeur 253, il y aura encore branchement à la ligne SUITE.

Le principe doit dès lors être compris : tant que le contenu de l'octet 19200 ne sera pas nul, le programme bouclera.

Vous avez certainement remarqué, en exécutant le programme, qu'à peine une demi-seconde s'écoulait entre les affichages des mots DEBUT et FIN sur l'écran. Essayez d'utiliser le principe de la boucle d'attente qui vient d'être étudié pour obtenir une temporisation plus grande. Un conseil : utilisez un nouvel octet que vous décrémenterez régulièrement à chaque fois que l'octet 19200 sera arrivé à zéro.

BHI BLO BHS BLS

Nous abordons maintenant quatre nouvelles instructions de branchement qui vont pouvoir, lorsque les conditions voulues seront réalisées, mettre une nouvelle valeur dans le registre PC et permettre ainsi d'annuler le déroulement séquentiel du programme. Ces instructions vont porter sur la comparaison des grandeurs de deux nombres, dont le premier sera toujours dans un registre. Rappelons que pour pouvoir utiliser BEQ et BNE, il fallait qu'une soustraction ou une comparaison ait été effectuée auparavant. Il va en être de même pour ces quatre nouvelles instructions.

BHI

Branch on Higher

Branchement si plus grand

BHI réalisera un branchement à l'un des octets du programme machine dans l'un des deux cas suivants :

1 — soustraction entre deux nombres dont le premier, celui contenu dans le registre, est strictement supérieur au deuxième.

2 — comparaison entre deux nombres dont le premier, là encore contenu dans le registre, est strictement supérieur au second.

L'octet suivant l'instruction BHI précisera, sur le mode complément à deux, quelle partie du programme devra alors être exécutée. Naturellement, cette instruction sera sans effet si le registre est inférieur ou égal à l'autre nombre.

A noter que la comparaison se fera sans que l'ordinateur tienne compte de la valeur en complément à deux de ces nombres. Si, par exemple, les deux nombres valent 254 et 10, le premier sera considéré comme supérieur au deuxième bien que ce soit en réalité le nombre −2 en complément à deux.

Si l'on veut trouver l'équivalent BASIC de BHI, on doit écrire :

```
IF A > B THEN ...
```

BLO

Branch on Lower

Branchement si plus petit

Cette fois le branchement s'effectuera dans l'un des cas suivants :

1 — soustraction entre deux nombres dont le premier est strictement inférieur au deuxième.

2 — comparaison entre deux nombres dont le premier est strictement inférieur au deuxième.

Le premier nombre étant toujours celui qui est dans le registre.

Pour cette instruction aussi, l'ordinateur ne tiendra pas compte des valeurs négatives, celles qui correspondent au complément à deux. Ecrivons la ligne BASIC correspondante :

```
IF A < B THEN ...
```

BHS

Branch on Higher or Same

Branchement si supérieur ou égal

Le branchement sera exécuté si l'une des deux conditions suivantes est vraie :

1 — soustraction entre deux nombres dont le premier est supérieur ou égal au deuxième.

2 — comparaison entre deux nombres dont le premier est supérieur ou égal au deuxième.

Là encore, d'une part le premier nombre se trouve dans le registre considéré, d'autre part on ne tient pas compte de la valeur en complément à deux. Voici comment, en BASIC, on écrirait cette instruction :

```
IF A > = B THEN ...
```

BLS

Branch on Lower or Same
Branchement si inférieur ou égal

Le branchement sera effectué si l'une des deux conditions suivantes est réalisée :

1 — soustraction entre deux nombres dont le premier est inférieur ou égal au deuxième.

2 — comparaison entre deux nombres dont le premier est inférieur ou égal au deuxième.

Cette dernière instruction respecte les mêmes règles que les précédentes : le premier terme de la différence ou de la comparaison provient du registre et le mode complément à deux n'est pas pris en compte. Donnons la traduction BASIC :

```
IF A < = B THEN ...
```


CMPA

Une comparaison sera réalisée avec le nombre placé immédiatement après cette instruction. Une instruction de branchement doit suivre normalement cette comparaison. Les modes d'adressage immédiat, indexé, direct et étendu peuvent être utilisés.

Exemple : MODE D'ADRESSAGE ÉTENDU (24 octets)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3   DEBUT        LDAA    $4B01     ; NOMBRE PROPOSE
4                CMPA    $4B00     ; NBRE TIRE AU SORT
5                BEQ     EGAL
6                BHI     SUP
7   INF          LDAA    #$3
8                BRA     FIN
9   EGAL         LDAA    #$1
10               BRA     FIN
11  SUP          LDAA    #$2
12  FIN          STAA    $4B02
13               RTS
```

Programme BASIC

```
10 X=RND(200) : POKE 19200 , X
20 INPUT " QUEL NOMBRE PROPOSEZ-VOUS " ; N : POKE 19201 , N
30 EXEC 18944 : ON PEEK (19202) GOTO 40 , 50 , 60
40 PRINT " VOUS AVEZ GAGNE " : END
50 PRINT " TROP GRAND " : GOTO 20
60 PRINT " TROP PETIT " : GOTO 20
```

Commentaires

Voici une version du jeu qui consiste à deviner un nombre que l'ordinateur aura choisi. Le branchement ON GOTO de la ligne BASIC 30 s'effectuera en fonction du nombre trouvé dans l'octet 19202. Voyons comment l'assembleur y aura placé la valeur correcte 1, 2 ou 3.

Ligne 3 : la ligne BASIC 20 aura écrit dans l'octet 19201 (4B01 hexa) le nombre N proposé. C'est donc le registre A qui va contenir ce nombre.

Ligne 4 : une comparaison est effectuée entre le nombre proposé et le contenu de l'octet 19200. Or dans cet octet, a été inscrit par POKE le nombre X que l'ordinateur a tiré au hasard. Voici donc la ligne qui va réaliser la comparaison sur laquelle est basée tout le programme.

Ligne 5 : si la comparaison a porté sur deux nombres égaux, cela voudra dire que l'on a gagné. BEQ va procéder alors à un branchement vers la ligne 9, quatre lignes plus loin. LDAA #$1 va, à ce moment-là, placer dans l'accumulateur le nombre 1, et un branchement inconditionnel (ligne 10) va entraîner le processeur à l'avant-dernière ligne du programme. Il ne restera plus alors qu'à écrire la valeur 1 dans l'octet 19202 ($4B02). Le BASIC retrouvera ce nombre et le branchement ON GOTO fera imprimer le message " VOUS AVEZ GAGNE ".

Ligne 6 : si l'on suppose maintenant que A est supérieur au nombre choisi par l'ordinateur, l'instruction BHI nous conduira à la ligne 11. Le registre A sera chargé avec la valeur 2, valeur qui sera ensuite écrite (ligne 12) dans l'octet 19202. Il sera alors à la charge du BASIC de retrouver le contenu de cet octet et le message " TROP GRAND " sera affiché sur l'écran.

Lignes 7 et 8 : dernière possibilité enfin, on a proposé à la machine un nombre trop petit. Les instructions BEQ et BHI sont restées sans effet et le programme s'est déroulé séquentiellement jusqu'à ces lignes. C'est le chiffre 3 qui sera écrit dans l'accumulateur avant qu'un branchement inconditionnel n'envoie le processeur à la ligne FIN. 3 est alors recopié dans l'octet 19202 et le BASIC donne la réponse à notre essai : " TROP PETIT ".

CMPB

D'une manière identique à CMPA, cette instruction compare le nombre écrit dans le registre B avec le nombre décrit immédiatement après. On peut se servir des modes d'adressage immédiat, indexé, étendu et direct.

CPX

CPX établit une comparaison entre deux nombres de 16 bits. Une instruction de branchement doit ensuite exploiter cette comparaison. On peut utiliser les modes d'adressage immédiat, direct, indexé et étendu.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (49 octets)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3   DEBUT        LDD     #$6361
4                STD     $BF21     ; C CEDILLE
5                LDD     #$0800
6                STD     $BF26     ; LIGNE 8 COL. 0
7                LDD     #$2871
8                STAB    $BF23     ; BLANC SUR ROUGE
9                LDX     #$0
10  CAR.         LDAB    #$1
11               STAB    $BF28     ; EXECUTION
12  TEMPO        LDAB    $BF20
13               CMPB    #$80
14               BHS     TEMPO
15               DECA
16               BNE     SUITE
17               LDAA    #$28
18               INC     $BF26     ; LIGNE SUIVANTE
19  SUITE        INX
20               CPX     #$3C0     ; 960 AFFICHAGES
21               BLO     CAR.
22  FIN          BRA     FIN
```

Commentaires

Lignes 3 et 4 : on écrit dans les octets 48929 et 48930 les valeurs 99 et 97 correspondant au caractère ç (voir Chapitre 2).

Lignes 5 et 6 : on détermine l'endroit de l'écran où apparaîtra le premier caractère : deuxième rangée (ligne n° 8) et colonne de gauche.

Lignes 7 et 8 : le registre A est chargé avec $28 (40 décimal) et le registre B avec la valeur $71. Cette valeur, quand elle se trouve dans l'octet $BF23, fait apparaître les lettres en blanc sur fond rouge.

Ligne 9 : le registre 16 bits X est mis à zéro. Il va contenir le nombre de passages dans la boucle CAR.

Lignes 10 à 14 : on donne, en écrivant 1 dans l'octet $BF28, l'ordre d'affichage de la lettre ç. On ne laisse pas le microprocesseur continuer la lecture du programme tant que l'affichage en question n'est pas complètement réalisé. Pour comprendre de quelle manière on s'y prend, il est nécessaire de savoir que l'octet $BF20 (48928) a son bit de gauche qui vaut 1 quand commence l'affichage et qui vaut 0 quand il se termine. L'ordinateur ne fait donc rien d'autre que d'attendre la mise à 0 de ce bit. Quand cela sera fait, le contenu de l'octet 48928 sera inférieur à $80 (128 décimal).

Lignes 15 à 18 : l'accumulateur A s'occupe du nombre de caractères qui sont affichés sur une ligne d'écran. Si ce nombre est inférieur à 40, rien n'est réalisé. Sinon on remet au niveau 40 le registre A, et on incrémente l'octet $BF26 pour que l'affichage ait lieu à la ligne suivante. Inutile de s'intéresser à l'octet $BF27 : il est automatiquement remis à 0 par la machine.

Lignes 19 à 21 : on ajoute 1 à X et on retourne à la ligne 10. A la fin du programme, quand l'instruction BLO (branchement si inférieur) ne sera plus d'aucun effet, 960 caractères seront visibles. La presque totalité de l'écran sera recouverte de ç.

ORAA

Un OU logique est effectué entre l'accumulateur A et le contenu d'un octet ou entre l'accumulateur et un nombre 8 bits. Les modes d'adressage permis sont l'immédiat, l'indexé, l'étendu et le direct.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (octets)

Programme assembleur

```
1            ORG    $4A00
2            EXC    DEBUT
3   DEBUT    LDX    #$4B00    ; 19200 DECIMAL
4            LDAA   $0,X
5            ORAA   #$1
6            CMPA   $0,X
7            BNE    PAIR
8   IMPAIR   LDAA   #$1
9            BRA    FIN
10  PAIR     LDAA   #$2
11  FIN      STAA   $1,X
12           RTS
```

Programme BASIC

```
10 INPUT " DONNEZ UN NOMBRE " ; N
20 POKE 19200 , N : EXEC 18944
30 ON PEEK (19201) GOTO 50 , 40
40 PRINT " CE NOMBRE EST PAIR " : GOTO 10
50 PRINT " CE NOMBRE EST IMPAIR " : GOTO 10
```

Commentaires

Réservons quelques lignes pour revoir de quelle façon s'exécute un OU logique entre deux nombres.

```
   1100
OU 1010
---------
=  1110
```

Notre programme, pour sa part, va effectuer un OU entre le contenu du registre A et le nombre 1. Puisque 1 s'écrit en binaire 00000001, seul le dernier bit de A sera concerné. Ainsi, si A se termine par 0, il se verra modifié car son dernier chiffre passera à 1. Par contre, si son dernier chiffre vaut 1, A gardera la même valeur.

Lignes 3 et 4 : le nombre que l'on a proposé à l'ordinateur a été rentré par POKE dans l'octet 19200 et c'est donc le registre A qui est chargé avec cette valeur.

Ligne 5 : le OU logique est réalisé entre le nombre que nous avons choisi et l'unité. Si ce nombre est pair, son écriture binaire se fera avec un 0 à la fin et, s'il est impair, il se terminera par 1. L'action de ORAA va donc consister à modifier la valeur de notre registre uniquement dans le cas où il est pair.

Lignes 6 et 7 : comparaison est faite entre les contenus de l'accumulateur et de l'octet 19200, octet qui, ne l'oublions pas, contient le nombre que nous avons tapé au clavier. Dans le cas où ce nombre est pair, ORAA l'a transformé et un branchement à la ligne 10 est effectué.

Lignes 8 et 9 : s'il s'agit d'un nombre impair, la valeur 1 est mise dans A pour être réécrite (ligne 11) dans l'octet 19201.

Ligne 10 : sinon, c'est le nombre 2 qui va transiter par l'accumulateur pour être placé ensuite dans ce même octet.

La ligne BASIC 30 va alors examiner cet octet et le branchement ON GOTO enverra à ce moment l'ordinateur à la bonne instruction.

ORAB

Un OU logique est réalisé entre le registre B et une valeur 8 bits. On peut employer les modes d'adressage immédiat, indexé, étendu et direct.

ANDA

ANDA réalise un ET logique entre l'accumulateur A et une donnée 8 bits. On peut utiliser les modes d'adressage immédiat, direct, indexé et étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (34 octets)

Programme assembleur

```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  LDD    #$4111
4          STD    $BF21     ; LETTRE A SOULIGNEE
5          LDD    #$1414
6          STD    $BF26     ; LIGNE 20 COL. 20
7          LDD    #$1701
8          STAA   $BF23     ; ROUGE/BLANC
9          STAB   $BF28     ; EXECUTION
10         LDAA   #$EF      ; 11101111 BINAIRE
11         ANDA   $BF22
12         STAA   $BF22
13         STAB   $BF28     ; EXECUTION
14  FIN    BRA    FIN
```

Commentaires

Lignes 3 et 4 : le contenu du registre A est écrit dans l'octet 48929 et celui du registre B dans l'octet suivant. Puisque $41 et $11 valent respectivement 65 et 17 en décimal, c'est la lettre majuscule A, soulignée, qui va apparaître sur l'écran.

Lignes 5 et 6 : les octets 48934 et 48935 contiennent les coordonnées du caractère : (20 , 20). L'affichage se situera donc au milieu de l'écran.

Lignes 7, 8 et 9 : on définit la couleur de la case (rouge sur blanc) et on donne l'ordre d'affichage. La lettre A, soulignée, est alors visible devant nos yeux.

Lignes 10 et 11 : un ET logique est effectué entre le contenu de l'octet $BF22 et le nombre $EF. Regardons cela de plus près :

0	0	0	1	0	0	0	1

octet $BF22 (48930)

1	1	1	0	1	1	1	1

accumulateur A : $EF (239)

Après l'instruction ANDA, voilà ce que contient A :

0	0	0	0	0	0	0	1

La nouvelle valeur de A est égale à 1 ; ANDA nous a permis de forcer à 0 le bit 4 de l'octet $BF22 et cela sans modifier les autres bits.

Lignes 12 et 13 : puisque 1 correspond au type de l'affichage alphanumérique normal, le caractère qui apparaît maintenant sur l'écran est identique au précédent à ceci près qu'il n'est plus souligné.

ANDB

Le registre B et un nombre 8 bits font l'objet d'un ET logique. Les modes d'adressage immédiat, direct, indexé et étendu peuvent être utilisés.

EORA

Comme les deux instructions précédemment étudiées, EORA réalise une opération logique : un OU exclusif est effectué entre le registre A et un nombre 8 bits. On peut, là aussi, utiliser les modes d'adressage immédiat, direct, indexé et étendu.

Exemple : MODE D'ADRESSAGE IMMÉDIAT (20 octets)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3     DEBUT      JSR     $FBD4     ; CLS
4                CLR     $E8
5                LDAA    #$42      ; ASCII B
6                LDAB    #$F0      ; 240 PASSES
7     SUITE      JSR     AFCAR
8                EORA    #$1       ; B PUIS C
9                DECB
10               BNE     SUITE
11    FIN        BRA     FIN
12               ;
13    AFCAR      =       $F9C6
```

Commentaires

L'utilisation du OU exclusif est faite ici dans le but d'inverser le dernier chiffre d'un nombre binaire.

```
   1          0
EOR 1      EOR 1
---        ---
=  0       =  1
```

Quand un OU exclusif est effectué entre un chiffre binaire et 1, le chiffre change de valeur.

Lignes 3 à 6 : après avoir effacé l'écran, on se prépare à utiliser la routine d'affichage d'un caractère. On annule le contenu de l'octet $E8 et on place dans le registre A le code ASCII du caractère que l'on veut faire apparaître, en l'occurrence la lettre B majuscule.

Ligne 7 : l'appel de la routine baptisée AFCAR est réalisé et la lettre B s'affiche en haut et à gauche de l'écran. Notons que cette routine positionne d'elle-même le curseur immédiatement à droite du caractère qu'elle vient d'écrire et que c'est donc là que la lettre suivante apparaîtra.

Ligne 8 : voici notre nouvelle instruction : elle réalise un OU exclusif entre le contenu du premier accumulateur et le chiffre 1. Analysons ce que cela va donner :

```
    1000010 <------ nombre hexadécimal 42
EOR 0000001
-----------
=   1000011
```

A contient donc maintenant le nombre $43 c'est-à-dire le code ASCII de la lettre C.

Lignes 9 et 10 : le registre B passe de 240 à 239 et l'ordinateur se rebranche à la ligne 7, pour afficher cette fois le caractère C sur l'écran. Puis, à nouveau un OU exclusif se fait entre les nombres $43 et 1, ce qui redonne le nombre $42 (nous vous laissons le soin de le vérifier).

En définitive, lorsque la boucle SUITE aura été parcourue 240 fois, six lignes alternant les lettres B et C seront visibles sur l'écran.

EORB

L'opération logique OU exclusif est effectuée entre le registre B et un nombre 8 bits. Sont permis les modes d'adressage immédiat, direct, indexé et étendu.

PSHA PULA

Ces deux instructions permettent l'empilement et le dépilement du registre A dans la pile système. Elles n'autorisent que le mode d'adressage inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (24 octets)

Programme assembleur

```
1            ORG    $4A00
2            EXC    DEBUT
3   DEBUT    LDAA   #$5A
4            STAA   $BF21     ; LETTRE Z
5            LDD    #$015F
6            STD    $BF22     ; MAUVE/BLANC + CLIGNO.
7            PSHA
8            LDD    #$1F27
9            STD    $BF26     ; COORDONNEES
10           PULA
11           STAA   $BF28     ; EXECUTION
12  FIN      BRA    FIN
```

Commentaires

Lignes 3 et 4 : l'écriture de $5A dans l'octet $BF21 va provoquer l'affichage de la dernière lettre de l'alphabet. En effet 90 (traduction décimale de $5A) est le code ASCII de la lettre Z.

Lignes 5 et 6 : le contenu de l'accumulateur A (1) est rangé dans l'octet $BF22, et ceci pour indiquer au système qu'il va s'agir d'un affichage de type alphanumérique normal. En même temps, la valeur $5F, provenant du registre B, est recopiée dans l'octet $BF23. Le choix des couleurs est donc réalisé : la lettre apparaîtra en mauve sur fond blanc, et en clignotant.

Ligne 7 : nous voici devant notre nouvelle instruction. Le contenu du registre A, le nombre 1 donc, est mis de côté dans la pile S ; on dit qu'il est sauvegardé. Nous procédons ainsi car la valeur de A va être perdue à la ligne suivante.

Lignes 8 et 9 : les registres 8 bits sont chargés avec les coordonnées du point de l'écran où doit avoir lieu l'affichage. En l'occurrence, il s'agit de la case située en bas et à droite, la dernière de l'écran donc.

Lignes 10 et 11 : l'instruction PULA ressort de la pile la valeur qui y avait été placée et va l'écrire dans A. Ce registre retrouve, à ce moment-là, la valeur 1 ; valeur qui nous sert alors à déclencher l'affichage effectif de notre caractère.

Résumons le rôle du tandem PSH-PUL : ces instructions permettent de mettre de côté la valeur d'un registre, d'utiliser ce registre pour remplir d'autres tâches, puis de replacer dans ce registre sa valeur initiale.

PSHB PULB PSHX PULX

Ces couples d'instructions permettent d'empiler et de dépiler les registres B et X. On ne les utilise que sur le mode inhérent.

LSRA

Abréviation de Logical Shift Right, *cette instruction décale tous les bits de l'accumulateur A vers la droite. Le bit de gauche est mis à zéro. On ne peut utiliser que le mode d'adressage inhérent.*

Exemple : MODE D'ADRESSAGE INHÉRENT (8 octets)

Programme assembleur

```
1            ORG     $4A00
2            EXC     DEBUT
3   DEBUT    LDAA    $4B00     ; 19200 DECIMAL
4            LSRA
5            STAA    $4B01     ; 19201 DECIMAL
6            RTS
```

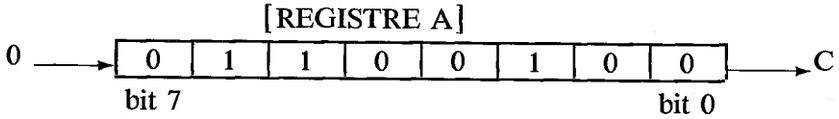
Programme BASIC

```
10 PRINT " DONNEZ UN NOMBRE INFERIEUR A 256 " ;
20 INPUT N : POKE 19200 , N : EXEC 18944
30 PRINT " LE QUOTIENT ENTIER DU NOMBRE PAR 2 " ;
40 PRINT " VAUT " ; PEEK (19201) : GOTO 10
```

Commentaires

Ce programme effectue en assembleur la division entière d'un nombre par deux. Voyons au niveau du binaire, comment cela se passe.

Considérons le nombre décimal 100 qui s'écrit en binaire 01100100.



Faisons subir aux chiffres qui constituent ce nombre un décalage sur la droite. Chaque chiffre va se retrouver dans le bit de rang immédiatement inférieur : le chiffre du bit 7, 0, va passer dans le bit 6, le chiffre 1 du bit 6 va s'écrire dans le bit 5, etc. Le dernier chiffre à droite (bit 0) va donc sortir de l'octet et sera perdu pour nous. L'ordinateur, lui, en gardera la trace en le mettant dans un endroit spécial

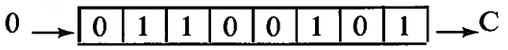
que l'on appelle l'indicateur de retenue et que l'on note C. Cet indicateur prendra donc la valeur 0, mais, répétons-le, ceci n'a aucune espèce d'importance pour notre exemple.

Sachant que LSRA remplace toujours le bit 7 par 0, voici ce que l'on obtient alors pour le registre A :

0	0	1	1	0	0	1	0

La traduction en décimal de cette valeur donne 50 ; on a donc bien divisé le nombre par 2.

Et si nous étions partis d'un nombre impair ? Essayons avec 101.



Quand LSRA aura agi, on obtiendra :

0	0	1	1	0	0	1	0

c'est-à-dire 50, ce qui est bien le quotient entier de 101 par 2. Dans ce deuxième cas, l'indicateur de retenue est passé à 1.

Il ne reste plus qu'à comprendre pourquoi le décalage à droite des chiffres a conduit à une division par deux. Prenons par exemple le chiffre 1 du bit 6 et voyons ce qu'il devient : il valait 2^6, c'est-à-dire 64 avant LSRA, il vaut 2^5, soit 32, après ; il a donc été réduit de moitié. Ce même raisonnement se fait pour tous les autres chiffres, ce qui nous donne l'explication voulue.

Les différentes lignes ne seront pas étudiées une à une, le programme assembleur se comprenant sans difficulté.

LSRB LSRD

LSRB : Un décalage de tous les bits du registre B est effectué sur la droite. Le bit 7 s'annule et le bit 0 est placé dans l'indicateur de retenue. C'est le mode d'adressage inhérent qui est employé.

LSRD : même chose avec le registre D mais c'est le bit 15 qui s'annule.

LSR

Comme pour LSRA, c'est un décalage sur la droite, mais ce décalage concernera le contenu d'un octet mémoire. Les modes d'adressage indexé et étendu sont permis.

Exemple : MODE D'ADRESSAGE ÉTENDU (41 octets)

Programme assembleur

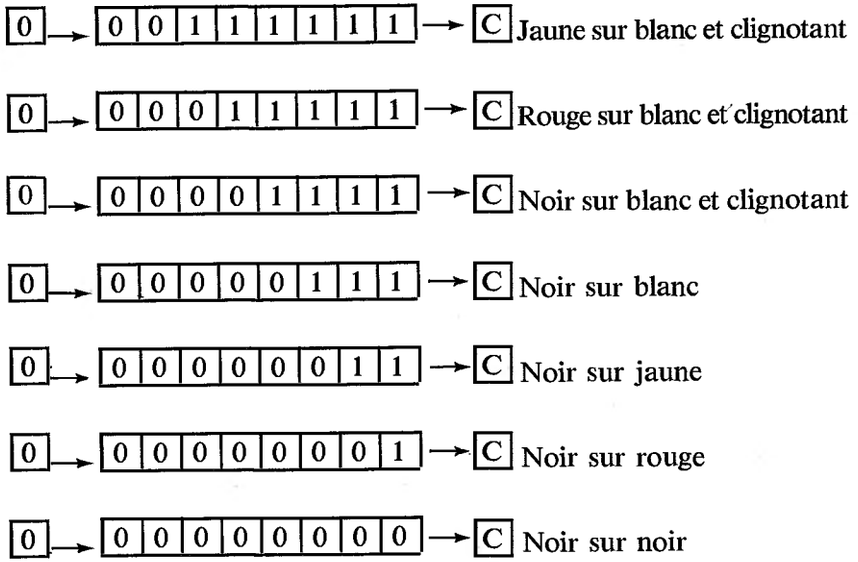
```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  JSR    $FBD4    ; CLS
4          LDD    #$4101
5          STD    $BF21    ; LETTRE A
6          LDD    #$1414
7          STD    $BF26    ; COORDONNEES
8          LDD    #$073F   ; B : 00111111 BINAIRE
9          STAB   $BF23
10  SUITE  LDAB   #$1
11         STAB   $BF28    ; EXECUTION
12  TEMPO  LDAB   $BF20
13         CMPB   #$80
14         BHS    TEMPO
15         LSR    $BF23    ; COULEURS
16         DECA
17         BNE    SUITE
18  FIN    BRA    FIN
```

Commentaires

Lignes 3 à 7 : on efface l'écran et on écrit dans les octets 48929, 48930, 48934 et 48935 les nombres décimaux 65 , 1 , 20 et 20. Nous programmons ainsi l'affichage de la lettre A à peu près au milieu de l'écran.

Lignes 8 et 9 : l'accumulateur A est chargé avec le chiffre 7. Ce registre décomptera le nombre de passages dans la boucle SUITE. Puis la valeur de B est transmise à l'octet $BF23 (48931 décimal).

Lignes 10 à 17 : l'affichage du caractère A est réalisé 7 fois de suite par écriture du nombre 1 dans l'octet $BF28. A chaque lecture de la boucle SUITE, les couleurs de la lettre changent du fait des modifications qui interviennent sur l'octet $BF23 (ligne 15). Voici les transformations successives que LSR fait subir à cet octet :



LSLA

C'est cette fois-ci d'un décalage vers la gauche qu'il est question, LSL étant l'abréviation de Logical Shift Left. *Le bit 7 passe dans l'indicateur de retenue et le bit 0 du registre A est mis à zéro. LSLA est utilisée avec le seul mode d'adressage inhérent.*

Exemple : MODE D'ADRESSAGE INHÉRENT (8 octets)

Programme assembleur

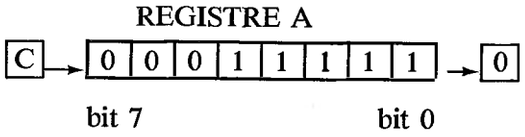
```
1                ORG     $4A00
2                EXC     DEBUT
3     DEBUT      LDAA    $4B00     ; 19200 DECIMAL
4                LSLA
5                STAA    $4B00
6                RTS
```

Programme BASIC

```
10 INPUT " DONNEZ UN NOMBRE " ; N : POKE 19200 , N
20 J = 1 : FOR I = 1 TO 3
30 J = J * 2 : EXEC 18944
40 PRINT " LE PRODUIT DU NOMBRE PAR " ; J ;
50 PRINT " VAUT " ; PEEK (19200)
60 NEXT : GOTO 10
```

Commentaires

On suppose que l'on propose à l'ordinateur le nombre 31 et on regarde ce qu'il devient quand on exécute l'instruction LSLA. 31 a pour équivalent binaire 00011111.



Le décalage va faire sortir du registre le contenu du bit 7 qui ira se placer dans l'indicateur de retenue, indicateur dont l'importance est nulle en l'état d'avancement de nos connaissances. Tous les chiffres étant translatés, on obtient pour A :

0	0	1	1	1	1	1	0

Le nombre 0 est venu prendre la place laissée libre dans le bit 0. En transcrivant le résultat obtenu en décimal, on obtient le nombre 62, c'est-à-dire le double de 31. Ainsi ce décalage à gauche de tous les bits a permis d'effectuer le produit par 2 du nombre qui se trouvait dans l'accumulateur. Ceci peut se comprendre puisque, en définitive, chaque chiffre se retrouvera avec une puissance de 2 supérieure d'une unité à la précédente.

Revenons à notre programme : le nombre que l'on a donné au départ à l'ordinateur est placé dans l'octet 19200 et, au premier passage de la boucle FOR NEXT, ce nombre est multiplié par 2. La ligne assembleur 5 replace la réponse dans ce même octet 19200. Au deuxième passage, le nombre est à nouveau multiplié par 2, ce qui fait que la valeur du début est, cette fois, multipliée par 4 ; elle le sera par 8 quand le programme BASIC sera terminé.

Bien entendu, ce programme donne des réponses cohérentes tant que l'on ne choisit pas un nombre supérieur ou égal à 32 (soit 00100000 en binaire). A partir de cette valeur, en effet, c'est le chiffre 1 qui est perdu dans les décalages rendant le résultat final incorrect (mais explicable).

LSLB LSLD

LSLB : Tous les bits du registre B sont décalés sur la gauche. Le bit de droite s'annule et celui de gauche passe dans l'indicateur de retenue. On emploie le mode d'adressage inhérent seulement.

LSLD : même action mais concernant le registre double D.

LSL

Un décalage d'un bit sur la gauche du contenu d'un octet mémoire est effectué. Le bit 0 passe à 0 et le bit 7 est écrit dans l'indicateur de retenue. Cette instruction s'exécute avec les modes d'adressage indexé et étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (32 octets)

Programme assembleur

```
 1          ORG    $4A00
 2          EXC    DEBUT
 3  DEBUT   JSR    $FBD4    ; CLS
 4          LDX    #$0A     ; 10 LIGNES
 5          LDAA   #$41     ; ASCII 65
 6          CLR    $E8
 7  SUIT2   LDAB   #$5
 8  SUIT1   LSL    $3281    ; COLONNE
 9          JSR    $F9C6    ; AFCAR
10          DECB
11          BNE    SUIT1
12          CLR    $3281    ; COLONNE
13          INC    $3280    ; LIGNE
14          DEX
15          BNE    SUIT2
16  FIN     BRA    FIN
```

Commentaires

Ligne 3 : un saut vers la routine d'effacement de l'écran est programmé. Le curseur est alors positionné automatiquement sur la première case de l'écran, ce qui nous permet d'en déduire que les octets $3280 et $3281 ont un contenu nul.

Lignes 4 et 5 : le registre X est chargé avec un nombre qui va correspondre, nous le verrons, aux dix premières lignes de l'écran. A, pour sa part, est chargé avec le code ASCII de la lettre majuscule A ; son contenu ne sera à aucun moment modifié au cours du programme.

Ligne 7 : cinq lettres vont apparaître successivement sur chaque ligne, et c'est B qui tiendra ce compte.

Lignes 8 à 11 : la boucle SUIT1 est parcourue 5 fois. A chaque passage, le contenu de l'octet $3281 subit un décalage sur la gauche, puis la routine d'affichage de la lettre A est mise à contribution. Voyons cela de près :

1. Premier passage :

- l'octet $3281 s'écrit 00000000 ;
- il subit un décalage sur la gauche et le chiffre 0 entre dans le bit 0 ; ainsi il garde la valeur 0 ;
- la routine AFCAR dessine la lettre A dans la colonne 0 mais, en même temps, ajoute une unité à l'octet qui nous intéresse. Ceci n'a rien de mystérieux, le curseur est toujours déplacé d'un cran à droite quand un caractère est affiché, non ?

2. Deuxième passage :

- l'octet $3281 s'écrit 00000001 ;
- LSL agit sur lui et le transforme en 00000010 ;
- la routine affiche la lettre A sur la colonne numéro 2 et incrémente le contenu de notre octet.

3, 4 et 5. En poursuivant les calculs, on arrive à comprendre pourquoi le caractère A est apparu dans les colonnes 0, 2, 6, 14 et 30.

Lignes 12 et 13 : l'octet $3280 prend la valeur 1 indiquant par là même que c'est la deuxième ligne de l'écran qui va être concernée. L'octet suivant, quant à lui, est ramené à zéro pour que l'affichage reparte à nouveau de la première colonne.

Lignes 14 et 15 : le programme bouclera tant que dix lignes identiques à la première n'auront pas été écrites.

ROLA

Tous les bits de l'accumulateur subissent une rotation vers la gauche. Le bit 7 passe dans l'indicateur de retenue et la valeur préalablement contenue par celui-ci est transférée dans le bit 0. ROLA est l'abréviation de ROtate Left A. *Seul le mode d'adressage inhérent est autorisé.*

Exemple : MODE D'ADRESSAGE INHÉRENT (15 octets)

Programme assembleur

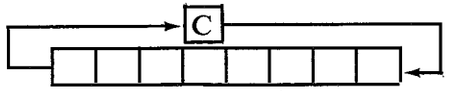
```
1            ORG    $4A00
2            EXC    DEBUT
3   DEBUT    CLRA
4            CLRB
5            ROLA
6            LDAA   $4B00     ; 19200 DECIMAL
7            ROLA
8            ROLB
9            STAB   $4B01
10           STAA   $4B02
11           RTS
```

Programme BASIC

```
10 INPUT " DONNEZ UN NBRE " ; N : POKE 19200 , N
20 EXEC 18944 : PRINT " SON DOUBLE VAUT : " ;
30 PRINT 256*PEEK(19201)+PEEK(19202) : GOTO 10
```

Commentaires

Vous vous souvenez de l'instruction LSLA ? Elle nous avait permis de multiplier un nombre par 2, 4 ou 8 mais cela n'était pas allé sans un ennui de taille : les chiffres 1 qui sortaient sur la gauche de l'accumulateur étaient perdus et, si l'on partait d'un nombre trop grand, la réponse n'était pas celle attendue. Regardons comment nous allons pouvoir y remédier avec notre nouvelle instruction ROLA.



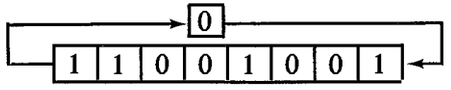
REGISTRE A

Tous les bits de l'accumulateur subissent un décalage sur la gauche ; le bit contenu dans l'indicateur de retenue passe dans le bit 0 et c'est le bit 7 qui prend sa place. Il s'agit donc là d'une rotation réalisée sur 9 bits.

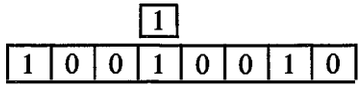
Lignes 3 et 4 : les deux registres 8 bits sont mis à zéro.

Ligne 5 : on fait subir à A une rotation ; puisque A s'écrit 00000000 en binaire, ceci n'a pas d'autre effet que de faire entrer le chiffre 0 dans l'indicateur de retenue.

Lignes 6 et 7 : on recopie dans A le nombre que nous avons écrit par POKE dans l'octet 19200 et, grâce à ROLA, on le multiplie par 2. Examinons cela de plus près et supposons, pour fixer les idées, que N ait été choisi égal à 201 (soit 11001001).



C est à zéro (ligne 5) et on obtient donc après ROLA :



Le bit C est passé à 1 et la nouvelle valeur de l'accumulateur est, en décimal, 146. Ceci n'est naturellement pas le double de 201, mais attendons la suite.

Lignes 8 et 9 : ROLB a pour effet de décaler les huit chiffres 0 du registre B et de faire rentrer sur sa droite le bit qui se trouvait dans l'indicateur, c'est-à-dire le bit 1. La nouvelle valeur de B est donc 1 ; elle est inscrite alors dans l'octet 19201.

Lignes 10 et 11 : le décimal 146 est, pour sa part, placé à l'adresse 19202 et le retour au BASIC est programmé. On va pouvoir vérifier la logique du programme assembleur :

```
PRINT 256*PEEK(19201) + PEEK(19202)
```


Réponse : 256*1 + 146 = 402.

Tout s'est donc passé en définitive comme si nous avions fait un décalage sur 9 bits.

011001001 (201 décimal) serait devenu 110010010 (402 décimal).

ROLB

Cette instruction agit sur B de la même façon que ROLA le fait sur A. C'est le mode d'adressage inhérent qui doit être utilisé.

ROL

Tous les bits de l'octet mémoire spécifié sont décalés d'une position sur la gauche. Le bit 7 est placé dans l'indicateur de retenue et la valeur d'origine de celui-ci est transférée dans le bit 0. Les modes d'adressage possibles sont l'indexé et l'étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (33 octets)

Programme assembleur

```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  JSR    $FBD4    ; CLS
4          LDD    #$4101
5          STD    $BF21    ; LETTRE A
6          LDD    #$1414
7          STD    $BF26    ; COORDONNEES
8          LDD    #$0501
9          STAA   $BF23    ; NOIR/MAUVE
10         STAB   $BF28    ; EXECUTION
11         ROLA
12         ROL    $BF23    NOIR/VERT + CLIGNO.
13         STAB   $BF28    ; EXECUTION
14  FIN    BRA    FIN
```

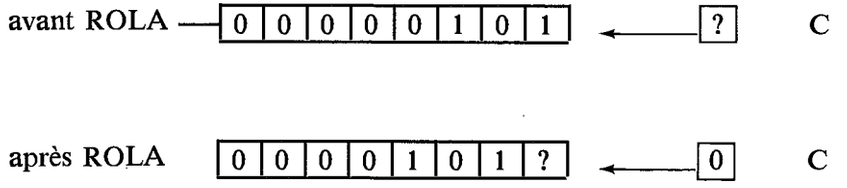
Commentaires

Lignes 3 à 7 : l'effacement de l'écran est réalisé par l'appel de la routine $FBD4. Puis on indique à l'ordinateur que l'on va utiliser le générateur vidéo pour procéder à l'affichage d'une lettre en couleur et au milieu de l'écran.

Ligne 8 : les nombres 5 et 1 sont écrits dans les accumulateurs 8 bits. La configuration binaire du registre A en est la suivante : 00000101.

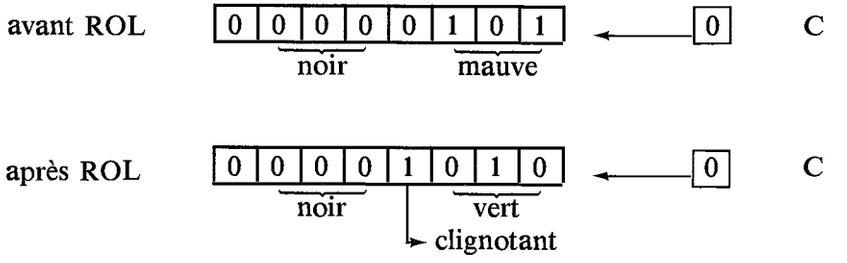
Lignes 9 et 10 : le contenu de A est rangé dans l'octet qui détermine la couleur, et l'affichage de notre caractère est réalisé : la lettre A apparaît en noir sur mauve.

Ligne 11 : on fait subir au registre A une rotation sur la gauche.



A cet instant, on ne peut savoir la valeur exacte du registre : son dernier bit est inconnu. Par contre, ce dont on est sûr, c'est que l'indicateur de retenue contient le chiffre 0. Ceci va nous être utile à la ligne suivante.

Ligne 12 : c'est cette fois sur l'octet $BF23 qu'une rotation est effectuée :



Ligne 13 : inutile de s'expliquer plus longtemps. La seconde lettre va être visible en noir sur fond vert ; de plus, elle va clignoter.

RORA

Cette instruction effectue une rotation sur la droite de tous les bits de l'accumulateur A. Le bit de retenue prend la place du bit 7 ; il est lui-même remplacé par le bit 0. On utilise le mode d'adressage inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (26 octets)

Programme assembleur

```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  JSR    $FBD4     ; CLS
4          LDX    #$190     ; 400 DECIMAL
5          LDAA   #$A6      ; 134 + 32
6          CLR    $E8
7   SUITE  JSR    $F9C6     ; AFCAR
8          CLC
9          RORA
10         JSR    $F9C6     ; AFCAR
11         CLC
12         ROLA
13         DEX
14         BNE    SUITE
15  FIN    BRA    FIN
```

Commentaires

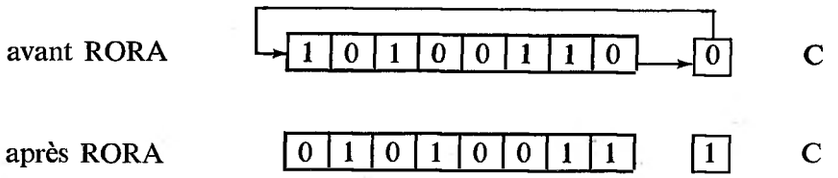
Ligne 3 : la routine d'effacement de l'écran est appelée et de ce fait le curseur est positionné en haut et tout à gauche de l'écran.

Ligne 4 : le registre X est chargé avec le nombre $190. Ce nombre correspond aux 400 passages que le programme effectuera dans la boucle SUITE.

Ligne 5 : A contient la valeur $A6 c'est-à-dire 166 décimal ou 10100110 binaire.

Ligne 7 : le microprocesseur est lancé dans la routine d'affichage. Le caractère qui va apparaître sur la première case de l'écran, ayant pour code 166, a les particularités suivantes : c'est un caractère graphique (le numéro 134) et sa couleur est bleue (166 = 134 + 32).

Lignes 8 et 9 : nous voulons faire exécuter une rotation sur la droite au contenu du registre A. Nous ne pouvons connaître à priori la valeur du bit de retenue C et pourtant nous avons besoin pour notre programme qu'il soit nul. Voilà donc à quoi sert l'instruction CLC : elle force à 0 l'indicateur de retenue. La commande qui a un effet opposé est SEC ; elle fait entrer le chiffre 1 dans C.



Ligne 10 : nouvel appel de la routine $F9C6. Et cette fois-ci, c'est la lettre S (code décimal 83 ou binaire 1010011) que l'on pourra voir.

Lignes 11 et 12 : après RORA, voici ROLA ; le registre A, voyant ses bits décalés sur la gauche, va retrouver sa configuration initiale (10100110 binaire ou 166 décimal). Remarquons que la précaution de remettre le chiffre 0 dans l'indicateur de retenue a été prise : ceci s'explique par le fait que l'ordinateur intervient sur cet indicateur dans la routine d'affichage et qu'il risque donc de nous retourner une valeur modifiée.

Lignes 13 et 14 : le registre X, décrémenté, voit sa valeur passer à 399 et le programme se relance dans l'exécution de la boucle SUITE. Le caractère graphique 166 puis la lettre S vont être une seconde fois dessinés.

Faisons le bilan : quand l'ordinateur sortira du programme, vingt lignes de quarante caractères (alternant graphiques et lettres S) seront visibles sur notre téléviseur.

RORB

Cette instruction permet de faire sur le registre B le même genre d'opération que RORA. On utilise le mode d'adressage inhérent.

ROR

Une rotation sur la droite du contenu d'un octet est réalisée. Le bit 0 prend la place du bit de retenue qui, lui-même, se retrouve à l'emplacement du bit 7. Il est permis d'utiliser les modes d'adressage indexé et étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (9 octets)

Programme assembleur

```
1          ORG     $4A00
2          EXC     DEBUT
3   DEBUT  CLRA
4          RORA
5          ROR     $4B00     ; 19200 DECIMAL
6          ROR     $4B01     ; 19201 DECIMAL
7          RTS
```

Programme BASIC

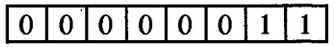
```
10 PRINT " DONNEZ UN NBRE INFERIEUR A 65536 "
20 INPUT N : POKE 19200 , INT(N/256)
30 POKE 19201 , N - INT(N/256)*256
40 J=1 : FOR I = 1 TO 4
50 EXEC 18944 : J = J * 2
60 PRINT " LE QUOTIEN DU NBRE PAR " ; J ;
70 PRINT " VAUT " ; 256 * PEEK(19200) + PEEK(19201)
80 NEXT : GOTO10
```

Commentaires

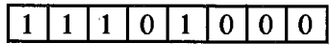
Ce programme exécute les divisions entières par 2, 4, 8 et 16 de n'importe quel nombre plus petit que 65536. Nous retrouvons donc une méthode déjà connue, le décalage sur la droite, mais nous allons l'appliquer ici, grâce à l'utilisation du bit de retenue, à une valeur écrite sur deux octets.

Prenons par exemple le nombre 1000 qui s'écrit en binaire 1111101000. Le programme BASIC le décompose en deux valeurs de

huit bits qui sont écrites dans l'octet 19200 pour le poids fort et dans l'octet 19201 pour le reste.



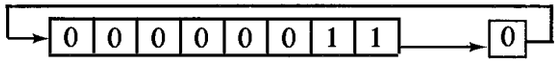
octet 19200



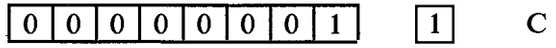
octet 19201

On procède maintenant de la même façon que la machine.

Ligne 5 : rotation des bits de l'octet 19200 sur la droite :



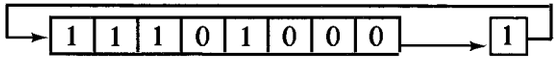
avant exécution



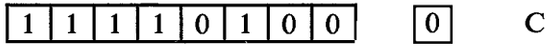
après exécution

Vous aviez remarqué que la précaution avait été prise aux lignes 3 et 4 de placer le chiffre 0 dans l'indicateur de retenue.

Ligne 6 : rotation des bits de l'octet 19201 sur la droite (le bit de retenue est à 1, ne l'oublions pas) :



avant exécution



après exécution

Faisons les comptes : les octets 19200 et 19201 valent respectivement en décimal 1 et 244. Lorsque l'on applique la règle poids fort/poids faible, on obtient :

$$1*256+244=500$$

Ceci est bien la réponse attendue.

Lorsqu'une nouvelle exécution du programme machine sera commandée par le BASIC, les octets 19200 et 19201 se verront décalés sur la droite, le bit sortant du premier étant réécrit au début du deuxième : c'est ainsi qu'une nouvelle division par deux sera réalisée.

ADCA

Cette instruction est l'abréviation de ADd with Carry into A. *Un nombre de 8 bits, le contenu de l'indicateur de retenue et la valeur de A sont ajoutés. Le résultat est mis dans l'accumulateur. Les modes d'adressage immédiat, indexé, étendu et direct sont autorisés.*

Exemple : MODE D'ADRESSAGE IMMÉDIAT (17 octets)

Programme assembleur

```
1            ORG    $4A00
2            EXC    DEBUT
3     DEBUT  LDD    $4B00     ; 19200 DECIMAL
4            ADDD   $4B02
5            STD    $4B0A     ; 19210 DECIMAL
6            LDAA   #$0
7            ADCA   #$0
8            STAA   $4B0C     ; 19212 DECIMAL
9            RTS
```

Programme BASIC

```
10 INPUT " PREMIER NBRE " ; N1
20 POKE 19200 , INT(N1/256)
30 POKE 19201 , N1 - INT(N1/256)*256
40 INPUT " DEUXIEME NBRE " ; N2
50 POKE 19202 , INT(N2/256)
60 POKE 19203 , N2 - INT(N2/256)*256
70 EXEC 18944 : PRINT " REPONSE " ;
80 ? 65536*PEEK(19212)+256*PEEK(19210)+PEEK(19211)
```

Commentaires

Voici une nouvelle utilisation du bit de retenue qui va nous permettre d'ajouter deux valeurs dont la somme dépasse 65535 et ceci avec, de la part de l'ordinateur, une réponse valable.

Lignes 3, 4 et 5 : l'accumulateur 16 bits D est chargé avec le nombre N1, il lui est ajouté le nombre N2, et le résultat est rangé, sous la forme poids fort/poids faible, dans les octets 19210 et 19211. Le programme pourrait très bien s'arrêter là si nous nous contentions d'ajouter deux nombres ayant une somme plus petite que 65536. Supposons qu'il n'en soit rien et proposons à l'ordinateur le calcul 50000 + 20000 ; il va considérer que 70000 se décompose en 65536 d'une part et en 4464 d'autre part. Cette dernière valeur sera écrite dans les octets 19210 et 19211 mais la machine va garder la trace du débordement de la capacité 16 bits en forçant à 1 le bit de retenue. Il nous faut voir comment nous allons pouvoir nous servir de cette indication.

Lignes 6 et 7 : ces deux lignes ont pour but d'écrire dans le registre A le chiffre du bit de retenue. On met l'accumulateur à zéro et on lui ajoute alors la retenue et la valeur 0. Au total, A contiendra bien la valeur d'origine de l'indicateur.

Ligne 8 : il ne reste plus qu'à ranger ce résultat dans l'octet 19212, là où le programme appelant pourra le retrouver.

En définitive, si le calcul de la somme dépasse 16 bits, le nombre 65536 est ajouté au résultat final par la ligne BASIC 80.

Un petit détail vous aura peut-être échappé : l'instruction LDAA #$0 de la ligne 6 n'a volontairement pas été remplacée par CLRA. Cette commande, nous l'avons vu, a une action sur le bit de retenue : elle le met toujours à zéro. Et, si nous l'avions utilisée, le programme n'aurait pas donné le résultat escompté. Vous vous en assurerez.

ADCB

Le résultat d'une somme entre une valeur 8 bits, le contenu de l'indicateur de retenue et celui du registre B est placé dans ce registre. On peut employer les modes immédiat, indexé, étendu et direct.

ASRA

Tous les bits de l'accumulateur sont décalés sur la droite et le bit 0 va dans l'indicateur de retenue. Mais le bit 7 reste inchangé. Le mode d'adressage inhérent est le seul possible.

Exemple : MODE D'ADRESSAGE INHÉRENT (8 octets)

Programme assembleur

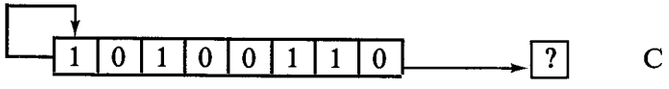
```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  LDAA   $4B00     ; 19200 DECIMAL
4          ASRA
5          STAA   $4B01     ; 19201 DECIMAL
6          RTS
```

Programme BASIC

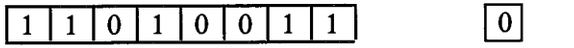
```
10 INPUT " DONNEZ UN NBRE NEGATIF " ; N
20 POKE 19200 , 256 + N : EXEC 18944
30 PRINT " VOICI SON QUOTIENT PAR 2 " ;
40 PRINT "-" ; 256 - PEEK(19201) : GOTO 10
```

Commentaires

Il faut rappeler que les nombres 8 bits dont l'écriture binaire commence par le chiffre 1 sont considérés par l'ordinateur comme négatifs. Par exemple – 90 s'obtient en calculant le complément à deux de 90, ce qui donne 10100110. Examinons quel sera l'effet de l'instruction ASRA sur ce nombre si l'on suppose qu'il est écrit dans l'accumulateur :



avant exécution



après exécution

Tous les chiffres ont été décalés sur la droite et le dernier d'entre eux est passé dans l'indicateur. Quant au bit 7, il valait 1 et, dans la place qu'il a laissée libre, le même chiffre 1 a été écrit. En se livrant au jeu des conversions, on obtient pour A la valeur décimale 211. Ainsi, à la suite de l'exécution de ASRA, l'accumulateur contient la traduction binaire de la valeur – 45. Voici donc compris le rôle de notre nouvelle instruction : elle permet de diviser par deux un nombre négatif tout en conservant son signe.

Il nous faut voir, au niveau du BASIC, par quelle gymnastique nous pouvons faire parvenir au processeur le nombre à diviser et récupérer ensuite le quotient.

N est un nombre négatif qu'il va falloir transmettre sur le mode complément à deux. Ceci se fait avec la commande POKE de la ligne 20 : en effet, en retranchant un nombre de 256, on obtient la valeur décimale de son complément à deux. 255 correspond par exemple à – 1, 254 à – 2, etc.

On reprendra la même méthode pour traduire (ligne 40) le nombre négatif que la machine aura calculé en une forme qui nous est habituelle.

Une dernière chose : ne manquez pas de proposer à l'ordinateur des nombres impairs ou des nombres dont la valeur absolue est supérieure à 127. Et essayez de retrouver à chaque fois où se trouve la logique d'une réponse apparemment incorrecte.

ASRB

C'est cette fois les bits de l'accumulateur B qui sont décalés sur la droite. Le bit 7 reste inchangé et le bit de droite tombe dans l'indicateur de retenue. On utilise cette instruction avec le mode inhérent.

ASR

Le contenu d'un octet mémoire est soumis à une rotation sur sa droite. Le bit 7 garde sa valeur d'origine et le bit 0 passe dans l'indicateur de retenue. On utilise les modes indexé et étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (42 octets)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3   DEBUT        JSR     $FBD4     ; CLS
4                LDD     #$4111
5                STD     $BF21     ; LETTRE A SOULIGNEE
6                LDD     #$1400
7                STD     #$BF26    ; COORDONNEES
8                LDD     #$8001
9                STAA    $BF23     ; COULEURS
10               LDX     #$8
11  SUITE        STAB    $BF28     ; EXECUTION
12  TEMPO        LDAA    $BF20
13               CMPA    #$80
14               BHS     TEMPO
15               ASR     $BF23     ; COULEURS
16               DEX
17               BNE     SUITE
18  FIN          BRA     FIN
```

Commentaires

Lignes 4 à 9 : on veut procéder à l'affichage d'un caractère ; on donne donc à l'ordinateur les renseignements suivants :

- le code de ce caractère : $41 (lettre A)
- le type de l'affichage : $11 (alphanumérique souligné)
- l'abscisse sur l'écran : $00 (colonne de gauche)
- l'ordonnée : $14 (rangée du milieu)
- la couleur : $80 (noir sur noir en vidéo inversée)

Ligne 10 : la boucle SUITE va être exécutée 8 fois. Le registre X, quand sa valeur sera ramenée à 0, indiquera au système que le programme est terminé.

Lignes 11 à 17 : voyons ce que fait le microprocesseur dans la boucle SUITE ; il donne l'ordre d'affichage de la lettre A, il attend que cet affichage soit effectivement réalisé puis il fait subir aux bits de l'octet $BF23 une rotation sur la droite. Puisque c'est de l'instruction ASR qu'il s'agit, cette rotation ne modifie pas la valeur du bit 7 de l'octet en question. Rappelons que le contenu de cet octet a une influence directe sur le coloriage des caractères. Voici donc les attributs des huit lettres qui apparaissent sur l'écran :

Valeurs successives de l'octet $BF23	Couleurs correspondantes (vidéo inversée)
1 0 0 0 0 0 0 0	Noir sur noir
1 1 0 0 0 0 0 0	Noir sur bleu
1 1 1 0 0 0 0 0	Noir sur turquoise
1 1 1 1 0 0 0 0	Noir sur blanc
1 1 1 1 1 0 0 0	Noir sur blanc + clignotant
1 1 1 1 1 1 0 0	Bleu sur blanc + clignotant
1 1 1 1 1 1 1 0	Turquoise sur blanc + clignotant
1 1 1 1 1 1 1 1	Blanc sur blanc + clignotant

COMA

Le contenu de l'accumulateur est remplacé par son complément logique. Chaque chiffre 1 est transformé en un chiffre 0 et réciproquement. Le mode d'adressage inhérent est le seul utilisable.

Exemple : MODE D'ADRESSAGE INHÉRENT (22 octets)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3     DEBUT      LDAA    #$F       ; 00001111 EN BINAIRE
4                PSHA
5     SUITE      JSR     $F883     ; INKEY$
6                CMPA    #$0
7                BEQ     SUITE
8                LDAB    #$5       ; DUREE
9                PULA
10               PSHA
11               JSR     $FFAB     ; SON
12               PULA
13               COMA
14               PSHA
15               BRA     SUITE
```

Commentaires

Lignes 3 et 4 : l'accumulateur A est chargé avec le nombre qui s'écrit 00001111 sur 8 bits en binaire (15 décimal) et cette valeur est aussitôt sauvegardée dans la pile S.

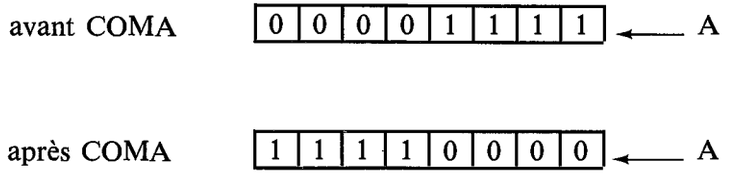
Lignes 5, 6 et 7 : on envoie le microprocesseur exécuter la routine qui débute à l'adresse $F883. Cette routine a pour mission de nous indiquer si une touche du clavier a été enfoncée. Elle agit en effet sur le registre A en y inscrivant soit la valeur 0 (aucune touche n'a été pressée), soit le code ASCII de la touche sur laquelle on a appuyé. Notre programme se maintient donc dans la boucle SUITE tant que nous n'intervenons pas sur le clavier.

Ligne 8 : une touche a été enfoncée. Une note de musique dont la durée est définie par le registre B va être jouée.

Ligne 9 : A contient pour l'instant le code de la touche pressée ; ce nombre n'a rigoureusement rien à voir avec notre programme et nous allons d'ailleurs le perdre au profit de la valeur qui est ressortie de la pile. A a maintenant la valeur $F (15 décimal).

Lignes 10 et 11 : on prend la précaution de remettre dans la pile l'accumulateur A car la routine qui est appelée détruit son contenu. Le passage du programme dans cette routine va produire un son grave de durée 5 (registre B) et de hauteur 15 (registre A).

Lignes 12 et 13 : nous retrouvons la valeur initiale de A en allant la chercher dans la pile et nous faisons agir notre nouvelle instruction :



Le registre prend la valeur $F0 (240 décimal).

Lignes 14 et 15 : on empile A et on retourne dans la routine de scrutation du clavier, routine de laquelle nous ne sortirons qu'en enfonçant une touche. Ce qui aura pour conséquence de nous faire entendre un nouveau son, très aigu celui-là (hauteur 240).

Puisque COMA aura fait reprendre au premier accumulateur la valeur 15, le son suivant sera très grave. Voilà comment s'explique l'alternance entre les deux types de son que nous entendons.

COMB

Chaque chiffre du registre B est remplacé par son opposé binaire. COMB est une instruction qui ne s'utilise que sur le mode inhérent.

COM

Cette instruction remplace le contenu d'un octet mémoire par son complément logique. On peut employer les adressages indexé et étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (35 octets)

Programme assembleur

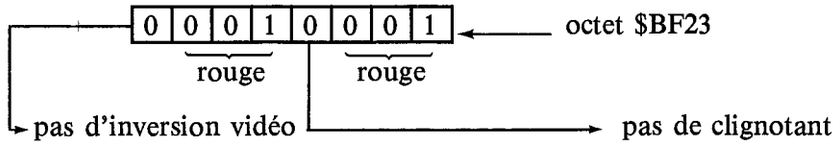
```
1                ORG     $4A00
2                EXC     DEBUT
3     DEBUT      JSR     $FBD4     ; CLS
4                CLR     $BF26     ; COLONNE 0
5                CLR     $BF27     ; LIGNE 0
6                LDD     #$2811
7                STAB    $BF23     ; ROUGE/ROUGE
8     SUITE      LDAB    #$1
9                STAB    $BF28     ; EXECUTION
10    TEMPO      LDAB    $BF20
11               CMPB    #$80
12               BHS     TEMPO
13               COM     $BF23     ; COULEUR
14               DECA
15               BNE     SUITE
16    FIN        BRA     FIN
```

Commentaires

Lignes 4 et 5 : nous allons utiliser le générateur vidéo pour afficher quarante caractères en couleur sur la première ligne de l'écran. En mettant à 0 les octets $BF26 et $BF27 on indique à l'ordinateur que le premier caractère doit apparaître en haut et à gauche de l'image.

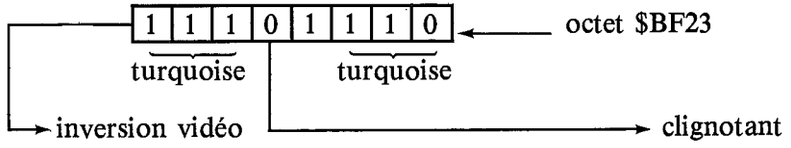
Lignes 6 et 7 : le registre A, chargé avec le nombre $28 (40 décimal), va être décrémenté (ligne 14) à chaque fois qu'un caractère sera affiché. C'est lui qui préviendra le microprocesseur que le programme

est terminé. Quant au registre B, il a comme rôle initial d'écrire la valeur $11 (17 décimal) dans l'octet $BF23. Cet octet a donc, pour l'instant, la structure suivante :



Lignes 8 et 9 : on envoie l'ordre d'affichage à l'octet voulu et un caractère apparaît alors sur la première case de l'écran. Notons que nous ne pouvons absolument pas savoir de quel caractère il s'agit mais une chose est certaine : il est colorié en rouge sur rouge. Voici que s'explique pourquoi nous ne voyons pas autre chose qu'un petit carré rouge.

Lignes 10 à 13 : on laisse le temps au système vidéo de terminer son travail et on remplace l'octet $BF23 par son complément logique. Il a donc maintenant la configuration suivante :



Au passage suivant dans la boucle SUITE, il sera donc procédé à l'affichage d'un petit carré bleu clair. Le fait qu'il soit dessiné en vidéo inversée, et sur le mode clignotant, ne sera pas visible pour nous. Impossible de distinguer quoi que ce soit quand l'affichage d'un caractère et de son fond est fait en utilisant la même couleur.

Lorsque le programme se terminera, nous pourrons voir en haut de l'écran une succession de cases rouges et bleues ; l'octet $BF23 aura pris alternativement les valeurs binaires complémentaires 00010001 et 11101110.

NEGA

La valeur du registre A est remplacée par son complément à deux. NEGA s'utilise en mode d'adressage inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (8 octets)

Programme assembleur

```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  LDAA   $4B00     ; 19200 DECIMAL
4          NEGA
5          STAA   $4B01     ; 19201 DECIMAL
6          RTS
```

Programme BASIC

```
10 INPUT " DONNEZ UN NBRE " ; N
20 POKE 19200 , N : EXEC 18944
30 PRINT " EN COMPLEMENT A 2 " ; -N ;
40 PRINT " S'ECRIT " ; PEEK(19201) : GOTO 10
```

Commentaires

Nous voici, avec ce programme, débarrassés de tous les problèmes d'écriture des nombres négatifs sur le mode complément à deux. L'instruction NEGA effectue pour nous les deux opérations nécessaires :

- complémentation logique,
- addition de 1 au résultat obtenu.

Ligne 3 : l'accumulateur est chargé avec le nombre N que le BASIC avait écrit dans l'octet 19200.

Ligne 4 : on recherche le complément à deux de N. Cette ligne aurait pu être remplacée par les deux instructions assembleur suivantes :

```
COMA
ADDA    #$1     (ou INCA)
```

Il ne reste ensuite plus qu'à écrire la réponse dans l'octet voulu. On a rencontré peu de programmes machine aussi faciles à comprendre, aussi ne nous attardons pas plus. Passons directement à l'étude de l'instruction suivante.

NEGB

Le complément à deux du contenu de B est calculé, puis remis dans ce registre. Le mode d'adressage utilisable est l'inhérent.

NEG

Le complément à deux du contenu d'un octet mémoire est effectué. Les modes d'adressage indexé et étendu sont permis.

Exemple : MODE D'ADRESSAGE ÉTENDU (44 octets)

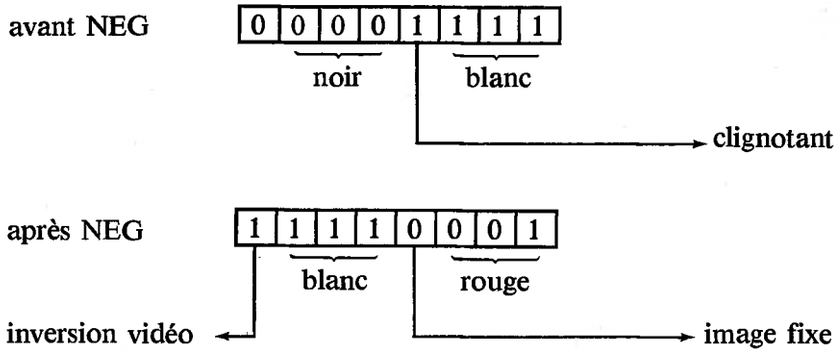
Programme assembleur

```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  JSR    $FBD4    ; CLS
4          LDD    #$4F01
5          STD    $BF21    ; LETTRE 0
6          LDD    #$0A0A
7          STD    $BF26    ; LIGNE 10 COL. 10
8          LDD    #$140F
9          STAB   $BF23    ; NOIR/BLANC+CLIGNO
10  SUITE  LDAB   #$01
11         STAB   $BF28    ; EXECUTION
12  TEMPO  LDAB   $BF20
13         CMPB   #$80
14         BHS    TEMPO
15         NEG    $BF23    ; CHANGEMENT COUL.
16         INC    $BF26    ; LIGNE SUIVANTE
17         DECA
18         BNE    SUITE
19  FIN    BRA    FIN
```

Commentaires

Lignes 4 à 11 : nous sommes maintenant habitués aux différentes actions que le microprocesseur doit effectuer pour afficher des caractères en couleur. Le premier de ceux-ci est un 0 qui clignotera en noir sur blanc à l'intersection de la quatrième ligne et de la onzième colonne.

Ligne 15 : passons quelques instants à étudier les modifications que réalise l'instruction NEG sur le contenu de l'octet $BF23. Il contient au début du programme (ligne 9) le nombre hexadécimal $0F c'est-à-dire le nombre binaire 000011111.



On en conclut qu'au deuxième passage de la boucle SUITE, la lettre 0 est dessinée en rouge sur blanc (en réalité : blanc sur rouge mais inversé). Au troisième passage, le caractère affiché retrouve ses attributs d'origine (noir sur blanc clignotant). Inutile de s'expliquer davantage, on voit bien ce que finit par réaliser ce programme. Notez toutefois que du fait que l'octet $BF26 est à chaque fois incrémenté, l'affichage change régulièrement de ligne. Voilà pourquoi les caractères sont disposés en diagonale sur l'écran.

JMP

Abréviation de JUMP, cette instruction branche le programme de manière inconditionnelle à l'adresse qui suit. On l'emploie avec les modes d'adressage indexé ou étendu.

Exemple : MODE D'ADRESSAGE ÉTENDU (22 octets)

Programme assembleur

```
 1            ORG     $4A00
 2            EXC     DEBUT
 3   DEBUT    JMP     BIDON
 4   BIDON    LDD     #$A       ; 10 DECIMAL
 5            LDX     #$3346    ; 13126 DECIMAL
 6   SUITE    STD     $2,X      ; NOUVEAU N°
 7            ADDD    #$A       ; N° LIGNE SUIV.
 8            LDX     $0,X
 9            CPX     #$0       ; FIN DU BASIC ?
10            BNE     SUITE
11            RTS
```

Programme BASIC

```
 5 REM RENUM
 8 PRINT " CE PROGRAMME "
14 PRINT " RENUMEROTE LES LIGNES "
22 PRINT " DE 10 EN 10 "
24 EXEC 18944
27 LIST
```

Commentaires

Ceci est le programme de renumérotation automatique des lignes : il permet d'obtenir des lignes BASIC écrites de 10 en 10. Démontons-en le mécanisme.

Ligne 3 : voici une présentation tout à fait artificielle ; elle ne sert qu'à mettre en avant la nouvelle instruction JMP. Celle-ci réalise un branchement inconditionnel vers n'importe quel octet de la mémoire ; en l'occurrence, pour nous, c'est du premier octet de la ligne suivante qu'il s'agit.

Lignes 4 et 5 : on charge D avec la valeur 10 ; c'est cet accumulateur qui nous servira à numéroter les lignes de 10 en 10. Puis on place dans X l'adresse du premier octet du programme BASIC, l'octet 13126 :

```
FOR I = 13126 TO 13141 : PRINT PEEK(I) ; : NEXT
```

Et voici la réponse :

51 82 0 5 131 32 82 69 78 85 77 0
(première ligne BASIC)
51 102 0 8 .... (début deuxième ligne BASIC)

51 et 82 donnent avec la règle poids fort/poids faible le nombre 13138, c'est-à-dire l'adresse du premier octet de la deuxième ligne.
0 et 5 forment le numéro de la première ligne BASIC.
131 est le code du mot réservé REM.
32, 82, 69, 78, 85 et 77 sont les codes ASCII des caractères qui suivent REM.
0 est le séparateur de deux lignes BASIC.
51 et 102 donnent l'adresse du premier octet de la troisième ligne.
0 et 8 constituent le numéro de la deuxième ligne, etc.

Revenons à l'assembleur. X pointe au départ sur le nombre 51 et ce que nous voulons, c'est remplacer les chiffres 0 et 5 par 0 et 10.

Lignes 6 et 7 : 0 est écrit dans l'octet 13128 et 10 dans le suivant. On ajoute ensuite 10 à D pour avoir le nouveau numéro de la ligne BASIC 8.

Ligne 8 : on passe à la deuxième ligne du programme BASIC et X, cette fois, va pointer sur l'octet 13138. C'est le numéro 20 qui se verra attribuer cette ligne.

Le programme bouclera tant que les deux zéros consécutifs qui indiquent la fin du BASIC n'auront pas été rencontrés.

TSTA

Cette instruction teste le contenu de l'accumulateur. Une instruction de branchement doit suivre normalement TSTA. Le seul mode d'adressage que l'on peut utiliser est l'inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (17 octets)

Programme assembleur

```
1           ORG     $4A00
2           EXC     DEBUT
3   DEBUT   JSR     EFF       ; CLS
4           CLR     $E8
5   SUITE   JSR     INKEY     ; TOUCHE ?
6           TSTA
7           BEQ     SUITE
8           JSR     AFCAR     ; AFFICHAGE
9           BRA     SUITE
10          ;
11  EFF     =       $FBD4
12  INKEY   =       $F883
13  AFCAR   =       $F9C6
```

Commentaires

Ligne 3 : l'écran est effacé à la suite de l'exécution de la routine $FBD4. Et le curseur positionné en haut et à gauche du téléviseur.

Lignes 5, 6 et 7 : l'ordinateur est envoyé dans la routine baptisée INKEY. Rappelons qu'à son retour, nous saurons si une touche a été enfoncée ou non et cela en étudiant le contenu du registre A. Si ce contenu est nul, aucune touche n'aura été pressée, sinon il sera égal au code ASCII de la touche appuyée.

L'instruction TSTA se borne à tester l'accumulateur A : a-t-il ou n'a-t-il pas la valeur 0 ? Comme on le voit, elle est parfaitement équivalente à CMPA #$0. Donc, tant qu'on ne touche pas le clavier, TSTA teste un registre nul et le programme boucle en parcourant les trois lignes 5, 6 et 7.

Ligne 8 : nous voici devant la routine que nous avons le plus souvent utilisée dans ce livre ; elle affiche le caractère dont le code est contenu par le registre A, c'est-à-dire très précisément le caractère que nous avons tapé au clavier. Ceci explique la raison pour laquelle ce caractère est apparu sur la première case de l'écran.

Ligne 9 : l'ordinateur est renvoyé de façon inconditionnelle à la ligne 5. Là, il attendra de nouveau que nous enfoncions une touche et, dès que ce sera fait, affichera le caractère correspondant sur l'écran, juste à droite du précédent.

Voilà, ceci se poursuivra jusqu'à ce que nous nous servions de la touche d'initialisation.

TSTB

C'est dans ce cas le contenu du registre B qui est testé. Cette instruction ne s'emploie qu'avec le mode inhérent.

TST

Cette instruction teste le contenu d'un octet mémoire. Un branchement doit normalement être effectué après ce test. On peut employer les modes d'adressage indexé et étendu.

Exemple : MODE D'ADRESSAGE INDEXÉ (42 octets + octets de la chaîne)

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3   DEBUT        JSR     $FBD4     ; CLS
4                LDD     #$0118
5                STD     $BF22     ; ROUGE/NOIR+CLIGNO.
6                LDD     #$0A00
7                STD     $BF26     ; LIGNE 4 COL. 0
8                LDX     #TEXTE
9   SUITE        LDAA    $0,X
10               STAA    $BF21     ; CARACTERE
11               LDAA    #$1
12               STAA    $BF28     ; EXECUTION
13  TEMPO        LDAA    $BF20
14               CMPA    #$80
15               BHS     TEMPO
16               INX
17               TST     $0,X
18               BNE     SUITE
19  FIN          BRA     FIN
20               ;
21  TEXTE        'ALICE AU PAYS DES MERVEILLES
22               DF0     $0
```


Commentaires

Lignes 4 à 7 : on se propose d'afficher le texte "ALICE AU PAYS DES MERVEILLES" à partir du début de la quatrième rangée. En inscrivant $18 (00011000 binaire) dans l'octet $BF23, on précise à l'ordinateur que les caractères doivent apparaître en rouge sur fond noir et qu'ils doivent clignoter.

Lignes 8, 9 et 10 : le registre X est chargé, en mode immédiat, avec l'adresse du premier octet de la chaîne TEXTE. Le couple d'instructions

```
    LDAA    $0,X
et  STAA    $BF21
```

va, par conséquent, écrire dans l'octet $BF21 le code ASCII de la première lettre de notre phrase.

Lignes 11 à 15 : la lettre A, début du mot ALICE, apparaît sur l'écran.

Ligne 16 : X est incrémenté et contient alors l'adresse de l'octet qui correspond au caractère L, caractère situé en deuxième position dans la chaîne TEXTE.

Lignes 17 et 18 : l'instruction TST vérifie que le contenu de l'octet pointé par X est différent de 0 et rebranche le programme à la ligne 9 pour, cette fois, provoquer l'apparition sur l'écran de la lettre L, en rouge sur fond noir là encore.

Ce qui vient d'être réalisé sur deux lettres va être reproduit pour tous les caractères de la phrase et le processus ne s'achèvera que lorsque le registre X pointera sur un octet nul. C'est pour cette raison que nous avons réservé un octet contenant la valeur 0 en fin de programme (instruction DF0 de la ligne 22).

TAB

TAB est une instruction qui permet de transférer le contenu du premier accumulateur dans le second. On emploie le mode d'adressage inhérent.

Exemple : MODE D'ADRESSAGE INHÉRENT (40 octets)

Programme assembleur

```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  JSR    $FBD4     ; CLS
4          LDX    #$0
5          LDAA   #$80      ; ASCII 128
6          LDAB   #$20      ; ASCII 32
7          CLR    $E8
8   PROG   JSR    $F9C6     ; AFCAR
9          PSHB
10         TAB
11         PULA
12         DEC    OCT
13         BNE    SUITE
14         PSHB
15         LDAB   #$28      ; 40 DECIMAL
16         STAB   OCT
17         TAB
18         PULA
19  SUITE  INX
20         CPX    #$3E7     ; 999 CASES
21         BNE    PROG
22  FIN    BRA    FIN
23  OCT    DF0    $28       ; 40 DECIMAL
```


Commentaires

Lignes 4, 5, 6 et 7 : le registre X est mis à zéro ; c'est lui qui comptera les 999 cases de l'écran. Les accumulateurs A et B sont chargés respectivement avec les codes ASCII de la case pleine — CHR$(128) — et de la case vide — CHR$(32). D'autre part, l'octet $E8 est forcé à 0.

Ligne 8 : branchement vers le sous-programme d'affichage. Puisque A contient la valeur 128, un petit carré va s'imprimer en haut et à gauche du téléviseur.

Lignes 9, 10 et 11 : on sauvegarde la valeur du registre B, soit 32, dans la pile système S. On transfère le contenu de A dans B : à ce moment précis, les deux registres contiennent le même nombre 128. Puis on ressort de la pile le nombre qui s'y trouvait pour l'écrire dans A ; ce qui fait que A contient maintenant la valeur 32. Ces trois lignes ont donc réalisé l'échange des deux accumulateurs.

Lignes 12 et 13 : on retranche 1 au contenu de l'octet OCT. Comme c'est le nombre 40 que nous y avions placé (ligne 23), l'instruction BNE va exécuter un branchement à la ligne 19 du programme. Là, on rajoute 1 au registre X et on retourne au sous-programme d'affichage. Puisque A vaut 32, c'est le caractère espace qui sera imprimé sur la deuxième case de l'écran. Un nouvel échange des accumulateurs va alors s'effectuer et au troisième passage de la boucle PROG, c'est une case pleine qui apparaîtra. L'alternance case foncée/case claire explique donc l'effet de damier obtenu.

Reste à voir les détails :

Lignes 14 à 18 : quand l'octet OCT arrive à zéro, cela correspond au fait que la droite de l'écran est atteinte. On procède alors à un nouvel échange des registres A et B pour que la deuxième ligne débute par une case claire. Le programme respectera alors la structure d'un damier. Naturellement il a fallu remettre au niveau 40 l'octet OCT pour que le passage de la deuxième à la troisième ligne se fasse de façon correcte.

TBA TAP TPA TSX TXS

Ces instructions réalisent, sur le mode d'adressage inhérent, des transferts entre les registres qui sont écrits à la suite de la lettre T.

CONCLUSION

Ce livre a constitué une introduction à la programmation en assembleur de l'ordinateur Alice. Nous en avons étudié les aspects les plus importants et réalisé une série d'exercices qui vous ont montré, c'est notre souhait, que l'assembleur pouvait être assimilé sans difficulté par un lecteur armé de sa seule bonne volonté. Nous sommes persuadés, pour notre part, qu'il est infiniment plus long d'acquérir la logique de la programmation BASIC que celle de l'assembleur.

Vous êtes maintenant en mesure de créer vos propres programmes et d'inclure dans vos lignes BASIC des effets spéciaux que seule l'impressionnante rapidité de l'assembleur autorise. Si l'occasion se présente, vous ne manquerez pas de chercher à quoi correspondent les codes machine que d'autres programmeurs auront obtenus, faisant ainsi le travail inverse de celui qui a été effectué jusqu'à maintenant. Cette opération, qui s'appelle le désassemblage, vous permettra de reconstruire le programme assembleur et éventuellement de le modifier pour qu'il s'adapte très précisément à votre cas.

Naturellement, rien ne vous empêche de franchir une nouvelle étape en vous orientant vers des ouvrages plus techniques que celui-ci[1]. Vous y trouverez des programmes applicables à la gestion des périphériques ainsi que des explications concernant les quelques instructions que nous avons volontairement passées sous silence, estimant que, dans un premier temps en tout cas, leur intérêt était négligeable.

Ce livre s'achève sur deux programmes un peu plus compliqués que les autres. Vous les aborderez sans complexes maintenant que vous est ouvert l'étroit mais ô combien royal chemin de l'assembleur.

1. *Programmation du 6800* par Rodnay Zaks et Daniel Jean David.

ANNEXE A
TRI EN MÉMOIRE CENTRALE

Programme assembleur

```
1                ORG     $4A00
2                EXC     DEBUT
3     DEBUT      CLRA
4     SUIT2      TAB
5                INCB
6     SUIT1      PSHA
7                PSHB
8                LDX     #$4B00    ; PREMIER ELEMENT
9                TAB
10               ABX
11               LDAA    $0,X
12               LDX     #$4B00
13               PULB
14               ABX
15               CMPA    $0,X
16               BLS     AUTRE     ; PAS D'ECHANGE
17    ECHAN      STAA    OCT1
18               LDAA    $0,X
19               STAA    OCT2
20               PULA
21               PSHA
22               PSHB
23               TAB
24               LDX     #$4B00
25               ABX
26               LDAA    OCT2
27               STAA    $0,X
28               PULB
29               LDX     #$4B00
30               ABX
31               LDAA    OCT1
32               STAA    $0,X
33    AUTRE      PULA
34               INCB
35               CMPB    #$63      ; FIN D'UNE PASSE ?
36               BLS     SUIT1
37               INCA
38               CMPA    #$62      ; FIN DU TRI ?
39               BLS     SUIT2
40               RTS
41    OCT1       DF0     $0
42    OCT2       DF0     $0
```


Programme BASIC

```
10 DIMA(99) : FOR I = 0 TO 99
20 A(I) = RND(250) : POKE 19200 + I , A(I)
30 NEXT : EXEC 18944
40 FOR I = 0 TO 99 : PRINT " A(";I;") VALAIT";A(I)
50 A(I) = PEEK(19200+I)
60 PRINT " A(";I;") VAUT";A(I) : NEXT
```

Commentaires

Cent nombres compris entre 1 et 250 sont tirés au sort et placés dans un tableau. Le but du programme assembleur est de trier les cent valeurs et de les recopier, rangées par ordre croissant, dans le tableau. A(0) sera donc l'élément minimum et A(99) l'élément maximum. Pour permettre au programme de retrouver sans peine ces cent valeurs, elles vont transiter dans les octets 19200 ($4B00) à 19299.

Lignes 3, 4 et 5 : les nombres 0 et 1 sont écrits respectivement dans les accumulateurs A et B. Ces deux registres, quand ils seront ajoutés à X, indiqueront de quel élément du tableau il est question.

Lignes 8 à 11 : on fait pointer le registre X vers le premier élément du tableau (qui se trouve donc dans l'octet 19200) et l'on charge A avec la valeur de cet élément.

Lignes 12 à 16 : X pointe cette fois-ci sur l'octet 19201 et une comparaison entre les nombres écrits dans les octets 19200 et 19201 est établie. Si le premier de ces nombres est inférieur ou égal au deuxième, rien n'est modifié et le programme se branche directement à la ligne 33. Là, le registre B est incrémenté et l'ordinateur retourne à l'étiquette SUIT1 pour comparer le premier et le troisième élément du tableau.

Lignes 17 à 32 : si, en revanche, le deuxième nombre est inférieur au premier, on procède à leur échange. Nous sommes donc sûrs, à la fin de la boucle SUIT1, que A(0) est l'élément minimum du tableau.

Lignes 37 à 39 : l'accumulateur A est incrémenté et un retour à SUIT2 est programmé. Pour, cette fois-ci, comparer A(1) avec les éléments suivants, un échange sera effectué à chaque fois que l'on aura trouvé un nombre plus petit que A(1).

Ainsi, après deux passages de la boucle SUIT2, A(0) et A(1) auront les deux plus petites valeurs du tableau, et ceci dans l'ordre croissant. Quand le programme arrivera à son terme, tous les éléments seront rangés correctement et les lignes BASIC 40, 50 et 60 nous en donneront la confirmation.

ANNEXE B
AFFICHAGE D'UN MESSAGE

Voici un programme qui permet l'affichage d'une phrase dans les couleurs de son choix. L'affichage est réalisé en lettres clignotantes de largeur double.

Programme assembleur (51 octets)

```
1          ORG    $4A00
2          EXC    DEBUT
3   DEBUT  LDX    #TEXTE
4   SUITE  INC    $BF27
5          LDD    #$2001
6          STD    $BF21     ;
7          BSR    SPROG     ; AFFICHAGE
8          DEC    $BF27
9          DEC    $BF27
10         LDAA   $0,X
11         LDAB   #$09      ; DOUBLE LARGEUR
12         STD    $BF21
13         BSR    SPROG
14         INC    $BF27
15         INX
16         TST    $0,X      ; FIN DE PHRASE ?
17         BNE    SUITE
18         RTS
19         ;
20  SPROG  LDAA   #$1
21         STAA   $BF28     ; EXECUTION
22  TEMPO  LDAA   $BF20
23         CMPA   #$80
24         BHS    TEMPO
25         RTS
26         ;
27  TEXTE
```


Programme BASIC

```
10 CLEAR 100 , 18000 : FOR I = 18944 TO 18994
20 READ J : POKE I , J : NEXT
30 INPUT " DONNEZ UNE PHRASE " ; A$
40 L = LEN (A$) : IF L<1 OR L>19 THEN 30
50 FOR I = 1 TO L : POKE 18994+I, ASC (MID$ (A$ , I , 1))
60 NEXT : POKE 18994 + I , 0
70 INPUT " COULEUR DES LETTRES " ; C
80 INPUT " COULEUR DU FOND " ; F
90 INPUT " CLIGNOTANT " ; C$ : CLS
100 POKE 48931 , 16*C+F-(C$="0") * 8 : REM COULEUR
110 POKE 48934 , 20 : REM CADRAGE HORIZONTAL
120 POKE 48935 , (40-2*L)/2 : REM CADRAGE VERTICAL
130 EXEC 18944
140 DATA 206 , 74 , 51 , 124 , 191 , 39 , 204 , 32 , 1 , 253 , 191 ,
    33 , 141 , 24
150 DATA 122 , 191 , 39 , 122 , 191 , 39 , 166 , 0 , 198 , 9 , 253 ,
    191 , 33
160 DATA 141 , 9 , 124 , 191 , 39 , 8 , 109 , 0 , 38 , 222 , 57 ,
    134 , 1 , 183
170 DATA 191 , 40 , 182 , 191 , 32 , 129 , 128 , 36 , 249 , 57
```

Commentaires

Le programme BASIC écrit dans l'octet 48931 la valeur correspondant à notre choix de couleurs et dans les octets 48934 et 48935 les coordonnées de la première lettre. C'est lui aussi qui place en mémoire, à partir de l'adresse 18995, les codes ASCII des caractères qui constituent le message à afficher. Cette liste de codes doit se finir par le chiffre 0 pour que l'ordinateur comprenne que le processus d'affichage est terminé.

Les manipulations que nous faisons subir à l'octet $BF27 (incrémentations et décrémentations successives) tiennent au fait que lorsque l'on veut faire apparaître un caractère en double largeur, sur deux cases donc, on est obligé de respecter l'enchaînement suivant :

- coloration de la deuxième case dans les tons voulus : ceci se réalise par impression d'un espace, de largeur normale, dans cette case ;
- affichage, dans la première case, du caractère ; son graphisme débordera, avec les couleurs correctes, sur la deuxième case.

ANNEXE C
JEU D'INSTRUCTIONS DU 6803

INSTRUCTIONS	FORME	MODES D'ADRESSAGE									
		IMMÉDIAT		DIRECT		INDEXÉ		ÉTENDU		INHÉRENT	
		CM	NO	CM	NO	CM	NO	CM	NO	CM	NO
AB	ABA ABX									1B 3A	1 1
ADC	ADCA ADCB	89 C9	2 2	99 D9	2 2	A9 E9	2 2	B9 F9	3 3		
ADD	ADDA ADDB ADDD	8B CB C3	2 2 3	9B DB D3	2 2 2	AB EB E3	2 2 2	BB FB F3	3 3 3		
AND	ANDA ANDB	84 C4	2 2	94 D4	2 2	A4 E4	2 2	B4 F4	3 3		
ASL	ASLA ASLB ASLD ASL					 68	 2	 78	 3	48 58 05	1 1 1
ASR	ASRA ASRB ASR					 67	 2	 77	 3	47 57	1 1
BIT	BITA BITB	85 C5	2 2	95 D5	2 2	A5 E5	2 2	B5 F5	3 3		
CBA	CBA									11	1
CLC	CLC									0C	1
CLI	CLI									0E	1
CLR	CLRA CLRB CLR					 6F	 2	 7F	 3	4F 5F	1 1
CLV	CLV									0A	1
CMP	CMPA CMPB CPX	81 C1 8C	2 2 3	91 D1 9C	2 2 2	A1 E1 AC	2 2 2	B1 F1 BC	3 3 3		

INSTRUCTIONS	FORME	MODES D'ADRESSAGE									
		IMMÉDIAT		DIRECT		INDEXÉ		ÉTENDU		INHÉRENT	
		CM	NO	CM	NO	CM	NO	CM	NO	CM	NO
COM	COMA COMB COM					 63	 2	 73	 3	43 53	1 1
DAA	DAA									19	1
DEC	DECA DECB DEC DES DEX					 6A	 2	 7A	 3	4A 5A 34 09	1 1 1 1
EOR	EORA EORB	88 C8	2 2	98 D8	2 2	A8 E8	2 2	B8 F8	3 3		
INC	INCA INCB INC INS INX					 6C	 2	 7C	 3	4C 5C 31 08	1 1 1 1
JMP	JMP					6E	2	7E	3		
JSR	JSR			9D	2	AD	2	BD	3		
LD	LDAA LDAB LDD LDS LDX	86 C6 CC 8E CE	2 2 3 3 3	96 D6 DC 9E DE	2 2 2 2 2	A6 E6 EC AE EE	2 2 2 2 2	B6 F6 FC BE FE	3 3 3 3 3		
LSR	LSRA LSRB LSRD LSR					 64	 2	 74	 3	44 54 04	1 1 1
MUL	MUL									3D	1
NEG	NEGA NEGB NEG					 60	 2	 70	 3	40 50	1 1
NOP	NOP									01	1
ORA	ORAA ORAB	8A CA	2 2	9A DA	2 2	AA EA	2 2	BA FA	3 3		
PSH	PSHA PSHB PSHX									36 37 3C	1 1 1
PUL	PULA PULB PULX									32 33 38	1 1 1

INSTRUCTIONS	FORME	MODES D'ADRESSAGE									
		IMMÉDIAT		DIRECT		INDEXÉ		ÉTENDU		INHÉRENT	
		CM	NO	CM	NO	CM	NO	CM	NO	CM	NO
ROL	ROLA ROLB ROL					 69	 2	 79	 3	49 59	1 1
ROR	RORA RORB ROR					 66	 2	 76	 3	46 56	1 1
RTI	RTI									3B	1
RTS	RTS									39	1
SBA	SBA									10	1
SBC	SBCA SBCB	82 C2	2 2	92 D2	2 2	A2 E2	2 2	B2 F2	3 3		
SEC	SEC									0D	1
SEI	SEI									0F	1
SEV	SEV									0B	1
ST	STAA STAB STD STS STX			97 D7 DD 9F DF	2 2 2 2 2	A7 E7 ED AF EF	2 2 2 2 2	B7 F7 FD BF FF	3 3 3 3 3		
SUB	SUBA SUBB SUBD	80 C0 83	2 2 3	90 D0 93	2 2 2	A0 E0 A3	2 2 2	B0 F0 B3	3 3 3		
SWI	SWI									3F	1
T	TAB TAP TBA TPA TSX TXS									16 06 17 07 30 35	1 1 1 1 1 1
TST	TSTA TSTB TST					 6D	 2	 7D	 3	4D 5D	1 1
WAI	WAI									3E	1

Note : LSL pourra être employée à la place de ASL.

Abréviations utilisées dans les modes d'adressage :

- CM : code machine hexadécimal de l'instruction.
- NO : nombre total d'octets nécessaires.

ANNEXE D
INSTRUCTIONS DE BRANCHEMENT

INSTRUCTION	CODE MACHINE	UTILISATION
BRA	20	Branchement inconditionnel
BRN	21	Branchement jamais
BHI	22	Branchement si supérieur
BLS	23	Branchement si inf. ou égal
BHS	24	Branchement si sup. ou égal
BLO	25	Branchement si inférieur
BNE	26	Branchement si non égal
BEQ	27	Branchement si égal
BVC	28	Branchement si débordement à 0
BVS	29	Branchement si débordement à 1
BPL	2A	Branchement si positif
BMI	2B	Branchement si négatif
BGE	2C	Branchement si sup. ou égal *
BLT	2D	Branchement si inférieur *
BGT	2E	Branchement si supérieur *
BLE	2F	Branchement si inf. ou égal *
BSR	8D	Branchement vers un s/programme

* valeurs en complément à deux.

Notes :

- Toutes ces instructions utilisent le mode d'adressage relatif (deux octets au total).
- BCC et BCS pourront être employées à la place de BHS et BLO.

ANNEXE E
TABLE DE CONVERSION HEXADÉCIMALE

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	B5	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

TABLE DES MATIÈRES

LA BIBLIOTHÈQUE SYBEX

OUVRAGES GÉNÉRAUX

VOTRE PREMIER ORDINATEUR *par Rodnay Zaks,*
296 pages, Réf. 394

VOTRE ORDINATEUR ET VOUS *par Rodnay Zaks,*
296 pages, Réf. 271

DU COMPOSANT AU SYSTÈME : une introduction aux microprocesseurs *par Rodnay Zaks,*
636 pages, Réf. 340

TECHNIQUES D'INTERFACE aux microprocesseurs *par Austin Lesea et Rodnay Zaks,*
450 pages, Réf. 339, 3ème édition

LEXIQUE INTERNATIONAL MICROORDINATEURS, avec dictionnaire abrégé en 10 langues
192 pages, Réf. 234

GUIDE DES MICRO-ORDINATEURS A MOINS 3 000 F *par Joël Poncet,*
144 pages, Réf. 322

LEXIQUE MICRO-INFORMATIQUE *par Pierre Le Beux,*
140 pages, Réf. 369

LA SOLUTION RS-232 *par Nelson Ford,*
208 pages, Réf. 352

BASIC

VOTRE PREMIER PROGRAMME BASIC *par Rodnay Zaks,*
208 pages, Réf. 263

INTRODUCTION AU BASIC *par Pierre Le Beux,*
336 pages, Réf. 335

LE BASIC PAR LA PRATIQUE : 60 exercices *par Jean-Pierre Lamoitier,*
252 pages, Réf. 395

LE BASIC POUR L'ENTREPRISE *par Xuan Tung Bui,*
204 pages, Réf. 253, 2ème édition

PROGRAMMES EN BASIC, Mathématiques, Statistiques, Informatique *par Alan R. Miller,*
318 pages, Réf. 259

AU COEUR DES JEUX EN BASIC *par Richard Mateosian,*
352 pages, Réf. 233

JEUX D'ORDINATEUR EN BASIC *par David H. Ahl,*
192 pages, Réf. 246

NOUVEAUX JEUX D'ORDINATEUR EN BASIC *par David H. Ahl,*
204 pages, Réf. 247

Imprimé en France par
ALSACIENNE D'IMPRESSION Colmar-Paris
Dépôt légal N° 5515 - Septembre 84