MICRO-ORDINATEURS

# LA CONDUITE DES ALICE 32 ET ALICE 9Ø

par

**François BERNARD**

Collection animée
par Richard SCHOMBERG

EYROLLES 61, Boulevard Saint-Germain — 75005 Paris
1985

Si vous désirez être tenu au courant de nos publications, il vous suffit d'adresser votre carte de visite au :

Service « Presse », Éditions EYROLLES
61, Boulevard Saint-Germain,
75240 PARIS CEDEX 05,

en précisant les domaines qui vous intéressent. Vous recevrez régulièrement un avis de parution des nouveautés en vente chez votre libraire habituel.

## CHEZ LE MEME EDITEUR

**Dans la même collection**

SAGUEZ et ANDRIEUX - *Maîtrisez les interfaces de votre micro-ordinateur* - 144 p. ; 1984.

SCHOMBERG - *Le Basic universel* - 128 p. ; 1983.

SCHOMBERG - *Micro-ordinateurs : comment ça marche ?* - 96 p. ; 1983.

VULDY - *Graphisme 3 D sur votre micro-ordinateur* - 128 p. ; 1985.

**Autres ouvrages**

DELAHAYE - *Dessins géométriques et artistiques avec votre micro-ordinateur* - 256 p. ; 1985.

DELANNOY - *Apprendre à programmer en Basic* - 272 p. ; 1984.

DE ROSSI - *Apprentissage rapide du Basic* - 216 p. ; 1984, (coll. Pratique de l'Informatique).

DUCAMP et SCHAEFFER - *Réalisez vos jeux éducatifs* - 144 p. ; 1985.

GROS - *Dessiner, peindre et jouer avec ALICE et TANDY MC 10* - 152 p. ; 1984.

HIRSCH - *Le Basic facile par une méthode progressive* - 288 p. ; 1984, (coll. Pratique de l'Informatique).

O'MALLEY - *25 programmes graphiques en Basic Microsoft* - 192 p. ; 1985.

PEZERET - *Créez vos jeux d'aventure sur micro-ordinateur.* Méthodes et idées - 144 p. ; 1985.

# Avant-propos

*Ce livre s'adresse à tous les possesseurs d'un Alice 32 ou d'un Alice 9Ø. En effet, ces deux machines, dotées d'une mémoire et de possibilités graphiques plus étendues que l'ancien Alice 4K, sont totalement compatibles tant sur le point matériel que logiciel, comme cela sera décrit tout au long de cet ouvrage et en particulier dans le chapitre 3 traitant de leur architecture matérielle.*

*Un chapitre entier est associé à la description des diverses instructions et commandes Basic. Le fonctionnement et la syntaxe de chacune d'elles est décrit en profondeur et illustré d'exemples pratiques.*

*Les possibilités graphiques de l'Alice ainsi que les différents modes d'affichage font l'objet d'un chapitre particulier.*

*Cependant, la majeure partie de cet ouvrage est consacrée à l'accès au langage assembleur. En effet, une des particularités les plus intéressantes des Alice 32 et Alice 90 est la présence d'un Éditeur-Assembleur incorporé en ROM.*

*Ainsi, grâce aux chapitres 4 et 5, vous découvrirez la structure du microprocesseur qui équipe ces machines, le 68Ø3, ses registres internes, ses modes d'adressage et son jeu d'instructions.*

*Ces dernières sont décrites les unes après les autres, illustrées de nombreux exemples qui utilisent la syntaxe de votre Éditeur-Assembleur. Pour ceux qui veulent aller plus loin dans ce domaine, le chapitre 6 décrit des routines présentes dans la ROM de l'Alice et qui, par un simple appel, permettent d'afficher des caractères sur écran, de lire le clavier, etc...*

*De plus, les passionnés de musique électronique trouveront dans le chapitre 7 un certain nombre de routines assembleur qui, incorporées à des programmes Basic ou Assembleur, produisent des effets sonores saisissants. Mais jugez-en par vous-même.*

# Table des matières

**Avant-propos** ..... VII

**1. Première prise de contact** ..... 1

**2. Le Basic de l'Alice** ..... 3

2.1. Introduction ..... 3
2.2. Les variables ..... 3
2.3. Les opérateurs ..... 4
2.4. Les commandes Basic ..... 6
2.5. Les instructions Basic ..... 10

**3. L'architecture du système** ..... 34

3.1. L'architecture ..... 34
3.2. Le Mapping ..... 37

**4. Introduction du 68Ø3** ..... 40

4.1. Introduction ..... 40
4.2. Les systèmes numériques ..... 45
4.3. La syntaxe assembleur 68Ø3 ..... 49
4.4. Les registres internes du 68Ø3 ..... 59
4.5. Les différents modes d'adressage du 68Ø3 ..... 68

**5. Le jeu d'instructions du 68Ø3** ..... 73
5.1. Introduction ..... 73
5.2. Les instructions de chargement ..... 74
5.3. Les instructions arithmétiques ..... 85
5.4. Les instructions logiques ..... 102
5.5. Les instructions sur le registre d'état ..... 114
5.6. Les instructions de comparaison ..... 115
5.7. Les instructions de branchement ..... 116
5.8. Les instructions d'appel et de retour de sous-programme ..... 122
5.9. Les instructions sur la pile ..... 123
5.10 Les instructions spéciales ..... 126

**6. Les sous-programmes système** ..... 130

**7. Les possibilités sonores de l'Alice** ..... 145

**8. Les possibilités graphiques de l'Alice** ..... 150
8.1. Introduction ..... 150
8.2. Les écrans de l'Alice ..... 151

**Table de conversion hexadécimal → décimal** ..... 159

**Tableau récapitulatif des instructions du 68Ø3** ..... 159

# 1
# Première prise de contact

L'Alice, fabriqué par la société française MATRA, est disponible sous deux versions : l'Alice 32 et l'Alice 90. Ces deux machines sont tout à fait différentes de la première version de l'Alice, similaire au MC 1Ø de Tandy.

En effet, cette dernière machine ne possédait que 4 Koctets de mémoire vive (RAM) et des possibilités graphiques quasiment inexistantes. L'Alice 32 et l'Alice 9Ø, au contraire, possèdent respectivement 8 Koctets et 32 Koctets de RAM utilisateur. En plus de cela, ils disposent chacun d'une mémoire écran de 8 Koctets ce qui leur permet d'afficher des graphismes en couleur de bonne résolution.

Mise à part la quantité de mémoire utilisateur et la présentation (clavier en particulier) ces deux machines sont identiques car elles possèdent :

— le même microprocesseur : le 68Ø3,

— Le même interpréteur Basic (similaire à celui de l'Alice 4K/Tandy MC 1Ø),

— un éditeur-assembleur en ROM,

— les mêmes modes d'affichage texte (16 lignes × 32 colonnes, 25 lignes × 4Ø colonnes et 25 lignes × 8Ø colonnes),

— les mêmes modes d'affichage graphique (16Ø × 125 sous Basic et 16Ø × 25Ø sous Assembleur),

— une sortie cassette pour la sauvegarde de données et programmes,

— une sortie imprimante permettant de connecter soit l'imprimante Alice, soit toute autre imprimante à liaison série.

Ces deux versions d'Alice sont connectables à tout téléviseur couleur muni d'une prise PERITEL.

Si vous ne disposez pas de téléviseur muni d'une telle prise, vous pourrez néanmoins disposer d'un affichage monochrome grâce à un adaptateur noir et blanc prévu pour fonctionner avec l'Alice.

Puisque, comme nous l'avons dit, l'Alice 32 et l'Alice 9Ø sont en tous points similaires du point de vue fonctionnement, cet ouvrage s'adresse indifféremment aux possesseurs de l'une ou l'autre machine.

Le chapitre consacré à l'"Architecture du système" donne tous les détails nécessaires concernant la carte mémoire des deux versions, ceci afin de permettre aux utilisateurs de l'Éditeur-Assembleur de définir l'adresse d'implantation en mémoire de leurs programmes.

Cette mise au point faite, vous pouvez connecter l'alimentation à votre Alice ainsi que le câble PERITEL.

Lors de la mise sous tension, le logiciel contenu dans la ROM 16 Koctets initialise certaines variables nécessaires à son fonctionnement ainsi que l'affichage. Vous vous trouvez alors en mode 16 lignes sur 32 colonnes et vous devez voir s'afficher le message de bienvenue suivant :

```
MICROCOLOR BASIC 1.0
COPYRIGHT 1982 MICROSOFT
OK
```

# 2

# Le Basic de l'Alice

## 2.1. INTRODUCTION

Dans ce chapitre, nous vous proposons de décrire toutes les instructions et commandes Basic présentes sur l'Alice. Nous avons choisi de vous les présenter en les regroupant par affinités, ceci afin de vous permettre de les utiliser plus rapidement de façon plus efficace. De plus, de nombreux exemples précisent leur emploi.

Ceci dit, ce chapitre ne prétend pas être un cours de Basic : pour plus de précisions concernant certaines instructions, vous pourrez vous reporter à des ouvrages spécialisés tels que "Le Basic Universel" paru dans la même collection.

## 2.2. LES VARIABLES

Le Basic de l'Alice accepte deux types de variables qui sont les suivants :

— type réel,

— type chaîne.

A chaque variable est attribué un nom comportant un ou deux caractères alphanumériques dont le premier est obligatoirement une

lettre. En fait, des noms plus longs peuvent être utilisés mais l'interpréteur Basic ne prend en compte que les deux premiers caractères.

Il convient donc de faire attention aux noms de variables différents mais commençant par deux lettres semblables.

Une variable de type chaîne est déclarée par le caractère additionnel **$** (dollar). En l'absence de ce dernier, le Basic considère la variable comme réelle.

*Exemples* :

| | | |
|---|---|---|
| AC | B | sont des variables réelles |
| A1$ | B$ | sont des chaînes de caractères |

## 2.3. LES OPÉRATEURS

Il existe trois types d'opérateurs :

— les opérateurs arithmétiques,
— les opérateurs de comparaison,
— les opérateurs logiques.

### 2.3.1. Les opérateurs arithmétiques

Ce sont :

— l'addition +,
— la soustraction —,
— la multiplication *,
— la division /,
— l'élévation à une puissance ↑,

Il existe une hiérarchie entre ces différents opérateurs :

a) l'élévation à une puissance,
b) la multiplication ou la division,
c) l'addition ou la soustraction.

*Exemple* :

3.28 + 4 * 6↑2 sera évalué de la façon suivante :
3.28 + (4 * (6↑2)) = 147.28

### 2.3.2. Les opérateurs de comparaison

Les comparaisons peuvent s'effectuer avec des variables numériques ou de type chaîne. Les opérateurs ont :

| | | |
|---|---|---|
| — égal à | : = | |
| — inférieur à | : < | (inégalité stricte) |
| — supérieur à | : > | (inégalité stricte) |
| — inférieur ou égal à | : <= | (inégalité large) |
| — supérieur ou égal à | : >= | (inégalité large) |
| — différent de | : <> | |

Dans le cas de variables de type chaîne, l'Alice prend chaque caractère l'un après l'autre et compare les codes ASCII correspondants.

### 2.3.3. Les opérateurs logiques

Les opérateurs logiques sont :

— le "ET" logique : AND
— le "NON" logique : NOT
— le "OU" logique : OR

Ces trois opérateurs travaillent sur des entiers uniquement et les opérations s'effectuent bit par bit.

Les tables de vérité de ces différents opérateurs sont données ci-dessus :

AND

| Y \ X | Ø | 1 |
|---|---|---|
| Ø | Ø | Ø |
| 1 | Ø | 1 |

Donc X AND 1 = X
et X AND Ø = Ø

OR

| X \ Y | Ø | 1 |
|---|---|---|
| Ø | Ø | 1 |
| 1 | 1 | 1 |

Donc X OR 1 = 1
et X OR Ø = X

NOT donne le complément de la variable considérée. On aura donc NOT X = $\overline{X}$ ($\overline{X}$ = complément de X)

| X | Ø | 1 |
|---|---|---|
| $\overline{X}$ | 1 | Ø |

*Exemple*:

36 AND 15 = 4

En effet:

36 = ØØ1ØØ1ØØ
15 = ØØØØ1111

36 AND 15 = ØØØØØ1ØØ = 4

De même:

et

36 OR 15 = ØØ1Ø1111 = 47

NOT 36 = 11Ø11Ø11 = –37

(en notation en complément à deux qui sera décrite dans le chapitre intitulé: "Introduction au 68Ø3").

## 2.4. LES COMMANDES BASIC

Dans les lignes qui suivent nous allons décrire les diverses commandes Basic disponibles sur l'Alice.

CLOAD

Cette commande permet de charger en mémoire RAM un programme Basic préalablement stocké sur cassette.

La syntaxe est la suivante :

```
CLOAD "nom de fichier"
```

Notons que le nom de fichier est optionnel et que, lorsqu'il est omis, il y a chargement du premier programme situé sur cassette.

Le nom de fichier ne peut contenir plus de 8 caractères.

Le bon fonctionnement de cette instruction est assuré par l'apparition des lettres "S" (venant de l'Anglais "searching" signifiant chercher) et "F" (de l'Anglais "found" signifiant trouve).

CLOAD*

Cette commande est une variante de la commande CLOAD. En effet, elle permet non pas de charger un programme enregistré sur cassette mais des données stockées sous la forme d'un tableau.

La syntaxe est la suivante :

```
CLOAD* N,"nom de fichier"
```

N désignant le nom du tableau à charger (ses éléments seront donc N(1), N(2),..., N(i), i désignant le nombre d'éléments, enregistrés sur cassette, que l'on désire charger).

Le tableau N devra avoir été préalablement dimensionné par une instruction DIM.

On aura par exemple :

```
10 DIM N(2)
20 CLOAD* N,"TOTO"
```

CLOADM

Cette commande permet de charger en mémoire RAM un programme assembleur préalablement stocké sur cassette.

La syntaxe est la suivante :

```
CLOADM "nom de fichier"
```

Comme dans l'instruction CLOAD, le nom de fichier est optionnel et ne peut contenir plus de huit caractères.

CSAVE

A l'inverse de la commande CLOAD, la commande CSAVE permet de sauvegarder un programme Basic se trouvant en mémoire sur cassette.

La syntaxe est la suivante :

```
CSAVE"nom de fichier".
```

CSAVE*

Cette commande est à CSAVE ce que CLOAD* est à CLOAD.

Elle permet donc de sauvegarder sur cassette des données stockées à l'intérieur d'un tableau.

La syntaxe est la suivante :

```
CSAVE* N,"nom de fichier".
```

N désignant toujours le nom du tableau constitué des éléments N(1), N(2),..., N(i).

On aura par exemple :

```
10 DIM N(2)
20 N(1)=10
30 N(2)=20
40 CSAVE* N,"TOTO"
```

SKIPF

Cette commande permet de positionner la cassette à la fin du programme précisé.

La syntaxe est la suivante :

```
SKIPF "nom de fichier"
```

**Remarque** : cette commande permet également de vérifier que l'enregistrement du programme spécifié dans le "nom de fichier" est correct.

CONT

Cette commande permet de relancer l'exécution d'un programme stoppé préalablement par une instruction STOP ou par la pression sur la touche BREAK.

Le programme continue alors à partir de la ligne où il s'était arrêté. L'utilisation conjointe de l'instruction STOP et de la commande CONT rend possible la mise au point de programmes. En effet, il est possible d'examiner la valeur de variables utilisées par le programme lorsque celui-ci est arrêté.

**Remarque** : un fonctionnement similaire peut être obtenu en pressant simultanément sur les touches SHIFT et @ , ce qui provoque un arrêt dans le déroulement du programme. Dans ce cas, il n'a pas d'affichage du message "BREAK IN N", comme nous le verrons dans la description de l'instruction STOP.

L'exécution peut reprendre par une simple pression sur n'importe quelle touche du clavier.

LIST ET LLIST

Ces commandes provoquent l'affichage de tout ou partie de votre programme Basic.

La commande LIST permet d'obtenir un affichage sur écran.
La commande LLIST permet d'obtenir un affichage sur imprimante.

La syntaxe est la suivante :

```
        LIST   (numéro ligne début)-(numéro ligne fin)
ou bien :
        LLIST  (numéro ligne début)-(numéro ligne fin)
```

Ainsi :

| | |
|---|---|
| LIST | provoque l'affichage de la totalité du programme. |
| LIST 2ØØ | provoque l'affichage de la ligne 2ØØ uniquement. |
| LIST 1ØØ–2ØØ | provoque l'affichage des lignes 1ØØ à 2ØØ. |
| LIST –2ØØ | provoque l'affichage du programme jusqu'à la ligne 2ØØ. |
| LIST 2ØØ– | provoque l'affichage du programme à partir de la la ligne 2ØØ. |

et de même dans le cas de l'instruction LLIST.

NEW

Cette commande détruit le programme Basic qui se trouve en mémoire. En fait, elle ne touche pas au contenu de la mémoire mais se contente de réinitialiser certains pointeurs de travail utilisés par l'Alice.

Cette commande peut être utilisée en mode programme, sachant qu'elle provoquera l'effacement de tout ce qui se trouve en mémoire au moment de son exécution.

RUN

Cette commande permet de lancer l'exécution d'un programme.

La syntaxe est la suivante :

RUN (numéro de ligne)

Ainsi RUN permet le lancement du programme résidant en mémoire à partir de la première ligne.

RUN 1ØØ permet son lancement à partir de la ligne 1ØØ.

## 2.5. LES INSTRUCTIONS BASIC

### 2.5.1. Les instructions arithmétiques

Nous les avons regroupées sous la forme d'un tableau (p. 11).

| *Nom* | *Fonction réalisée* | *Condition sur l'argument X* |
|---|---|---|
| ABS(X) | Valeur absolue de X | aucune |
| COS(X) | Cosinus de X | X en radians |
| EXP(X) | Exponentielle de X | X<88.Ø3 |
| INT(X) | Partie entière de X | aucune |
| LOG(X) | Logarithme népérien de X | X>Ø |
| RND(X) | Génération d'un nombre aléatoire compris entre Ø et X (ou Ø et 1 pour RND(Ø)) | aucune |
| SGN(X) | signe de X : +1 si X>Ø<br>−1 si X<Ø<br>Ø si X=Ø | aucune |
| SIN(X) | Sinus de X | X en radians |
| SQR(X) | Racine carrée de X | X>=Ø |
| TAN(X) | Tangente de X | X = PI/2 + K * PI, K entier |

### 2.5.2. Les instructions logiques

Ces instructions ont déjà été rencontrées dans le paragraphe consacré aux opérateurs Basic.

Il s'agit des instructions :

```
AND
OR
NOT
```

*Exemple* :

```
Z= X AND Y
Z= NOT X
Z= X OR Y
```

### 2.5.3. Les instructions sur les chaînes de caractères

ASC

Cette instruction permet de renvoyer le code ASCII du premier caractère d'une chaîne X$.

La syntaxe est la suivante :

```
ASC(X$)
```

*Exemple* :

```
10 X$ = "ALICE"
20 PRINT ASC(X$)
```

et le résultat affiché sur l'écran sera donc 65 (code ASCII de la lettre A).

Notons que le nombre renvoyé par une instruction ASC est toujours compris entre Ø et 255.

CHR

Cette instruction réalise l'opération inverse de ASC.

En effet, elle permet de transformer un code ASCII en son caractère équivalent.

La syntaxe est la suivante :

```
CHR$(X)
```

On écrira par exemple PRINT CHR$(13) qui provoque un retour chariot (ENTER) lors d'un affichage sur écran par exemple.

LEFT

Cette instruction permet de renvoyer les I premiers caractères d'une chaîne X$.

La syntaxe est la suivante :

```
LEFT$(X$, I)
```

*Exemple* : Si l'on tape :

PRINT LEFT$(X$,2) avec X$ = "ALICE", on obtiendra la chaîne de caractères "AL".

Notons que I doit toujours être inférieur à 255 et que si I est supé-

rieur à la longueur de la chaîne de caractères X$, celle-ci sera entièrement renvoyée comme résultat.

LEN

Cette instruction renvoie la longueur d'une chaîne de caractères.

La syntaxe est la suivante:

```
LEN(X$)
```

*Exemple*:

```
10 X$= "ALICE"
20 PRINT LEN(X$)
```

donnera un résultat égal à 5.

MID

Cette instruction permet de fournir les J caractères d'une chaîne à partir du Ième.

La syntaxe est la suivante:

```
MID$(X$, I, J)
```

*Exemple*:

```
10 X$= "ALICE"
20 PRINT MID$(X$, 2, 3)
```

donnera en résultat la chaîne "LIC".

Notons que I et J doivent être inférieurs à 255. Si I>LEN(X$) ou si J=Ø le résultat obtenu est une chaîne de caractères "vide".

RIGHT

Cette instruction permet de renvoyer les I derniers caractères d'une chaîne.

La syntaxe est la suivante:

```
RIGHT$(X$, I)
```

*Exemple*:

```
10 X$="ALICE"
20 PRINT RIGHT$(X$,2)
```

donnera la chaîne "CE".

STR

Cette instruction permet de transformer un nombre en une chaîne dont les caractères sont ceux utilisés pour son affichage sur l'écran.

La syntaxe est la suivante:

```
STR$(X)
```

*Exemple*:

```
10 X=1.25E3
20 Y$=STR$(X)
30 PRINT Y$
```

donnera le résultat 125Ø.

VAL

Cette instruction réalise l'opération inverse de STR. En effet, elle renvoie la valeur numérique représentant la donnée présente dans la chaîne de caractères.

La syntaxe est la suivante:

```
VAL(X$)
```

*Exemple*:

```
10 INPUT X$
20 IF VAL(X$))75001 AND VAL(X$)(75020
   THEN PRINT "PARIS  ";RIGHT$(X$,2)
```

Notons que si le premier caractère de la chaîne **X$** n'est pas un chiffre ou l'un des signes + ou —, le résultat obtenu sera Ø.

### 2.5.4. Les instructions de branchement

END

Cette instruction marque la fin d'un programme Basic et provoque l'affichage du OK.

Notons que cette instruction n'est pas nécessaire car le programme s'arrête automatiquement à la dernière ligne exécutable.

EXEC

Cette instruction permet de lancer l'exécution d'une routine écrite en Assembleur et se terminant par une instruction de retour de sous-programme RTS (l'équivalent du RETURN Basic).

La syntaxe de cette instruction est la suivante:

EXEC A

A désignant l'adresse de début de la routine concernée avec:

$O \leqslant A \leqslant 65535$

GOSUB

Il s'agit de l'instruction d'appel de sous-programme.

La syntaxe est la suivante:

GOSUB numéro de ligne

*Exemple*:

10 GOSUB 100

provoque un appel du sous-programme commençant à la ligne 100 et se terminant par une instruction RETURN lorsque le programme exécute la ligne numéro 10.

GOTO

Il s'agit de l'instruction de branchement inconditionnel classique.

La syntaxe est la suivante :

GOTO numéro de ligne

*Exemple* :

GOTO 1ØØ

provoque un branchement à la ligne 1ØØ lorsque le programme exécute la ligne numéro 1Ø.

IF... THEN

Il s'agit de l'instruction de branchement conditionnel au numéro de ligne indiqué.

La syntaxe est la suivante :

IF expression THEN numéro de ligne

ou bien IF expression GOTO numéro de ligne

ou bien IF expression THEN suite d'instructions Basic

Si l'expression présente après le "IF" est vraie alors il y a soit branchement au numéro de ligne indiqué soit exécution de la suite d'instructions présentes après le "THEN".

L'expression peut exprimer :

— une condition simple :

*Exemple* :

A > Ø

– une condition sous-entendue :

*Exemple* :

A OR B (la condition sous-entendue est A OR B = 1 puisqu'il s'agit d'une opération logique).

— plusieurs conditions :

*Exemple* :

```
A=2Ø AND B=13
```

*Exemple d'utilisation* :

```
1Ø INPUT N
2Ø IF N>Ø THEN GOTO 4Ø
3Ø N=ABS(N)
4Ø PRINT SQR(N)
5Ø END
```

ou bien :

```
1Ø INPUT N
2Ø IF N<Ø THEN N=ABS(N)
3Ø PRINT SQR(N)
4Ø END
```

ON

Cette instruction s'utilise avec les instructions GOTO et GOSUB. Elle permet des branchements ou appels de sous-programmes "calculés" comme nous allons le voir ci-dessous.

La syntaxe est la suivante :

ON variable GOTO numéro de ligne 1 (,numéro de ligne 2, etc...)
ON variable GOSUB numéro de ligne 1 (,numéro de ligne 2, etc...)

Selon la valeur de la variable, il y a branchement à une ligne Basic ou appel du sous-programme commençant à cette ligne.

Si variable=1, il y a branchement au premier numéro de ligne indiqué, si variable=2, on se branche au deuxième, etc... Le programme suivant illustre le fonctionnement de ON... GOTO :

```
10 INPUT A
20 ON A GOTO 30,50,70
30 PRINT "LIGNE 30"
40 GOTO 10
50 PRINT "LIGNE 50"
60 GOTO 10
70 PRINT "LIGNE 70"
80 GOTO 10
90 END
```

Le fonctionnement de ON... GOSUB est en tous points similaire.

RETURN

Comme nous l'avons vu plus haut, il s'agit de l'instruction de retour de sous-programme.

STOP

Cette instruction provoque l'arrêt du programme Basic et l'affichage de BREAK IN N, N étant le numéro de la ligne contenant l'instruction STOP.

Cette instruction ne détruit pas les variables. Pour relancer le programme à partir de la ligne où se trouve le STOP, il suffit de taper au clavier la commande CONT vue précédemment.

### 2.5.5. Les instructions d'Entrées-Sorties

INKEY

Cette instruction permet de rentrer un caractère au vol à partir du clavier sans avoir à taper sur la touche ENTER. Cette instruction sera en général incluse dans une boucle d'attente.

La syntaxe est la suivante :

INKEY$

*Exemple 1* :

```
10 A$ = INKEY$
20 IF A$ = "M" THEN PRINT "BONJOUR"
30 IF A$ = "S" THEN PRINT "BONSOIR"
40 GOTO 10
```

*Exemple 2* :

```
10 FOR I=1 TO 5
20 A$= INKEY$
30 IF A$= "" THEN 20
40 NEXT
50 END
```

INPUT

Cette instruction permet de rentrer la valeur de variables à partir du clavier.

La syntaxe est la suivante :

INPUT ("TEXTE";) VARIABLE 1 (,VARIABLE 2,...,VARIABLE N)

*Exemple* :

```
10 INPUT "NOMBRE  ";N
20 INPUT "ANIMAL  ";A$
30 PRINT A$;N
```

Le fonctionnement de l'instruction INPUT est le suivant :

— si aucune touche n'est frappée ou si vous ne tapez pas de ENTER après avoir rentré la donnée, l'Alice reste en état d'attente,

— si une erreur sur le type de variable est faite (exemple : on rentre une chaîne de caractères alors que la variable a été déclarée comme réelle), le Basic renvoie le message REDØ ? invitant l'utilisateur à rentrer à nouveau la variable.

— pour rentrer plusieurs données, il faut utiliser une virgule (",") pour séparer chacune d'elles,

— si toutes les données nécessaires n'ont pas été introduites, vous voyez apparaître sur l'écran un double point d'interrogation ("??").

Notons que l'instruction INPUT ne peut être utilisée qu'en mode programme.

PRINT ET LPRINT

Ces instructions provoquent l'affichage sur l'écran (cas de l'instruction PRINT) ou l'imprimante (cas de l'instruction LPRINT) des textes et valeurs de variables précises juste après.

La syntaxe est la suivante :

```
PRINT VARIABLE 1(,VARIABLE 2,...,VARIABLE N)
```

ou

```
LPRINT VARIABLE 1(,VARIABLE 2,...,VARIABLE N)
```

La syntaxe ci-dessus est bien sûr valable pour des textes qui devront être donnés entre guillemets.

La virgule séparatrice (",") peut être remplacée par un point-virgule (";"). Le formatage sera alors différent.

Dans le cas d'une virgule séparatrice, l'affichage des différentes variables se fera tous les 16 caractères. Par contre la présence d'un point-virgule provoquera la juxtaposition de ces variables.

Notons que l'instruction PRINT (ou bien LPRINT) utilisée seule provoque un retour à la ligne sans autre affichage.

PRINT TAB

Cette instruction permet de réaliser une tabulation.

La syntaxe est la suivante :

```
PRINT TAB(X)
```

*Exemple* : PRINT TAB(2Ø) permet de positionner le curseur à la 2Ø$^{e}$ colonne de l'écran (celles-ci étant numérotées de Ø à 79 dans le cas d'un affichage 8Ø colonnes, par exemple).

Nota : X doit toujours être inférieur à 255.

PRINT @

Cette instruction permet d'afficher des textes ou valeurs de variables à partir d'une position déterminée sur l'écran.

La syntaxe est la suivante :

```
PRINT @X, VARIABLE1 (, VARIABLE2, . . . , VARIABLE N)
```

X est un nombre compris entre :

— Ø et 511 dans le cas d'un affichage sur 32 colonnes,

— Ø et 999 dans le cas d'un affichage sur 4Ø colonnes,

— Ø et 1999 dans le cas d'un affichage sur 8Ø colonnes.

En effet, ces valeurs représentent le nombre maximal de caractères qui peuvent être affichés sur l'écran dans le mode d'affichage choisi.

Par exemple, pour commencer l'impression à la 6e ligne, 10e colonne, on écrira, dans le cas d'un affichage sur 4Ø colonnes :

```
PRINT @25Ø, variable1, . . . , variableN
```

En effet : 25Ø = 6 * 4Ø + 1Ø

### 2.5.6. Les instructions graphiques et sonores

Le rôle succinct des diverses instructions graphiques est décrit ici. Nous reviendrons en détail sur les possibilités graphiques de l'Alice dans le chapitre 8 avec, en particulier, l'accès à partir de l'Assembleur.

CLS

Cette instruction permet d'accomplir deux tâches.

La première permet de sélectionner le mode d'affichage choisi :

L'instruction CLS 32 sélectionne le mode sur 32 colonnes.

L'instruction CLS 4Ø sélectionne le mode sur 4Ø colonnes.

L'instruction CLS 8Ø sélectionne le mode sur 8Ø colonnes.

L'instruction CLS 81 sélectionne le mode sur 8Ø colonnes en vidéo inverse.

La deuxième permet d'effacer l'écran et de choisir la couleur de fond (dans le cas des modes d'affichage sur 32 et 4Ø colonnes uniquement).

Dans le cas de l'instruction CLS, on obtiendra un écran vert.

Dans le cas de l'instruction CLS N, on obtiendra un écran dont la couleur est déterminée par la valeur de N.

Le tableau suivant résume les couleurs possibles:

| *Couleur* | *Valeur de N* |
|---|---|
| noir | Ø |
| vert | 1 |
| jaune | 2 |
| bleu roi | 3 |
| rouge | 4 |
| ivoire | 5 |
| bleu pâle | 6 |
| mauve | 7 |
| orange | 8 |

POINT

Cette instruction fait partie des trois instructions semi-graphiques disponibles sur l'Alice.

Elle permet de savoir si un point situé sur l'écran est allumé ou non et quel est le type d'information qu'il contient (information alphanumérique, semi-graphique, couleur).

La syntaxe de cette instruction est la suivante:

```
POINT(X,Y)
```

Les coordonnées du point considéré sont X et Y. X est l'abscisse et peut donc varier entre Ø et 31, 39 ou 79 selon le mode d'affichage choisi.

L'ordonnée est Y et peut donc varier entre Ø et 15 ou 24 selon le mode d'affichage choisi.

Le tableau suivant résume les informations retournées par l'instruction POINT en fonction du mode d'affichage choisi.

| *Mode d'affichage* | *Valeur* | *Information* |
|---|---|---|
| 32 ou 4Ø colonnes | Ø | point éteint |
| 32 ou 4Ø colonnes | −1 | caractère alphanumérique |
| 32 ou 4Ø colonnes | 1 à 8 | couleur du point |
| 8Ø colonnes | Ø | couleur de marge |
| 8Ø colonnes | −1 | caractère alphanumérique |
| 8Ø colonnes | 1 | couleur d'intensité |
| 8Ø colonnes | 1 | couleur de demi-intensité |

RESET

Cette instruction permet d'éteindre un point de coordonnées données situé sur l'écran.

La syntaxe est la suivante:

```
RESET(X,Y)
```

X et Y étant définis comme dans l'instruction POINT vue précédemment.

En mode 32 ou 4Ø colonnes, le point concerné prend donc la couleur du fond de l'écran après une instruction RESET.

Dans le cas de l'affichage sur 8Ø colonnes, la couleur prise par le point sera la couleur de marge.

SET

Cette instruction permet d'allumer un point de coordonnées données, ceci avec la couleur désirée.

La syntaxe de cette instruction est la suivante:

```
SET(X,Y,N)
```

X et Y ont été décrits précédemment et donnent la coordonnée du point concerné.

N spécifie la couleur du point à allumer.

Dans le cas d'un affichage sur 32 ou 4Ø colonnes, la couleur N peut varier entre Ø et 8 tandis que dans le cas d'un affichage sur 8Ø colonnes, N varie entre Ø et 2.

Ces différentes valeurs de N, dans le cas d'un affichage sur 8Ø colonnes, ont les significations suivantes:

— Ø est relatif à la couleur de marge (couleur du cadre de l'écran ainsi que des caractères alphanumériques affichés),

— 1 est relatif à la couleur d'intensité (couleur du fond de l'écran),

— 2 est relatif à la couleur de demi-intensité (couleur d'affichage de chaque point élémentaire).

Ces différentes couleurs sont spécifiées une fois pour toutes grâce à l'instruction SET* que nous allons décrire ci-dessous.

SET*

Cette instruction est utile uniquement dans les modes d'affichage sur 8Ø colonnes sélectionnés grâce aux instructions CLS 8Ø et CLS 81. Elle permet de sélectionner les couleurs de marge, d'intensité et de demi-intensité.

Chacune de ces couleurs est donnée pour un nombre compris entre Ø et 8. La syntaxe est la suivante:

SET* M,I,D

M désignant la couleur de marge,

I désignant la couleur d'intensité,

D désignant la couleur de demi-intensité.

Nous allons illustrer cette instruction ainsi que l'instruction SET par des exemples d'application.

Nous vous proposons tout d'abord de tracer sur l'écran la courbe:

$$Y = \frac{\sin(X)}{X}$$

Le programme est le suivant:

```
30 INPUT "INTERVALLE D'ETUDE";X1,X2
40 PA=(X2-X1)/160
50 MI=1.7E38:MA=-1.7E38
60 FOR X=0 TO 159
70 X3=X1+X*PA
80 IF X3=0 THEN Y=1:GOTO100
90 Y=SIN(X3)/X3
100 IF Y<MI THEN MI=Y
110 IF Y>MA THEN MA=Y
120 NEXT
130 B=MA*124/(MA-MI):A=124/(MI-MA)
140 SET* 2,2,4
150 FOR X=0 TO 159
160 X3=X1+X*PA
170 IF X3=0 THEN Y=INT(A+B):GOTO 190
180 Y=INT(B-A*SIN(X3)/X3)
190 SET(X,Y,2)
200 NEXT
210 GOTO 210
```

Après avoir rentré ce programme, tapez CLS 8Ø puir RUN. Entrez l'intervalle d'étude (par exemple 1,1Ø). Après quelques secondes, vous verrez s'afficher la courbe $\frac{\sin(X)}{X}$ en rouge sur fond jaune.

Nous allons expliciter brièvement le fonctionnement de ce programme.

— la ligne 3Ø permet de rentrer l'intervalle d'étude de cette fonction (valeurs minimale et maximale de l'abscisse).

— la ligne 4Ø calcule le "pas", c'est-à-dire la distance selon l'axe des X entre deux points étudiés, sachant que la résolution de l'affichage est 16Ø.

— les lignes 4Ø à 12Ø permettent de calculer les minimum et maximum de la fonction $\frac{\sin(X)}{X}$ sur l'intervalle d'étude, ceci afin d'obtenir une mise à l'échelle automatique.

— la ligne 8Ø (ainsi que la ligne 17Ø) tiennent compte du cas particulier $\frac{\sin(O)}{O} = 1$.

— la ligne 14Ø permet de sélectionner les couleurs de marge, d'intensité et de demi-intensité.

La couleur de marge est ici le jaune (code=2) de même que la couleur d'intensité, ce qui permet d'avoir des caractères alphanumériques invisibles et de ne conserver que l'affichage graphique haute-résolution (de couleur rouge ici).

— les lignes 15Ø à 2ØØ permettent de tracer la courbe proprement dite.

Notons que ce programme est quasi-universel. Il peut être utilisé pour tracer toutes sortes de courbes. Il vous suffira de modifier les lignes 9Ø et 18Ø ainsi que les lignes 8Ø et 17Ø qui traitent les cas particuliers.

Le petit programme présenté ci-dessous permet de tracer une série de cercles de rayon croissant arrangés de façon à former un cône.

```
10 SET* 2,2,4:X1=10:Y1=100
20 FOR R=10 TO 50 STEP 5
30 FOR AL=0 TO 2*3.14159 STEP 0.02
40 X=X1+R*COS(AL)
50 Y=Y1+R*SIN(AL)
60 SET(X,Y,2):NEXT
70 X1=X1+15:Y1=Y1-5
80 NEXT
```

Les lignes 3Ø à 6Ø permettent de tracer un cercle de rayon R, de centre (X1,Y1).

L'équation du cercle est donnée en polaires :

$X = X1 + R\cos(AL)$

$Y = Y1 + R\sin(AL)$

SOUND

Cette instruction est la seule instruction sonore présente sur l'Alice.

Elle permet de générer une note musicale de durée et de fréquence

variables dans le haut-parleur présent sur votre téléviseur.

La syntaxe est la suivante :

```
SOUND F, D
```

F permet de programmer la fréquence de la note musicale et est un nombre compris entre Ø et 255.

D permet de programmer la durée de la note musicale et est également un nombre compris entre Ø et 255.

Le petit programme suivant illustre le fonctionnement de cette instruction et permet de jouer un air bien connu.

```
10 FOR I=1 TO 11
20 READ F,DU
30 SOUND F,DU
35 FOR J=1 TO 50:NEXT
40 NEXT
50 DATA 180,5,180,5,180,5,189,5,196,10
60 DATA 189,10,180,5,196,5,189,5,189,5,180,10
70 GOTO 70
```

A chaque note est associée une fréquence (18Ø = DO, 189 = RE, 196 = MI) et une durée.

### 2.5.7. Les autres instructions

CLEAR

Cette instruction joue trois rôles différents selon la syntaxe utilisée.

— l'instruction CLEAR permet de remettre à zéro toutes les variables et chaînes de caractères.

— l'instruction CLEAR N permet de réserver un espace de N/2 cases-mémoires pour y stocker des chaînes de caractères.

En effet, la taille maximale d'une chaîne de caractères rentrée par une instruction INPUT est égale à 127. Dans certains cas, il peut être nécessaire de limiter cette valeur à un nombre inférieur (pour économiser

la place mémoire) ou au contraire d'augmenter cette valeur (jusqu'à une limite de 255 caractères).

On écrira donc CLEAR N avec N compris entre Ø et 51Ø.

— l'instruction CLEAR N1,N2 est utilisée avec des programmes écrits en Assembleur.

N1 permet de déterminer le nombre d'octets mémoire qui seront utilisés pour le stockage de fichiers source. N2 permet de sélectionner l'adresse de départ du fichier source.

C'est ainsi que CLEAR 1ØØ,2ØØØØ permet de réserver 1ØØ cases-mémoires à partir de l'adresse décimale 2ØØØØ.

DATA

Cette instruction permet le stockage des données à l'intérieur d'un programme Basic.

La syntaxe est la suivante:

DATA valeur1(,valeur2,...,valeurN)

Elle peut être mise à n'importe quel endroit et est utilisée conjointement avec l'instruction READ.

Dans une ligne DATA, on peut stocker n'importe quoi (variable, chaîne de caractères) en séparant chaque donnée par une virgule.

DIM

Il s'agit de l'instruction de déclaration de tableau.

La syntaxe est la suivante:

DIM variable1 (dim1,dim2) (,variable2(...),...,variableN(...)))

Les variables variable1 et variable2 peuvent être de type réel ou de type chaîne ($). Un tableau non déclaré par une instruction DIM est automatiquement dimensionné à dix dans chaque dimension.

Notons qu'un tableau ne peut être dimensionné qu'une fois au cours d'un programme.

*Exemple*: un tableau déclaré par DIM A(2,1) contiendra les six éléments suivants:

```
A(0,0)
A(1,0)
A(2,0)
A(0,1)
A(1,1)
A(2,1)
```

Notons que le Basic de l'Alice ne supporte pas de tableaux comprenant plus de deux dimensions.

FOR... TO... STEP

Cette instruction est la classique boucle FOR... NEXT.

La syntaxe est la suivante:

```
FOR variable=debut TO fin (STEP pas)
```

Un compteur de boucle est initialisé à la valeur "début" et incrémenté à chaque exécution de boucle de la valeur "pas" jusqu'à ce qu'il atteigne la valeur "fin". Si aucun pas n'est précisé (pas de STEP), l'incrément est égal à 1.

**Remarque**: une boucle FOR... NEXT est toujours exécutée au moins une fois, quelles que soient les valeurs "début" et "fin".

*Exemple*:

```
10 FOR I=1 TO 10
20 PRINT I*4
30 NEXT I
```

LET

Cette instruction est utilisée pour affecter une valeur à une variable.

La syntaxe est la suivante:

```
LET variable= expression
```

On écrira par exemple LET A=1Ø. En fait, cette instruction est inutile et ne sert qu'à gaspiller de l'espace mémoire. On écrira à la place A=1Ø tout simplement.

MEM

Cette instruction permet de connaître à tout instant le nombre d'octets de mémoire vive restant dans l'Alice.

On écrira :

```
PRINT MEM
```

PEEK

Cette instruction va lire le contenu de la case-mémoire d'adresse X. La syntaxe est donc la suivante :

```
PEEK(X)    avec    Ø <= X  <=65535
```

*Exemple* :

```
Y= PEEK(X)
```

ou bien

```
PRINT PEEK(X)
```

Le résultat est donné sous forme décimale et est donc compris entre Ø et 255.

POKE

Cette instruction permet de ranger une valeur dans une case-mémoire déterminée.

La syntaxe est la suivante :

```
POKE adresse,valeur
```

et on a :

```
Ø <=adresse <=65535
Ø <=valeur  <=255
```

*Exemple*: POKE 2000,45

READ

Cette instruction permet d'assigner des valeurs à des variables au cours de l'exécution d'un programme. Elle est utilisée conjointement à l'instruction DATA.

La syntaxe est la suivante:

```
READ variable1(,variable2,...,variableN)
```

*Exemple*:

```
10 DATA 10,20,30,"VACHE"
20 READ A,B,C,D$
```

A A sera donc effectée la valeur 1Ø, à B la valeur 2Ø, à C la valeur 3Ø et à D$ la chaîne de caractères VACHE.

Un même variable peut accéder à différentes données et à différentes lignes DATA.

*Exemple*:

```
10 DATA 1,2,3,4,5
20 DATA 10,20,30,40,50
30 FOR I=1 TO 10
40 READ A
50 PRINT A
60 NEXT I
```

En effet, à chaque instruction READ, un pointeur est incrémenté et va ainsi pointer sur la donnée suivante. Ce pointeur peut être réinitialisé à l'aide de l'instruction RESTORE comme précisé dans les prochaines lignes.

Notons que le nombre de "READ" entre deux "RESTORE" doit toujours être inférieur ou égal au nombre d'éléments présents sur l'ensemble des lignes DATA qu comporte le programme Basic.

REM

L'instruction REM indique une ligne de commentaire.

*Exemple*:

```
10 REM PROGRAMME DE JEU
```

Lors de l'exécution du programme, l'interpréteur saute une ligne qui commence par un REM et passe à la ligne suivante.

RESTORE

Cette instruction, utilisée conjointement avec les instructions DATA et READ permet de remettre à zéro le pointeur de DATA (il pointe alors sur le premier élément de la première ligne DATA présenté dans le programme).

Le programme ci-dessous illustre le fonctionnement des instructions DATA, READ, RESTORE et permet de connaître le jour de la semaine en fonction de la date:

```
10 INPUT J,M,A
20 PRINT "LE";J;"";M;"";A;"EST UN"
30 IF M<=2 THEN 50
40 N=0:GOTO90
50 IF A-4*INT(X/4)=0 THEN 70
60 N=2:GOTO90
70 IF A=0 THEN 60
80 N=1
90 C=INT(365.25*A)+INT(30.56*M)+J+N
100 I=C-7*INT(C/7)
110 FOR J=1 TO I+1:READ A$
120 NEXT J
130 PRINT A$
140 RESTORE
150 DATA "MERCREDI","JEUDI","VENDREDI","SAMEDI
160 DATA "DIMANCHE","LUNDI","MARDI"
170 GOTO 10
```

Après avoir introduit ce programme, tapez RUN puis entrez le jour, le mois et l'année ($\leqslant$99), par exemple 5, 3, 85 pour le 5 mars 1985.

VARPTR

Cette instruction permet de connaître l'adresse d'implantation en mémoire d'une variable.

On écrira par exemple, apès un NEW :

```
10 A=1
20 PRINT VARPTR(A)
```

ce qui donne le résultat 13149.

Cette instruction ne vous sera probablement d'aucune utilité sauf si vous désirez traiter en assembleur des variables utilisées par l'interpréteur Basic.

# 3

# L'architecture du système

L'Alice est ce que l'on appelle un "micro-ordinateur" individuel. A ce titre, il comporte tous les composants nécessaires à son fonctionnement en tant que système autonome (mis à part l'écran bien sûr). Le but de ce chapitre est de vous faire comprendre le fonctionnement général de votre machine sans rentrer dans l'"électronique" à proprement dit.

## 3.1. L'ARCHITECTURE

L'Alice est bâti, comme tout micro-ordinateur, autour de ce cafard appelé micro-processeur. Dans votre cas, il s'agit du 68Ø3, dérivé du 68ØØ de MOTOROLA qui fut un des premiers microprocesseurs à apparaître au début des années 7Ø.

En fait le 68Ø3 est plus qu'un simple microprocesseur puisqu'il intègre sur la même "puce" (circuit intégré) un grand nombre de lignes d'Entrées-Sorties, un Timer, 128 octets de mémoire vive et un interface série, en plus bien sûr du microprocesseur proprement dit.

Nous reviendrons sur ces notions un peu plus loin.

Le 68Ø3 possède comme tout microprocesseur, un bus d'adresses, un bus de données ainsi qu'un bus de contrôle.

Ces bus permettent d'accéder à la mémoire contenant des programmes (en Basic ou en langage machine). Celle-ci peut être divisée en deux groupes, la mémoire morte (ROM = Read Only Memory) et la mémoire volatile (RAM = Random Access Memory). La première, comme son nom l'indique, ne peut être que lue et contient le logiciel de l'Alice (gestion du clavier, de l'écran, de l'Interface cassette, etc...), le Basic et l'Éditeur-Assembleur.

La seconde contient d'une part les programmes utilisateurs (programmes Basic ou en langage machine), d'autre part la mémoire écran qui contient les codes de tous les caractères affichés.

Nous n'allons pas, nous l'avons déjà dit, aborder la façon dont le microprocesseur lit et écrit dans la mémoire, ni comment s'effectue la synchronisation et l'enchaînement de toutes les opérations élémentaires nécessaires à l'exécution d'une tâche plus complexe (comme par exemple aller afficher un caractère sur l'écran).

Pour satisfaire votre curiosité et en apprendre plus sur le sujet, reportez-vous plutôt à des ouvrages spécialisés traitant de cette question (et notamment au livre "Micro-ordinateurs, comment ça marche?", paru dans la même collection). Nous nous contenterons ici de décrire les schémas de principe du système.

En plus de la mémoire dont nous venons de parler, il existe un circuit spécialisé ayant pour référence EF9345 qui n'est autre que le contrôleur d'écran dont le rôle est de générer tous les signaux nécessaires à l'affichage sur moniteur ou télévision, ceci de manière totalement (ou presque) transparente pour le microprocesseur. En mode alphanumérique, la mémoire écran contient le code caractère à afficher et ce code est présenté à l'entrée d'une mémoire appelée générateur de caractère. Cette mémoire peut être de type RAM ou ROM. En fonctionnement standard, le générateur de caractères est situé dans une ROM contenue dans le circuit intégré contrôleur d'écran. Celui-ci se charge alors de générer, à partir des informations contenues dans cette ROM, le signal RVB qui est ensuite envoyé sur votre téléviseur par l'intermédiaire du câble PERITEL. Mais revenons sur le microprocesseur de type 68Ø3.

Il possède un certain nombre de lignes d'Entrées-Sorties (29 pour être précis) dont le rôle est de lui permettre de communiquer aisément

avec le monde extérieur (nous appellerons Entrée toute opération durant laquelle le microprocesseur acquiert une information de l'extérieur, et Sortie toute opération durant laquelle le microprocesseur envoie une donnée vers l'extérieur).

Par exemple le lecteur de cassettes, l'écran sont des organes de Sortie, tandis que le clavier constitue un organe d'Entrée.

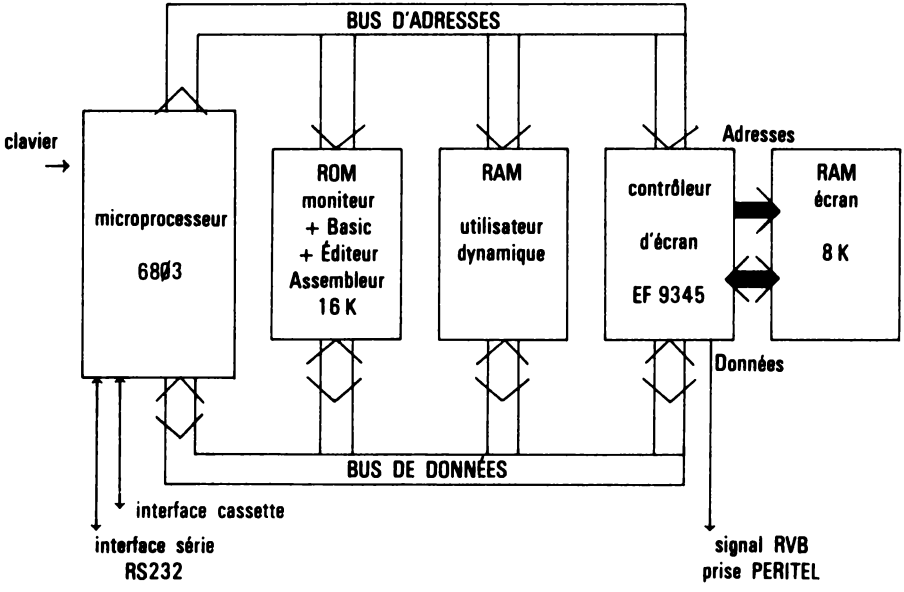
Le rôle principal joué par les lignes d'Entrées-Sorties du 68Ø3 est la gestion du clavier de l'Alice. Les touches de celui-ci sont arrangées sous forme de matrice (8 lignes sur 7 colonnes).

Lorsqu'une touche est pressée, le code correspondant est chargé dans le 68Ø3 qui l'envoie ensuite vers le contrôleur d'écran en vue de son affichage.

Les 128 octets de mémoire vive contenus dans le 68Ø3 sont destinés à stocker des variables système importantes. Ils constituent la zone de travail du moniteur et du Basic.

L'interface série, lui, permet les liaisons avec imprimante via la prise RS232 ainsi que la gestion de l'interface cassette.

Le schéma de principe de l'Alice est le suivant:



Nous remarquons sur ce schéma de principe que la mémoire écran est séparée de la mémoire utilisateur destinée à stocker des programmes Basic ou Assembleur. Cette mémoire écran est entièrement gérée par le contrôleur de type EF 9345 dont le rôle, comme nous l'avons dit précédemment, est d'aller lire ou écrire dans cette RAM les informations qui seront ensuite visualisées sur votre téléviseur.

La RAM utilisateur a une capacité différente selon qu'il s'agit d'un Alice 32 ou d'un Alice 9Ø.

L'Alice 32 possède 8 Koctets de RAM utilisateur tandis que l'Alice 9Ø en possède 32 Koctets. Nous allons maintenant envisager le mapping de l'Alice, c'est-à-dire les emplacements dans l'espace adressable du 68Ø3 de tous les composants dont nous avons parlé jusqu'à maintenant, à savoir mémoire RAM, ROM, circuit contrôleur d'écran.

## 3.2. LE MAPPING

Nous avons dit que le microprocesseur possédait un bus d'adresses. Ce dernier est formé de 16 bits comme c'est le cas de tous les microprocesseurs 8 bits existant actuellement sur le marché. Il peut donc adresser $2^{16} = 65536$ cases différentes comprises entre les adresses $0000 = Ø et $FFFF = 65535.

Nous allons examiner dans ce paragrahe comment est divisé cet espace en fonction des différents composants existant.

### Mapping de l'Alice 9Ø

| *Adresse* | *Fonction* |
|---|---|
| 65535 = $FFFF | ROM moniteur + Basic + Éditeur-Assembleur 16 Koctets |
| 49152 = $CØØØ | Espace inutilisé |
| 45Ø56 = $BØØØ | RAM utilisateur 32 Koctets |

| *Adresse* | *Fonction* |
|---|---|
| 12288 = $3ØØØ | |
| | Espace inutilisé |
| 256 = $Ø1ØØ | |
| | RAM interne du 68Ø3 (128 octets) |
| 128 = $ØØ8Ø | |
| | Espace inutilisé |
| 32 = $ØØ2Ø | |
| | Registres internes au 68Ø3 |
| Ø = $ØØØØ | |

## Mapping de l'Alice 32

| *Adresse* | *Fonction* |
|---|---|
| 65535 = $FFFF | |
| | ROM moniteur + Basic + Éditeur-Assembleur 16 Koctets |
| 49152 = $CØØØ | |
| | Espace inutilisé |
| 2Ø48Ø = $5ØØØ | |
| | RAM utilisateur 8 Koctets |
| 12288 = $3ØØØ | |
| | Espace inutilisé |
| 256 = $Ø1ØØ | |
| | RAM interne du 68Ø3 (128 octets) |
| 128 = $ØØ8Ø | |
| | Espace inutilisé |
| 32 = $ØØ2Ø | |
| | Registres internes au 68Ø3 |
| Ø = $ØØØØ | |

Les Mapping des deux versions de l'Alice sont à peu près similaires et ne diffèrent que par la quantité de mémoire utilisateur disponible.

Les adressees $ØØØØ à $ØØ1F contiennent les registres internes du 68Ø3 destinés à contrôler les ports d'Entrées-Sorties, l'interface série et le Timer (ou temporisateur).

Les adresses $ØØ8Ø à $ØØFF contiennent les 128 octets de RAM interne au 68Ø3 dont nous avons parlé précédemment.

La RAM utilisateur est située à partir de l'adresse $3ØØØ.

La ROM contenant le moniteur, le Basic ainsi que l'Éditeur-Assembleur se trouve en haut de l'espace adressable du 68Ø3, à savoir entre les adresses $CØØØ et $FFFF.

Maintenant que vous connaissez un peu l'architecture de votre micro-ordinateur, nous allons pouvoir passer à un sujet passionnant qui est l'accès au langage-machine avec, tout d'abord, la description du 68Ø3 et de son jeu d'instructions.

# 4

# Introduction au 68Ø3

## 4.1. INTRODUCTION

Dans ce chapitre, nous nous proposons de décrire la structure interne du 68Ø3, son mode de fonctionnement et de vous faire sentir l'attrait de la programmation en Assembleur.

Le 68Ø3 est, nous le rappelons, le microprocesseur équipant l'Alice.

Considérons le petit programme Basic suivant:

```
10 INPUT A
20 IF A<0 THEN GOTO 50
30 B=-A
40 GOTO 60
50 B=A
60 PRINT B
70 END
```

Dans un programme Basic, l'interpréteur lit une ligne, la traduit en instructions machine et l'exécute. De plus, les différentes lignes sont traitées les unes après les autres. Ici, le déroulement est le suivant:

— le microprocesseur attend tout d'abord qu'une valeur soit rentrée au clavier,

— lorsque celle-ci a été introduite, il teste si elle est négative,

— dans l'affirmative, il y a branchement à la ligne 5Ø (il s'agit donc d'un branchement conditionnel), sinon le programme continue à se dérouler normalement à partir de la ligne 3Ø.

Le "5Ø" du GOTO 5Ø, de même que le "6Ø" du GOTO 6Ø sont des étiquettes de branchement.

— si A<Ø alors B= A

— si A>Ø alors B=−A

— le programme se termine par l'impression de la valeur prise par B.

Comme vous avez pu le constater, tout cela n'a rien de sorcier. La seule chose à toujours bien garder en tête, c'est la tâche que l'on veut faire exécuter à son programme. Le reste n'est finalement qu'une question d'écriture, que ce soit en Basic, en Fortran ou en Assembleur comme nous allons maintenant le voir.

Considérons le petit programme suivant:

```
        LDAA   VAR     ;CHARGE A
        BLT    NEG     ;SI NEGATIF, B=A
        NEGA           ;SI POSITIF, B=-A
NEG     STAA   AFFI    ;RANGE EN MEMOIRE
        SWI
```

Sous des apparences un peu barbares, ce petit programme effectue quasiment la même tâche que le programme Basic donné précédemment.

Examinons-le un peu sans nous attacher à la signification des symboles tels que "LDAA" etc...

A première vue, nous voyons apparaître quatre zones séparées entre elles par un espace.

1) la zone étiquette: "NEG"; cette étiquette correspond exactement au "5Ø" du programme Basic.

2) la zone instruction: exemple "LDAA". Dans cette zone se trouvent les mnémoniques ou ensembles de lettres correspondant à des instructions exécutables par le microprocesseur. Notons que ces mnémo-

niques doivent être traduits sous forme binaire afin de pouvoir être compris par la machine (nous reviendrons sur ce point un peu plus loin): c'est la phase d'assemblage.

3) la zone opérande: exemple "VAR". Dans cette zone sont rangés divers renseignements concernant les données sur lesquelles s'effectue l'instruction placée sur la même ligne.

4) la zone commentaire: exemple ";SI NEGATIF B=A". Elle n'intervient pas dans le programme exécuté par le microprocesseur mais sert à expliquer le fonctionnement d'une ou de plusieurs lignes du programme. Vous pouvez constater que, hormis la forme qui est un peu différente, l'analogie entre un programme en Assembleur et un programme en Basic est très grande.

— une ligne de programme contient toutes les informations nécessaires à son exécution correcte.

— les lignes sont exécutées les unes après les autres.

Examinons de plus près notre programme.

A la première ligne on charge l'Accumulateur A, qui est une case-mémoire particulière située à l'intérieur du 68Ø3, avec la valeur A située initialement à l'adresse VAR.

Grâce à l'instruction BLT, cette valeur est comparée avec zéro et dans le cas où elle est négative, il y a branchement à l'étiquette NEG. La valeur B=A est donc toujours stockée dans l'Accumulateur et est ensuite rangée à l'adresse AFFI. Si par contre la valeur A avait été positive ou nulle, le programme aurait continué normalement.

L'instruction NEGA calcule la valeur B=—A (nous ne rentrerons pas ici dans les détails concernant cette instruction) qui est également stockée dans l'Assembleur puis rangée à l'adresse AFFICH.

Pour effectuer exactement la même tâche que dans le programme Basic, il faudrait rajouter au programme Assembleur deux routines:

— l'une permettant d'afficher sur l'écran cathodique ou sur l'imprimante le contenu de la case-mémoire AFFI.

Ceci dit, la comparaison des deux programmes nous amène à une réflexion: le temps d'exécution du programme Basic sera beaucoup plus long que celui du programme Assembleur. En effet, l'interpréteur Basic

de l'Alice traduit tout d'abord chaque ligne de programme en une série d'instructions "machine" qui sont ensuite exécutées par le microprocesseur.

En pratique, vous utiliserez l'Assembleur :

— chaque fois que le facteur temps d'exécution aura une grande importance : par exemple lorsque vous aurez de longs calculs à effectuer, ou lorsque vous programmerez des jeux animés sur écran.

— lorsque vous aurez besoin de fonctions spécifiques dont le Basic de l'Alice n'est pas doté ; exemple : tracer une droite reliant deux points sur l'écran.

Nous avons jusqu'à présent effectué une approche de l'Assembleur à partir du Basic afin de vous faire sentir qu'après tout ce n'est pas si compliqué que cela en a l'air à première vue.

Nous allons maintenant vous proposer une deuxième approche en remontant aux sources et en partant, cette fois-ci, du microprocesseur.

Nous en profiterons pour expliquer au passage quelques termes courants qui font partie du vocabulaire de l'Assembleur.

Au début était le microprocesseur : "un cafard" à 4Ø pattes qui finalement ne paye pas de mine : on ne soupçonnerait pas, en le voyant, qu'il est capable d'effectuer par exemple plusieurs centaines de milliers d'additions par seconde.

Or, s'il est sûr qu'un microprocesseur est capable de beaucoup de choses (pour ne pas dire des miracles), il faut tout de même se faire à l'idée que c'est un "être" profondément stupide. Ainsi il faut lui mâcher le travail et lui indiquer à chaque instant ce qu'il a à faire car il n'aura jamais d'initiatives personnelles. Le 68Ø3 possède un jeu d'instructions qui lui permet d'accomplir des tâches aussi diverses que chargements mémoire, opérations arithmétiques, comparaisons, branchements, appels de sous-programmes, etc... Mais ne rentrons pas dans les détails, nous verrons cela de manière plus approfondie plus loin.

Afin de voir de quelle façon sont codées ces instructions, il nous faut d'abord rappeler qu'un microprocesseur travaille avec de la mémoire dans laquelle il stocke à la fois les programmes et les données sur lesquelles ceux-ci travaillent. La mémoire est divisée en un certain

nombre de cases, chacune d'elles étant numérotée. Le nombre qui les identifie est appelé "adresse".

Vous savez probablement qu'un ordinateur ne travaille qu'avec des "Ø" ou des "1" (le courant passe ou ne passe pas): c'est ce qu'on appelle le binaire. Un chiffre binaire (donc Ø ou 1) est appelé bit. La mémoire, dans le cas du 68Ø3, est composée de mots de 8 bits en octets qui ne sont en fait que les cases-mémoire dont nous avons parlé ci-dessus. Chacune d'elles reçoit une adresse comprise entre Ø et 65535, ce dernier nombre représentant l'espace-mémoire maximal adressable par le microprocesseur.

Un octet est donc de la forme suivante:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $a_7$ | $a_6$ | $a_5$ | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |

avec $a_i = 1$ ou Ø pour $Ø \leqslant i \leqslant 7$.

Le jeu d'instructions du 68Ø3 est un ensemble d'octets (compris entre Ø et 255 ou bien entre $ØØ et $FF en hexadécimal puisqu'avec 8 bits, il est possible de coder $2^8$ mots différents): on les nomme code-opérations.

Par définition, un programme exécutable par le 68Ø3 sera une série d'instructions associées à leurs opérandes respectives et destinées à accomplir une tâche déterminée.

Ces instructions sont exécutées par le microprocesseur séquentiellement, donc les unes après les autres.

Afin de minimiser les risques d'erreurs liés à un codage manuel en binaire (ou même en hexadécimal), on associe à chaque instruction un mnémonique de trois ou quatre lettres.

*Exemple:* l'addition peut s'écrire ADDA ou bien ADCB. Ainsi ADDA $F823 veut dire additionner le contenu de la case-mémoire d'adresse hexadécimale $F823 avec l'Accumulateur A (nous reviendrons sur le système de numération hexadécimale un peu plus loin).

De plus des noms sont assignés aux différents registres internes du 68Ø3.

*Exemple* :

A = Accumulateur
X = Registre d'index

Le rôle de l'Assembleur contenu dans la ROM de l'Alice est de traduire une suite de mnémoniques 68Ø3 associés à des opérandes en un ensemble d'octets exécutables par le microprocesseur.

L'ensemble de ces mnémoniques est appelé programme source tandis que le résultat, après assemblage, est appelé code objet.

Dans les lignes ci-dessus que, nous l'espérons, vous n'avez pas trouvées trop longues, nous avons essayé de vous faire sentir les avantages de la programmation en Assembleur.

L'Alice possédant un Éditeur-Assembleur incorporé en ROM, vous allez pouvoir aborder sans problème ce domaine passionnant dans les pages qui vont suivre.

## 4.2. LES SYSTÈMES NUMÉRIQUES

### 1) Le binaire

Le 68Ø3, nous l'avons dit, ne comprend que ce type de numération.

De même que le décimal travaille sur les chiffres, Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9 le binaire ne travaille que sur Ø et 1.

On dit que le décimal et le binaire sont respectivement des systèmes de base 1Ø et 2.

Considérons par exemple le nombre 3783. Il est composé de chiffres des milliers (3), des centaines (7), des dizaines (8) et des unités (3).

On peut donc écrire :

$3783 = (3 \times 1ØØØ) + (7 \times 1ØØ) + (8 \times 1Ø) + (3)$

ou bien encore :

$3783 = (3 \times 1Ø^3) + (7 \times 1Ø^2) + (8 \times 1Ø^1) + (3 \times 1Ø^0)$

Si nous nous plaçons maintenant dans un système de base 2, nous avons :

$$3783 = (1 \times 2Ø48) + (1 \times 1Ø24) + (1 \times 512) + (Ø \times 256) + (1 \times 128) + (1 \times 64) + (Ø \times 32) + (Ø \times 16) + (Ø \times 8) + (1 \times 4) + (1 \times 2) + (1 \times 1)$$

ou encore :

$$3783 = (1 \times 2^{11}) + (1 \times 2^{10}) + (1 \times 2^{9}) + (Ø \times 2^{8}) + (1 \times 2^{7}) + (1 \times 2^{6}) + (Ø \times 2^{5}) + (Ø \times 2^{4}) + (Ø \times 2^{3}) + (1 \times 2^{2}) + (1 \times 2^{1}) + (1 \times 2^{0})$$

Le nombre 3783 en base 2 s'écrira donc :

$$3783 = 111Ø11ØØØ111$$

Ce nombre peut donc être représenté en binaire à l'aide d'un mot de 12 bits.

Un simple calcul montre qu'avec 12 bits, il est possible de coder des nombres compris entre Ø et 4Ø95. De même on peut montrer qu'avec N bits, on peut coder des nombres compris entre Ø et $2^N-1$.

Les code-opérations du 68Ø3 étant codés sur un octet, il peut en exister au plus 256. En fait leur nombre est inférieur à cette valeur car ils ne sont pas tous utilisés.

### 2) L'hexadécimal

De même qu'il existe des systèmes numériques de base 1Ø (le décimal) et de base 2 (le binaire) il existe un système numérique de base 16 que l'on nomme l'hexadécimal. Ce système comporte donc 16 caractères qui sont les suivants :

Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

On a donc les équivalences suivantes :

| *Hexadécimal* | *Décimal* | *Binaire* |
|---|---|---|
| Ø | Ø | ØØØØ |
| 1 | 1 | ØØØ1 |
| 2 | 2 | ØØ1Ø |
| 3 | 3 | ØØ11 |
| 4 | 4 | Ø1ØØ |
| 5 | 5 | Ø1Ø1 |
| 6 | 6 | Ø11Ø |
| 7 | 7 | Ø111 |
| 8 | 8 | 1ØØØ |
| 9 | 9 | 1ØØ1 |
| A | 1Ø | 1Ø1Ø |
| B | 11 | 1Ø11 |
| C | 12 | 11ØØ |
| D | 13 | 11Ø1 |
| E | 14 | 111Ø |
| F | 15 | 1111 |

Afin de convertir un nombre binaire en hexadécimal, on le sépare, de droite vers la gauche, en groupes de 4 bits, chacun de ces derniers étant converti en son équivalent hexadécimal.

Par exemple on a :

$\underbrace{Ø11Ø}_{6}\ \underbrace{1ØØ1}_{9}$ = $69 en hexadécimal
= 1Ø5 en décimal

Le signe **$** (dollar) indique que l'on est en présence d'un nombre hexadécimal.

De même, on aura :

$$11ØØ1ØØ11Ø11Ø1 = \underbrace{ØØ11}_{3}\ \underbrace{ØØ1Ø}_{2}\ \underbrace{Ø11Ø}_{6}\ \underbrace{11Ø1}_{D}$$
= $326D en hexadécimal.

### 3) Représentation des nombres négatifs

Jusqu'à présent, nous avons parlé de mots binaires sans en spécifier le signe. De même qu'il existe des nombres décimaux négatifs pourquoi n'existerait-il pas des nombres binaires négatifs ?

Avec un octet il est possible de coder des nombres compris entre Ø et 255 (ou bien $ØØ et $FF en hexadécimal).

Considérons l'opération Ø−1 = −1 en décimal et tentons de la réaliser en binaire (sur un octet).

```
  Ø      ØØØØ ØØØØ
 −1    −ØØØØ ØØØ1
-------------------
=−1     =1111 1111 = $FF
```

Donc −1 sera représenté par $FF.

De même −2 sera représenté par $FF et ainsi de suite.

Cette méthode de détermination de la représentation binaire d'un nombre négatif n'est pas très commode, c'est pourquoi nous allons introduire la notion de notation en complément à 2.

La méthode est la suivante:

— on prend le mot binaire correspondant à la valeur absolue du nombre dont on veut déterminer l'opposé (soit X),

— tous les bits sont changés en leur opposé: (Ø→1 et 1→Ø),

— on ajoute 1 au nombre trouvé, ce qui donne la représentation binaire de (−X).

*Exemple*: Soit à déterminer la représentation binaire −2. On a:

2 = ØØØØØØ1Ø

qui donne:

1111111Ø1 après inversion des 8 bits

et:

1111111Ø = $FE après addition de 1.

Le résultat trouvé est bien le même que précédemment.

Nous pourrons vérifier que l'opération +2−2 donne bien zéro (sur 8 bits).

Nous donnons en annexe un tableau donnant la valeur décimale associée à chacune des valeurs $ØØ à $FF (en notation en complément à 2 bien sûr).

La notation en complément à deux sur 16 bits, utile dans le cas de certaines instructions arithmétiques du 68Ø3 suit exactement le même principe que la notation sur 8 bits vue ci-dessus.

**4) Le code BCD** (Binaire codé Décimal)

Ce code permet de représenter d'une manière simple les nombres décimaux. Chaque chiffre décimal est transformé en son équivalent sur 4 bits. On a donc:

Ø = ØØØØ
1 = ØØØ1
2 = ØØ1Ø
3 = ØØ11
4 = Ø1ØØ
5 = Ø1Ø1
6 = Ø11Ø
7 = Ø111
8 = 1ØØØ
9 = 1ØØ1

Dans un nombre décimal, chaque chiffre est remplacé par son équivalent binaire. Par exemple:

36 = ØØ11 Ø11Ø
(3) (6)

Cela nous amène à une constatation: ce code perd une grande quantité de place mémoire. En effet un octet ne permet de coder que des nombres décimaux compris entre Ø et 99 (contre Ø et 255 pour le binaire). Ce code n'en reste pas moins très utilisé.

## 4.3. LA SYNTAXE ASSEMBLEUR 68Ø3

Dans ce paragraphe, avant d'envisager la description proprement dite du 68Ø3, nous allons décrire la syntaxe qui sera utilisée dans tout le restant de cet ouvrage. Cette syntaxe est bien sûr celle utilisée par l'Éditeur-Assembleur de l'Alice.

Le chargement de celui-ci est instantané puisqu'il est contenu en ROM. Son appel nécessite juste une initialisation préalable.

En effet, il est nécessaire de renseigner l'Éditeur où (à quelle adresse) il va stocker les caractères composant le programme source (ensemble des mnémoniques et opérandes associés représentant le programme destiné à être assemblé).

Ceci peut se faire grâce à l'instruction Basic CLEAR qui a été vue dans le chapitre consacré au Basic.

La syntaxe de cette instruction est: CLEAR N1,N2

N1 étant le nombre d'octets à réserver en mémoire et N2 étant l'adresse de départ du programme source.

Notons que N2 ne correspond en aucune façon à l'adresse de début du programme objet (c'est-à-dire du programme assemblé).

Dans le cas de l'Alice 32, N2 peut prendre des valeurs comprises entre 13287 et 2Ø48Ø ($33E7 à $5ØØØ en hexadécimal).

Dans le cas de l'Alice 32 avec extension RAM de 16 Koctets, N2 sera compris entre 13287 et 36864 ($33E7 à $9ØØØ en hexadécimal).

Dans le cas de l'Alice 9Ø, N2 sera compris entre 13287 et 45Ø56 = $BØØØ.

Après avoir effectué cette initialisation, tapez simultanément sur les touches SHIFT et & puis ENTER. L'écran doit devenir alors bleu, indiquant que l'Éditeur attend que vous rentriez des lignes assembleur.

Ceci correspond à un "démarrage à froid" de l'Éditeur-Assembleur: le programme source est chargé à partir de l'adresse N2 et tout éventuel autre programme assembleur rentré préalablement est perdu.

Pour sortir de l'Éditeur-Assembleur il vous suffit de taper deux fois sur la touche BREAK ce qui provoque l'affichage du "OK" caractéristique de l'interpréteur Basic.

Pour revenir à l'Éditeur-Assembleur et ne pas perdre un éventuel programme source présent en mémoire, il vous suffit de taper simultanément sur les touches SHIFT et % puis ENTER. C'est ce que l'on appelle un démarrage à chaud.

Mais voyons maintenant comment écrire un programme dit "source". Une ligne en langage d'assemblage est subdivisée en un

certain nombre de parties appelées "champ". En effet, un programme écrit en assembleur véhicule un certain nombre d'informations distinctes qui, pour être bien comprises par l'Assembleur, doivent être bien séparées les unes des autres.

Cette séparation peut s'effectuer par une pression simultanée sur les touches CTRL et T (ce qui correspond à une tabulation). A chaque fois que cette séquence de touches est rentrée au clavier le curseur se déplace de six cases vers la droite. Nous allons passer en revue dans les lignes qui suivent ces différents champs.

### 1) Le champ étiquette (label)

En Basic, une étiquette représente une adresse de branchement. (*Exemple*: GOTO 1Ø0).

La zone étiquette est le premier champ dans une ligne écrite en assembleur. Elle peut être vide (pas d'étiquette) ou remplie. Les labels sont utilisés lors des instructions de saut inconditionnel, conditionnel et d'appel de sous-programme.

Du fait de la taille limitée du champ qui lui est affecté, une étiquette doit avoir un nombre maximum de caractères égal à 5.

*Exemple*:

```
BOUCL
BRAN
DECIM
```

ou si vous aimez l'Anglais:

```
LOOP
START
```

Nous vous donnons ci-dessous à titre indicatif le listing d'une ligne en Assembleur:

```
SOMME ADDA   #$30   ;EFFECTUE LA SOMME
champ
label
```

**2) Le champ opération**

Le champ opération est situé juste après le champ étiquette duquel il est espacé par un espace ou une tabulation. Dans ce champ, on laisse le mnémonique de l'instruction.

```
SOMME ADDA   #$30   ;EFFECTUE LA SOMME
      ----
      champ
      opération
```

**3) Le champ opérande**

Ce champ est celui qui risque de poser le plus de problèmes au programmeur novice ou assembleur.

L'opérande sert à définir la donnée sur laquelle s'effectue l'instruction. Elle doit donner à l'assembleur toutes les précisions nécessaires à sa compréhension du programme et en particulier concernant le mode d'adressage.

Dans ce champ opérande, on peut trouver :

— des nombres,

— des noms de variables,

— des étiquettes.

*a) Les nombres*

L'Assembleur de l'Alice n'accepte que des nombres hexadécimaux. Le système de numération hexadécimale a été vu précédemment dans ce chapitre. Par analogie avec la plupart des Assembleurs existant actuellement, les nombres hexadécimaux présentent comme caractère distinctif la présence d'un **$** (dollar).

C'est ainsi que 32F8 sera écrit $32F8 dans tout programme source.

*b) Les noms de variables*

Dans le champ opérande peuvent apparaître des noms de variables pour définir une adresse par exemple. L'Assembleur les traite comme des nombres (à condition qu'une valeur ait été assignée à chacune d'elles auparavant).

*Exemple* :

```
ADCA   VALEUR
```

Supposons que VALEUR = $1Ø, alors il y aura addition entre l'Accumulateur et la case-mémoire d'adresse $ØØ1Ø.

*c) Les noms d'étiquette*

Dans les instructions de branchement, par exemple, on peut voir apparaître une étiquette dans le champ opérande. Cette étiquette possède les caractéristiques décrites précédemment.

*Exemple* :

```
JMP    BOUCL
```

**4) Le champ commentaire**

Dans un programme en assembleur, le commentaire n'a aucune influence (à condition, bien sûr, qu'il soit placé au bon endroit et séparé du champ opérande par un espace ou une tabulation ainsi que d'un délimiteur.

Les commentaires, bien que trop souvent négligés, sont très utiles car ils permettent de documenter un programme et de ce fait de le rendre plus intelligible.

Le délimiteur, destiné à signaler à l'assembleur qu'il est en présence d'un commentaire, est un point-virgule (";").

*Exemple* :

```
;BOUCLE DE DELAI
```

Contrairement à ce que l'on pourrait croire, écrire des commentaires utiles est assez difficile. En effet, il ne faut pas écrire n'importe quoi. Voici quelques règles et conseils à observer :

— Les commentaires doivent servir à éclairer le fonctionnement global du programme.

— Ils peuvent servir à expliquer non seulement l'utilité d'une instruction particulière mais aussi d'un morceau de programme (sous-programme par exemple).

— Un commentaire doit être clair et concis.

— Il est inutile de s'attarder sur des points évidents mais il ne faut pas hésiter à insister sur des endroits clefs.

— Écrivez par exemple: "TEMPS MAXIMAL ECOULE?" ou bien "TESTE SI L'INTERRUPTEUR EST FERME" plutôt que "TESTE LA VALEUR DU BIT N", "BRANCHEMENT AU DEBUT".

Il faut savoir que le temps passé à commenter un programme est pratiquement toujours récupéré et même fait gagner du temps lors de la phase "mise au point" par exemple.

Tous ces détails vous semblent peut-être un peu abstraits. Ne vous inquiétez pas, nous vous donnerons un peu plus loin un exemple de listing en assembleur afin de vous familiariser avec la syntaxe. Mais avant cela examinons un point très important dans l'écriture d'un programme en assembleur.

**5) Les pseudo-instructions**

Nous appelons pseudo-instructions les instructions qui ne font pas partie de la bibliothèque du 68Ø3 et qui sont juste destinées à donner des informations (ou directives) à l'Assembleur.

Nous allons repertorier toutes celles dont dispose l'Éditeur-Assembleur de l'Alice. Notons que les pseudo-instructions sont situées dans le champ opération d'un programme source.

*a) La directive origine (ORG)*

Elle permet de définir l'adresse de départ d'un programme. Supposons que nous voulions faire commencer votre programme à l'adresse $58ØØ, nous écrirons alors:

```
ORG     $58ØØ
```

Après l'assemblage, l'adresse de la première instruction sera donc $58ØØ.

*b) Le point d'entrée du programme (EXC)*

De même qu'il est nécessaire de fournir à l'Assembleur une indication concernant l'adresse d'implantation en mémoire du programme, il faut qu'il connaisse son adresse de lancement, c'est-à-dire l'adresse de la première instruction qui devra être exécutée.

Cette adresse peut dans certains cas être égale à l'adresse d'implantation déterminée par la pseudo-instruction ORG.

Le point d'entrée est déterminé par la pseudo-instruction EXC.

On écrira par exemple :

```
        ORG    $5800
        EXC    INIT
INIT    ...
```

Dans ce cas la première instruction à exécuter sera située à l'adresse $5800.

*c) Affecter une valeur à une étiquette*

Lorsqu'il est nécessaire d'attribuer une valeur hexadécimale à une étiquette ou à une variable que l'on désire par la suite référencer par un nom, on utilisera la directive "=".

On écrira par exemple : VAL = $3F58 qui attribue à VALEUR le nombre hexadécimal $3F58.

```
        ORG    $5800
VAL     =      $3F58
        EXC    INIT
INIT    RTS
```

*d) Définir une chaîne de caractères*

A chaque caractère peut être associé un code sur un octet que l'on nomme le code ASCII (de l'Anglais American Standard Code for Information Interchange).

La directive "'" (apostrophe) permet de stocker en mémoire les codes ASCII des caractères présents dans la chaîne précisée après l'apostrophe.

On écrira par exemple :

```
TEXT  'CECI EST UN ESSAI
```

*e) Les directives de réservation d'espace-mémoire*

Il peut arriver que l'on veuille réserver des octets, des doubles octets ou des portions entières de mémoire pour y stocker des données ou des tableaux. Nous allons définir ici un certain nombre de directives qui remplissent ces rôles.

— la directive DFO : elle permet de réserver un octet en mémoire.

*Exemple* :

```
VAL    DFO    $00
```

Lorsque l'assembleur rencontre cette ligne, il place la valeur $OO dans la première case-mémoire et lui assigne le nom VAL.

— la directive DFD : elle permet de réserver un double octet en mémoire.

*Exemple* :

```
VAL    DFD    $12F4
```

Lorsque l'assembleur rencontre cette ligne, il place la valeur $12 dans la première case-mémoire et lui assigne le nom VAL. Il place ensuite la valeur $F4 dans la case-mémoire suivante.

— la directive BLC : elle permet de réserver un bloc mémoire. Ce bloc a une taille inférieure ou égale à 255.

*Exemple* :

```
BLC    $45
```

Quand l'Assembleur rencontre cette ligne, il saute $45 = 69 cases-mémoire et reprend son assemblage. Ces 69 cases-mémoire pourront sans dommage être ensuite utilisées par le programme en langage machine pour y stocker des données.

L'Éditeur-Assembleur présent dans l'Alice n'est pas un des plus performants existant actuellement. En revanche, il est très simple d'utilisation et permet de résoudre tous les problèmes courants de programmation.

Nous vous donnons ci-dessous un exemple de programme non assemblé (programme source) puis assemblé (programme source + code objet généré).

```
        ORG    $4000 ;DEBUT PROGRAMME
        EXC    SON
SON     LDAB   #$FF
        LDAA   #$00  ;MISE A ZERO DE VAL
        STAA   VAL
INV     LDAA   VAL   ;INVERSION DE VAL
        EORA   #$FF
        STAA   VAL
        STAA   $BFFF ;ENVOIE CETTE VALEUR
                     ;DANS GENERATEUR
        TBA          ;B MIS DANS A
LOOP    DECA         ; BOUCLE D'ATTENTE
        BNE    LOOP  ;TERMINE?
                     ;NON, CONTINUE
        DECB         ;B DIMINUE D'UNE UNITE
        BNE    INV   ;SI <>0 CONTINUE
        JMP    SON   ;SINON, RECOMMENCE
        ;VAL CONTIENT LA VALEUR A ENVOYER
        ;DANS LE GENERATEUR DE SON
VAL     DFO    $00
```

Après assemblage, cela donne :

```
ASSEMBLEUR ALICE REV 1.00*
COPYRIGHT MATRA, 1984

1                       ORG    $4000 ;DEBUT PROGRAMME
2                       EXC    SON
3  4000 C6FF    SON     LDAB   #$FF
4  4002 8600            LDAA   #$00  ;MISE A ZERO DE VAL
5  4004 B7401C          STAA   VAL
6  4007 B6401C  INV     LDAA   VAL   ;INVERSION DE VAL
7  400A 88FF            EORA   #$FF
```

```
 8 400C B7401C        STAA  VAL
 9 400F B7BFFF        STAA  $BFFF ;ENVOIE CETTE VALEUR
10                               ;DANS GENERATEUR
11 4012 17           TBA         ;B MIS DANS A
12 4013 4A      LOOP DECA        ; BOUCLE D'ATTENTE
13 4014 26FD         BNE   LOOP  ;TERMINE?
14                               ;NON,CONTINUE
15 4016 5A           DECB        ;B DIMINUE D'UNE UNITE
16 4017 26EE         BNE   INV   ;SI <>0 CONTINUE
17 4019 7E4000       JMP   SON   ;SINON,RECOMMENCE
18                   ;VAL CONTIENT LA VALEUR A ENVOYER
19                   ;DANS LE GENERATEUR DE SON
20 401C 00      VAL  DFO   $00
21
22

0 ERREUR(S) PASSE 1
0 ERREUR(S) PASSE 2
SYMBOLES :
--------
SON  =4000 INV  =4007 LOOP =4013 VAL  =401C
```

Si vous faites exécuter ce programme, vous obtiendrez un son "galactique" dans le haut-parleur de votre téléviseur. Nous allons décrire brièvement les opérations effectuées lors de l'assemblage. Le rôle d'un assembleur est de générer à partir d'un programme source (dont la structure a été décrite ci-dessus) un code-objet exécutable par la machine (le 68Ø3).

En fait, lors de l'assemblage, il se produit deux choses :

*a)* l'assembleur produit un listing du programme assemblé qui comporte les caractéristiques suivantes :

— à droite se trouve le programme source inchangé avec, en plus, une numérotation des lignes ;

— à gauche se trouvent deux colonnes qui représentent d'une part le code-machine correspondant au programme (à gauche directement du code-source) et d'autre part les adresses d'implantation en mémoire des codes-opération ;

— il détecte les erreurs éventuelles (erreurs de syntaxe, sur des variables, des étiquettes, etc...). Ces erreurs et les codes associés sont

décrits dans le "guide d'instructions de l'Éditeur-Assembleur", c'est pourquoi nous n'y reviendrons pas ici.

*b)* l'assembleur génère un code-objet destiné à être stocké sur cassette et qui est, lui, directement exécutable par le 68Ø3.

L'Assembleur de l'Alice fonctionne en deux passes (une passe est une lecture du programme source). Dans la première, il examine chaque instruction avec son opérande et il détermine la longueur (1, 2, ou 3 octets) du code-objet résultant. De plus il charge tous les symboles (étiquettes, variables) dans une table.

Chaque fois qu'il rencontre une étiquette, il lui assigne l'adresse qu'occupera le code-opération du mnémonique présent sur la même ligne. Afin que l'assembleur puisse connaître cette adresse, il existe un pointeur d'adresse que l'assembleur fixe à une valeur origine au début du programme. A chaque instruction, ce pointeur est incrémenté selon la longueur de celle-ci. Pendant cette première passe, l'assembleur détecte les erreurs de syntaxe.

Durant la seconde passe, il fait l'assemblage proprement dit afin de générer le code-objet. Chaque fois qu'il rencontre un symbole (nom de variable, étiquette), il le recherche dans la table générée durant la passe 1 et lui affecte l'adresse ou la donnée correspondante.

Il détecte simultanément des erreurs plus délicates portant par exemple sur des étiquettes ou des variables.

## 4.4. LES REGISTRES INTERNES DU 68Ø3

### a) Les Accumulateurs

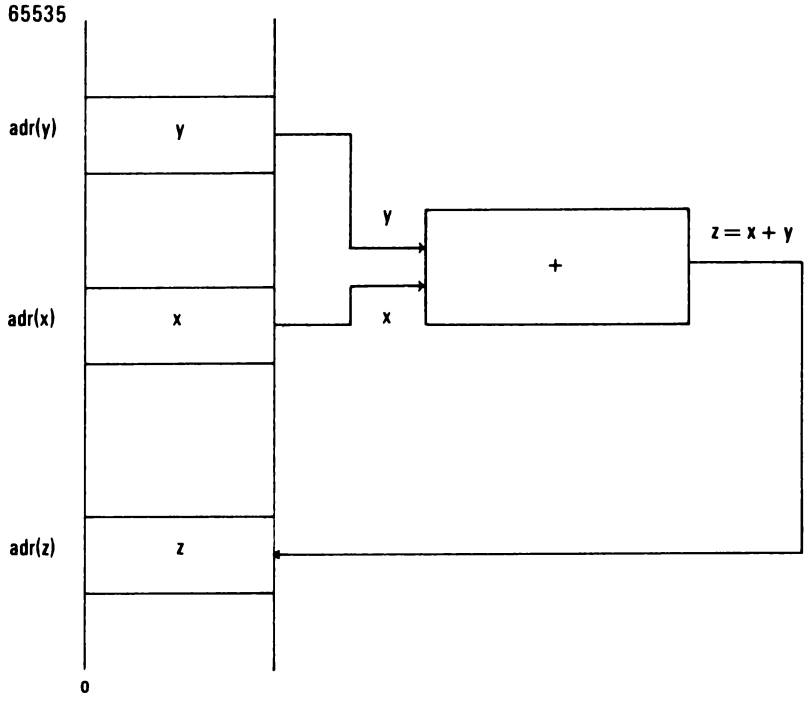
Avant de vous présenter les différents registres du 68Ø3, nous allons essayer de vous faire sentir leur nécessité à travers un exemple.

Soit à effectuer l'addition de deux nombres x et y : $z = x + y$.

Nous allons supposer que x et y sont situés dans deux cases-mémoire distinctes donc à deux adresses différentes que nous appellerons adr(x) et adr(y).

Le résultat de l'addition, soit z, sera stocké à l'adresse adr(z).

Le chemin suivi par les données est le suivant:




Nous avons déjà dit que le 68Ø3 pouvait adresser 64 Koctets de mémoire (ou plus exactement 65536 = $2^{16}$ octets). Afin de pouvoir sélectionner une case-mémoire et une seule, on définit une adresse sur 16 bits (ce qui fait donc 2 mots de 8 bits). Nous avons vu d'autre part qu'une instruction (ici addition) pouvait être codée par un mot de 8 bits appelé code-opération.

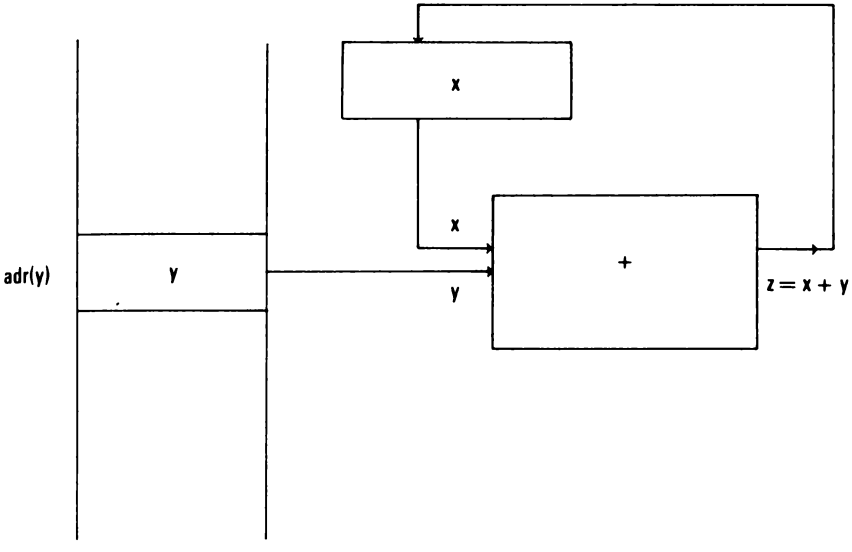
Pour effectuer l'addition z = x + y nous devons donc connaître:

| | |
|---|---|
| adr(x) | : 2 octets |
| adr(y) | : 2 octets |
| code-opération | : 1 octet |
| adr(z) | : 2 octets |
| Total | : 7 octets |

7 octets sont donc nécessaires à l'exécution de cette opération. Il est

donc aisé de concevoir que si les choses se passaient comme décrit ci-dessus, on arriverait rapidement à des programmes de taille gigantesque.

Fort heureusement les concepteurs du 68Ø3 se sont penchés sur le problème. Examinons maintenant la figure suivante:




Dans ce cas le contenu de l'adresse de y, (adr(y)) est ajouté à x contenu dans une case-mémoire particulière et située en dehors de l'espace mémoire adressable du microprocesseur. Le résultat z = x + y est ensuite mis dans cette même case-mémoire.

Afin de définir complètement l'opération, nous devons donc connaître:

| | |
|---|---|
| adr(y) | : 2 octets |
| code-opération: | 1 octet |
| Total | : 3 octets |

Comme vous pouvez le constater, nous avons réduit considérablement le nombre de mots mémoire nécessaires pour définir l'addition de 2 nombres x et y. En revanche, cela a nécessité la présence d'une case-mémoire particulière (contenant x puis z = x + y) que nous nommerons registre.

Plus précisément le rôle joué par celui-ci dans le cas qui nous intéresse est généralement confié à l'Accumulateur qui est un registre fondamental du 68Ø3. En fait, il existe dans ce dernier 2 accumulateurs de 8 bits, A et B qui jouent le même rôle.

Pratiquement, on utilise ces accumulateurs pour les instructions suivantes :

— les instructions arithmétiques et logiques,

— les instructions de comparaison.

Ils sont de plus utilisés lors de transferts Mémoire → Accumulateur, Accumulateur → Mémoire, Registre → Accumulateur et Accumulateur → Registre.

Les Accumulateurs A et B font partie de l'ensemble des registres internes du 68Ø3.

Notons qu'ils peuvent être réunis pour former un unique registre de 16 bits appelé D. (A constitue alors l'octet de poids fort tandis que B constitue l'octet de poids faible).

Le 68Ø3 dispose d'ailleurs d'instructions spécialisées dans le traitement d'informations 16 bits.

**b) Le registre d'index X**

A côté des accumulateurs A et B, il existe un registre spécialisé de 16 bits appelé registre d'index X. Bien qu'il soit destiné plus particulièrement à une utilisation dans le cadre du mode d'adressage indexé (nous reviendrons sur ce point dans le prochain paragraphe) il peut être considéré comme un registre d'usage général, c'est-à-dire :

— comme registre de stockage de résultats intermédiaires,

— comme compteur de boucle.

En effet, il est mis en jeu dans les opérations de chargement, de comparaison, etc...

**c) Le registre de condition CCR**

Ce registre est un pu particulier. Il comporte 8 bits dont seulement 6 sont utilisés.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | Ø |
|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | H | I | N | Z | V | C |

Les bits 6 et 7 sont toujours positionnés à 1 et n'ont aucune signification. Les bits Ø à 5 sont les suivants :

bit 5 = H = indicateur de demi-retenue (en anglais HALF-CARRY)

bit 4 = I = masque d'interruption (IRQ)

bit 3 = N = indicateur de résultat négatif

bit 2 = Z = indicateur de zéro

bit 1 = V = indicateur de débordement (OVERFLOW)

bit Ø = C = indicateur de retenue (CARRY)

Ces indicateurs peuvent jouer deux rôles :

— ils peuvent influencer le déroulement futur d'un programme si on les prépositionne à une certaine valeur (Ø ou 1),

— leur état peut dépendre des résultats d'un calcul antérieur et peut donc constituer une information précieuse par la suite.

Le 68Ø3 possède des instructions particulières permettant de modifier ou de lire la valeur prise par certains indicateurs du registre de condition CCR. D'autre part, certaines instructions du 68Ø3 ont un déroulement qui dépend de la valeur prise par ces indicateurs (instructions de branchement conditionnel). Nous reviendrons sur ces indicateurs lors de la description des instructions du 68Ø3.

### d) Le compteur ordinal : registre PC

Un programme est exécuté par le microprocesseur de manière séquentielle, c'est-à-dire que celui-ci exécute les instructions contenues dans la mémoire les unes après les autres (et une seule à la fois).

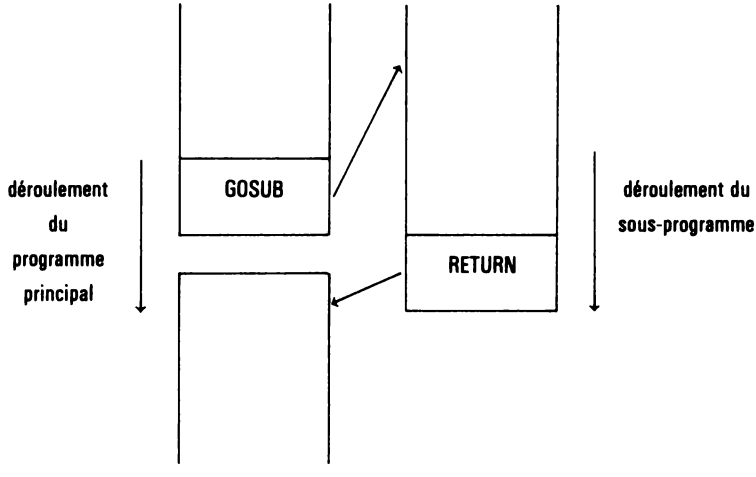
Afin que le programme se déroule correctement, il apparaît évident que le microprocesseur, lorsqu'il est en train d'exécuter une instruction, doit connaître l'adresse de la suivante.

Dans ce but, il existe dans le microprocesseur un registre spécial appellé compteur ordinal (registre PC : de l'Anglais Program Counter).

Ce registre comporte 16 bits puisque l'adresse d'un mot mémoire est définie de manière unique à l'aide de 16 bits. Chaque fois que le microprocesseur va chercher un octet en mémoire, le compteur ordinal est incrémenté de 1.

**e) Le pointeur de pile: registre SP**

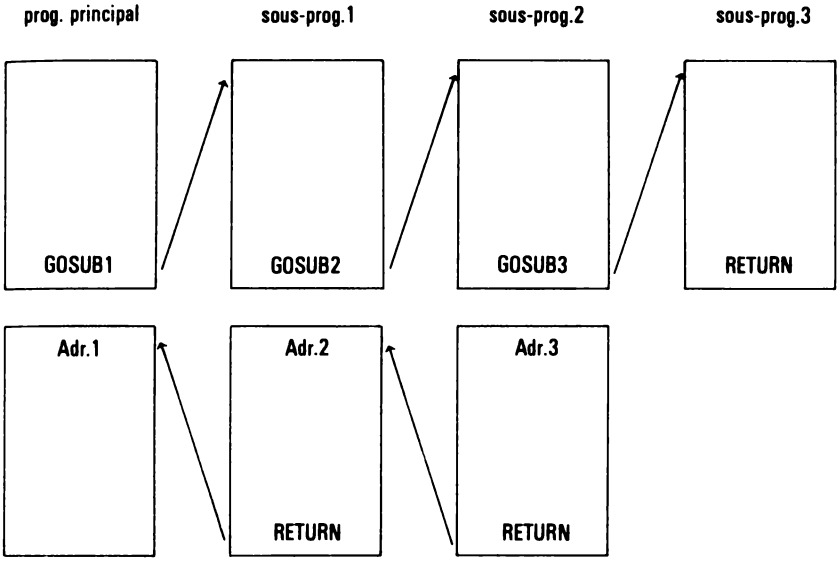
Revenons sur le rôle du compteur ordinal (PC): nous avons vu qu'il était nécessaire que le microprocesseur sache à tout instant l'adresse de la prochaine instruction à exécuter. Examinons ce qu'il se passe lorsque le programme principal fait appel à un sous-programme.




Le déroulement du programme est le suivant: le programme principal s'exécute normalement et atteint une instruction spécifique d'appel de sous-programme (l'équivalent d'un GOSUB en Basic). Le compteur ordinal se charge alors avec l'adresse de début de ce sous-programme qui s'exécute à son tour jusqu'à ce que le microprocesseur rencontre une instruction de retour de sous-programme (l'équivalent de RETURN en Basic). Le problème est que le microprocesseur, tel que nous l'avons décrit jusqu'à présent, ne sait absolument pas à quelle adresse reprendre le déroulement du programme principal.

Il apparaît donc nécessaire, lors de l'appel d'un sous-programme, de mémoriser l'adresse de l'instruction suivant immédiatement le GOSUB.

Allons maintenant un peu plus loin: imaginons un programme principal et un ensemble de sous-programmes imbriqués les uns dans les autres:




Lors de l'appel du sous-programme 1, il est nécessaire de mémoriser la valeur de Adr.1 qui est l'adresse de l'instruction suivant immédiatement l'appel du sous-programme 1 dans le programme principal. Puis, lors de l'appel du sous-programme 2, nous devons mémoriser Adr.2 et de même pour le sous-programme 3.

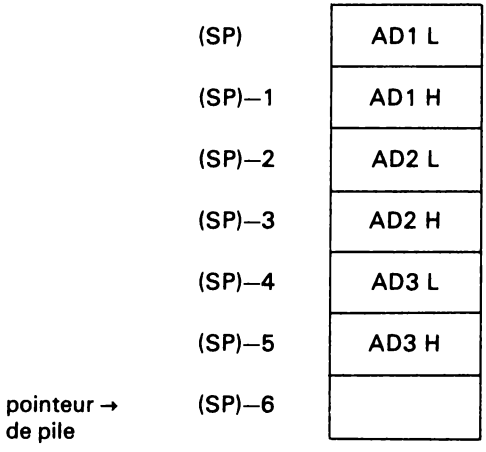
Lorsque, dans le déroulement de ce dernier, le processeur rencontre l'instruction de retour de sous-programme, il faut que le compteur ordinal vienne se charger avec la dernière adresse mémorisée soit Adr.3. Ensuite, lorsque le sous-programme 2 aura fini de se dérouler, le PC devra se charger avec l'avant-dernière adresse mémorisée soit Adr.2. Il en sera de même pour Adr.1. Donc la dernière adresse mémorisée est la première sortie et la première adresse mémorisée la dernière sortie.

Imaginons une pile d'assiettes: on les empile une à une et on les lave ensuite. La dernière posée sur le dessus sera la première lavée. Voilà qui nous amène directement à parler de la notion de pile dans un microprocesseur.

Il existe une zone mémoire située dans l'espace adressable du microprocesseur et qui fonctionne suivant le principe de la pile d'assiettes décrite ci-dessus.

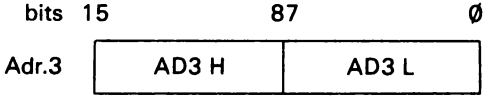
Cette zone mémoire est fixée par le contenu du registre SP qui possède 16 bits. Son contenu doit être programmé chaque fois que le 68Ø3 est mis sous tension, ceci par l'intermédiaire d'instructions qui apparaîtront au début du premier programme exécuté.

Cette pile peut donc être située n'importe où dans l'espace adressable du 68Ø3. Considérons l'exemple choisi précédemment (appel de sous-programmes). La pile contiendra après l'appel du sous-programme 3 les octets suivants :




On notera :

AD3L = partie basse de l'adresse Adr.3 (bits de poids faible)
AD3H = partie haute de l'adresse Adr.3 (bits de poids fort)




(SP) désignant le contenu du registre (SP) avant l'appel du sous-programme 1.

On remarque que les adresses sont chargées dans la pile à partir du

haut. Lors du stockage d'une adresse de retour de sous-programme les opérations suivantes sont effectuées :

— l'octet de poids faible de l'adresse de retour est chargé dans la pile ;

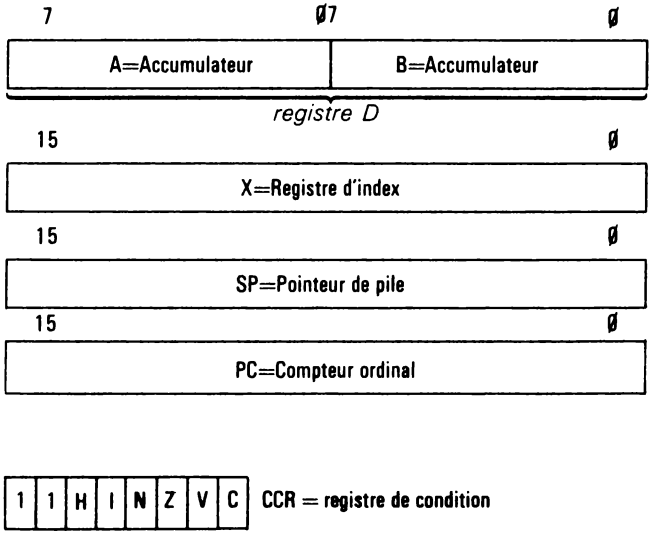
— le pointeur de pile est décrémenté ;

— l'octet de poids fort de l'adresse de retour est chargé dans la pile ;

— le pointeur de pile est décrémenté à nouveau.

Lors d'une instruction de retour de sous-programme, l'opération inverse se produit : le compteur ordinal est chargé par l'octet de poids fort puis par l'octet de poids faible de l'adresse de retour. Le pointeur de pile se déplace vers le haut (est incrémenté de 2) et l'incrémentation se fait avant le chargement (contrairement au cas de l'appel de sous-programme).

Pour résumer ce paragraphe, les différents registres internes du 68Ø3 sont présentés ci-dessous :



## 4.5. LES DIFFÉRENTS MODES D'ADRESSAGE DU 68Ø3

Tout d'abord qu'appelle-t-on mode d'adressage ? Par définition, un mode d'adressage est un moyen d'accéder à une case-mémoire donnée.

### 4.5.1. L'adressage implicite

L'adressage implicite n'est pas en réalité un véritable mode d'adressage. En effet aucune adresse n'est nécessaire pour définir les instructions correspondant à ce mode. Le code-opération de ces instructions tient, comme c'est toujours le cas du 68Ø3, sur un octet. On peut mentionner parmi elles les instructions portant directement sur le contenu de registres internes (instructions de décalage par exemple), les instructions de retour de sous-programme ou d'interruption, les interruptions logicielles, les instructions de transfert de registre interne, les opérations sur la pile.

*Exemples* :

```
ASLA
RTI
SWI
TAB
PSHA
```

### 4.5.2. L'adressage immédiat

Dans ce mode d'adressage les instructions comportent deux parties :

— le code-opération proprement dit qui est, comme d'habitude, codé sur un octet,

— l'opérande qui comprend, selon le cas, un ou deux octets selon que l'instruction considérée porte sur des mots de 8 bits ou de 16 bits. Cette opérande correspond tout simplement à la donnée sur laquelle porte l'instruction considérée.

*Exemple 1* : Soit à additionner la valeur $Ø5 au contenu de l'accumulateur A.

L'instruction sera la suivante :

```
ADDA #$05
```

Le symbole "#" précise à l'assembleur que l'on est dans le cas d'un adressage immédiat.

*Exemple 2* : Soit à comparer le contenu du registre X avec la valeur 16 bits $12F3.
L'instruction sera la suivante :

```
CPX #$12F3
```

### 4.5.3. L'adressage étendu

Ce mode d'adressage permet d'accéder à la totalité de l'espace adressable du 68Ø3. L'opérande est ici une adresse de 16 bits (donc 4 chiffres hexadécimaux). La donnée située à cette adresse (et parfois à la suivante dans le cas d'instructions portant sur des mots de 16 bits) est utilisée par l'instruction considérée.

*Exemple 1* : Soit à additionner le contenu de la case-mémoire d'adresse $53F8 à l'accumulateur B.
L'instruction s'écrit :

```
ADDB $53F8
```

### 4.5.4. L'adressage direct

Ce mode d'adressage est similaire à l'adressage étendu sauf qu'ici l'opérande tient sur un seul octet. Elle permet donc d'adresser un espace-mémoire de 256 octets. Cet espace est situé en page-zéro qui est la première page de 256 octets adressable par le 68Ø3 (adresses comprises entre Ø et 255).

*Exemple* : Soit à charger l'accumulateur D avec le contenu des cases-mémoire d'adresses $12 et $13.
On écrira :

```
LDD < $12
```

Le symbole "<" est destiné à renseigner l'assembleur qu'il est en présence d'un mode d'adressage direct.

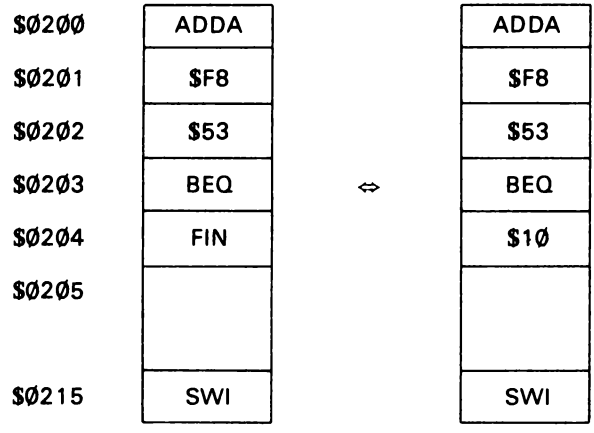
### 4.5.5. L'adressage relatif

L'adressage relatif est utilisé uniquement pour les instructions de branchement conditionnel (le branchement n'a lieu que si la condition désirée est réalisée). Ces instructions sont l'équivalent du IF... THEN en Basic. Afin de bien saisir le fonctionnement de ce mode d'adressage, le mieux est de donner un exemple.

Soit le petit programme suivant:

```
      ADDA $53F8
      BEQ  FIN
      .
      .
      .
FIN   SWI
```

On effectue l'addition du contenu de la case-mémoire d'adresse $53F8 avec l'accumulateur. L'instruction BEQ veut dire "branchement si égalité". Ici il y a donc branchement vers FIN si le contenu de l'accumulateur après addition est égal à zéro. Le bit Z du registre de condition est alors égal à 1. Sinon, le programme continue et exécute les instructions présentes après le BEQ. Supposons que le début du programme (instruction ADDA) soit placé à l'adresse $Ø2ØØ. Nous donnons ci-dessous le contenu de la mémoire à partir de cette adresse:



Nous n'avons pas introduit les code-opérations des instructions ADDA, BEQ et SWI afin que le programme soit plus explicite.

L'étiquette FIN est située à l'adresse $Ø215.

Lors d'un branchement relatif on charge l'octet suivant l'instruction de branchement par la différence (en binaire) qui existe entre la valeur de l'adresse d'arrivée (ici $Ø215) et la valeur du registre PC pointant sur l'instruction qui suit immédiatement le BEQ, donc ici $Ø2Ø5. Nous chargerons donc la case-mémoire d'adresse $Ø2Ø4 par la valeur hexadécimale :

```
$0215- $0205 =$10
```

Le contenu de la case-mémoire d'adresse $Ø2Ø4 aurait très bien pu être négatif (avec la notation en complément à 2) si le branchement avait dû être effectué vers l'arrière.

Par exemple, pour un branchement en $Ø2ØØ, le contenu de la case-mémoire d'adresse $Ø2Ø4 aurait été :

```
$0200- $0205 = $-05
             = $FB
```

La syntaxe assembleur relative au mode d'adressage relatif est la suivante :

BEQ FIN (si on désire un branchement à l'étiquette FIN)

La valeur du déplacement ne dépend absolument pas de l'adresse d'implantation en mémoire du programme. L'adressage relatif permet donc d'avoir des programmes translatables (qui fonctionnent à n'importe quelle adresse mémoire), ceci à condition que les branchements soient tous relatifs à l'intérieur du programme. En particulier il ne doit pas y avoir de sauts inconditionnels ni d'appels de sous-programmes internes.

### 4.5.6. Le mode d'adressage indexé

Dans ce mode d'adressage, l'adresse de la donnée désirée est calculée de la manière suivante :

Le contenu du registre d'index X est additionné en tenant compte de la retenue à un déplacement considéré comme un nombre non signé et correspondant au deuxième octet de l'instruction.

La syntaxe de ce mode d'adressage est la suivante.
ADCB $12,X dans le cas d'une instruction d'addition avec un déplacement non signé égal à $12, ",X" indiquant que l'on est en présence d'un adressage indexé.

# 5

# Le jeu d'instructions du 68Ø3

## 5.1. INTRODUCTION

Dans ce chapitre, nous avons choisi de vous présenter le jeu d'instructions du 68Ø3 afin que vous puissiez tirer au mieux parti de l'Éditeur-Assembleur qui est incorporé dans la mémoire morte de cette machine.

Bien que le 68Ø3 ne soit pas un des microprocesseurs les plus complexes du marché, il ne possède pas moins de 126 instructions différentes, certaines d'entre elles travaillant avec différents modes d'adressage.

Dans ce chapitre, chacune d'elle sera décrite de manière détaillée et, dans la mesure du possible illustrée d'un exemple écrit à l'aide de l'Assembleur de l'Alice, ceci afin de ne pas décourager les lecteurs novices en programmation Assembleur.

Nous avons regroupé ces différentes instructions en 9 groupes qui sont les suivants:

— les instructions de chargement,
— les instructions arithmétiques,
— les instructions logiques,

— les instructions sur le registre d'état,
— les instructions de comparaison,
— les instructions de branchement,
— les instructions d'appel et de retour de sous-programme,
— les instructions sur la pile,
— les instructions spéciales.

## 5.2. LES INSTRUCTIONS DE CHARGEMENT

Nous regrouperons sous cette appellation toutes les instructions qui permettent de charger une case-mémoire ou un registre qu'elle qu'en soit la source (mémoire, registre, etc...).

Il est logique de commencer par ce groupe d'instructions car les opérations qu'effectue le microprocesseur portent bien évidemment sur le contenu d'une case-mémoire ou d'un registre.

Nous allons tout de même les séparer en trois groupes qui sont les suivants :

— les instructions de chargement de registre,
— les instructions de chargement mémoire,
— les instructions d'échange de registres.

De plus certaines instructions opèrent sur des octets tandis que d'autres travaillent sur des mots de 16 bits. Nous serons donc amenés à effectuer une distinction entre ces deux types d'instructions.

### 5.2.1. Les instructions de chargement de registre

Il existe sous cette rubrique des instructions portant sur des octets et des mots de 16 bits.

#### 5.2.1.1. Instructions sur 8 bits

Il s'agit des instructions CLR, CLRA, CLRB, LDAA LDAB.

En bon anglais "CL" veut dire "CLEAR" (mettre à zéro). Par conséquent les instructions CLRA, CLRB, CLR permettent de remettre

à zéro respectivement le contenu des registres A, B ou de la case-mémoire d'adresse spécifiée.

En bon anglais "LD" veut dire "LOAD" (charger). Donc les instructions LDAA et LDAB permettent de charger respectivement les registres A et B.

Par la suite, nous donnerons pour chaque instruction un tableau regroupant les différents modes d'adressage, le code-opération correspondant et les indicateurs du registre de condition qui sont affectés par cette instruction.

Mais auparavant, effectuons un retour sur les bits N et Z du registre de condition CCR.

Nous avons vu précédemment la signification de ces 2 bits:

N = indicateur de résultat négatif,

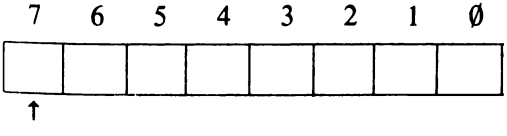
Z = indicateur de zéro.

Dans le cas des instructions de type "LD", ces deux bits seront positionnés à "1" si la condition qu'ils représentent est réalisée.

Autrement dit, si l'accumulateur est chargé avec la valeur $00, le bit Z sera mis à 1. Dans le cas contraire, il sera mis à zéro.

Le fonctionnement du bit "N" est le suivant:

Nous savons qu'un octet peut représenter soit un nombre positif compris entre 0 et 255 soit un nombre signe compris entre −128 et +127, un nombre négatif étant représenté par son complément à 2. Le bit de poids fort (bit 7) représente alors le signe du nombre binaire.




bit 7 = bit N

Le bit N est donc tout simplement la recopie du bit 7 de l'octet.

*Exemple*:

$17=00010111

d'où N=Ø

$F8=11111ØØØ

d'où N=1

*a) Les instructions CLR, CLRA et CLRB*

Grâce à ces instructions il est possible, nous l'avons vu, de remettre à zéro le contenu des registres A ou B ou le contenu de la case-mémoire d'adresse spécifiée.

On aura donc:

Ø → A (instruction CLRA)

Ø →B (instruction CLRB)

Ø →M (instruction CLR)

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| CLR $LL,X | indexé | 6F | N=V=C=Ø,Z=1 |
| CLR $HHLL | étendu | 7F | N=V=C=Ø,Z=1 |
| CLRA | implicite | 4F | N=V=C=Ø,Z=1 |
| CLRB | implicite | 5F | N=V=C=Ø,Z=1 |

Explicitons un peu les notations employées:

Les mnémoniques qui sont donnés (et ce sera vrai dans tout cet ouvrage) sont les mnémoniques standard MOTOROLA et correspondent à ceux employés dans l'Éditeur-Assembleur de l'Alice.

Il en est de même pour la syntaxe des différents modes d'adressage qui a été décrite précédemment.

LL désigne une valeur hexadécimale (donc précédée d'un "dollar") représentant un octet (2 chiffres hexadécimaux). HHLL représente une valeur binaire sur 16 bits (4 chiffres hexadécimaux).

*b) Les instructions LDAA et LDAB*

Ces instructions permettent de charger les registres A et B à l'aide

du contenu d'une case-mémoire donnée ou avec un nombre binaire (cas d'un adressage immédiat).

On aura donc:

M→A (instruction LDAA)
M→B (instruction LDAB)

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| LDAA #$LL | immédiat | 86 | N,Z,V=Ø |
| LDAA <$LL | direct | 96 | N,Z,V=Ø |
| LDAA $LL,X | indexé | A6 | N,Z,V=Ø |
| LDAA $HHLL | étendu | B6 | N,Z,V=Ø |
| LDAB #$LL | immédiat | C6 | N,Z,V=Ø |
| LDAB <$LL | direct | D6 | N,Z,V=Ø |
| LDAB $LL,X | indexé | E6 | N,Z,V=Ø |
| LDAB $HHLL | étendu | F6 | N,Z,V=Ø |

Nous allons illustrer cette instruction par un exemple très simple utilisant l'adressage étendu.

Soit l'instruction suivante:

```
LDAA   $5800
```

Le contenu des registres du 68Ø3 avant et après l'exécution de cette instruction est le suivant (on suppose que la case-mémoire d'adresse $58ØØ contient la valeur $Ø5 avant l'exécution de cette instruction).

| *Avant* | *Après* |
|---|---|
| A = $LL | A = $Ø5 |
| B = $LL | B = $LL |
| X = $HHLL | X = $HHLL |
| SP = $HHLL | SP = $HHLL |
| PC = $HHLL | PC = $HHLL+3 |
| CCR = 11XXXXXX | CCR = 11XXØØØX |

Comme vu précédemment $LL désigne un octet tandis que $HHLL désigne un mot de 16 bits.

Dans l'exemple ci-dessus A = $LL indique que A contient un octet non connu avant l'exécution de l'instruction. Après son exécution, A contient la valeur $Ø5.

Le registre de condition est donné sous forme binaire pour bien préciser les indicateurs positionnés par certaines instructions.

L'instruction LDAA affecte les bits N, Z et V du registre de condition CCR, comme nous l'avons vu plus haut.

Le chargement de la valeur $Ø5 provoque donc le résultat suivant :

N = Ø (résultat positif ou nul)
Z = Ø (résultat non nul)
V = Ø (LDAA remet à zéro cet indicateur)

Si l'instruction avait été similaire mais avec ($58ØØ) = $ØØ initialement ($58ØØ désignant le contenu de la case-mémoire d'adresse $58ØØ) alors les indicateurs N, Z et V auraient été positionnés de la manière suivante :

N = Ø (résultat positif ou nul)
Z = 1 (résultat nul)
V = Ø

De même si on avait eu ($58ØØ) = $A5, on aurait obtenu :

N = Ø (résultat négatif)
Z = Ø (résultat non nul)
V = Ø

#### 5.2.1.2. Les instructions sur 16 bits

Il s'agit de l'instruction de chargement de registre 16 bits LDD qui fonctionne de manière similaire aux instructions LDAA et LDAB vues précédemment ainsi que des instructions de chargement du registre d'index X et du pointeur de pile SP.

Les instructions LDD, LDX et LDS permettent respectivement de charger les registres D (constitué de la juxtaposition des registres A et

B), X et SP avec le contenu de la case-mémoire spécifiée ainsi que la suivante, ou avec une valeur binaire codée sur 16 bits (cas d'un adressage immédiat).

On aura donc :

M,M+1→D (instruction LDD)
M,M+1→SP (instruction LDS)
etc...

L'octet de poids fort du registre concerné est chargé en premier, c'est-à-dire avec le contenu de la case-mémoire d'adresse M.

L'octet de poids faible est chargé en second avec le contenu de la case-mémoire d'adresse M+1.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| LDD #$HHLL | immédiat | CC | N,Z,V=Ø |
| LDD <$LL | direct | DC | N,Z,V=Ø |
| LDD $LL,X | indexé | EC | N,Z,V=Ø |
| LDD $HHLL | étendu | FC | N,Z,V=Ø |
| LDS #$HHLL | immédiat | 8E | N,Z,V=Ø |
| LDS <$LL | direct | 9E | N,Z,V=Ø |
| LDS $LL,X | indexé | AE | N,Z,V=Ø |
| LDS $HHLL | étendu | BE | N,Z,V=Ø |
| LDX #$HHLL | immédiat | CE | N,Z,V=Ø |
| LDX <$LL | direct | DE | N,Z,V=Ø |
| LDX $LL,X | indexé | EE | N,Z,V=Ø |
| LDX $HHLL | étendu | FE | N,Z,V=Ø |

## 5.2.2. Les instructions de chargement mémoire

### 5.2.2.1. Les instructions de chargement 8 bits

Il s'agit des instructions STAA et STAB qui sont exactement l'équivalent des instructions LDAA et LDAB. En anglais les deux lettres "ST" veulent dire "STORE" (ranger). Donc par exemple STAA range le contenu de l'accumulateur A dans la case-mémoire d'adresse spécifiée.

On aura donc :

A→M (instruction STAA)
B→M (instruction STAB)

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| STAA <$LL | direct | 97 | N,Z,V=Ø |
| STAA $LL,X | indexé | A7 | N,Z,V=Ø |
| STAA $HHLL | étendu | B7 | N,Z,V=Ø |
| STAB <$LL | direct | D7 | N,Z,V=Ø |
| STAB $LL,X | indexé | E7 | N,Z,V=Ø |
| STAB $HHLL | étendu | F7 | N,Z,V=Ø |

En regardant ce tableau, on peut remarquer tout de suite qu'il n'y a pas d'adressage immédiat puisque l'instruction STAA (par exemple) range le contenu de l'accumulateur dans une case-mémoire d'adresse définie.

### 5.2.2.2. Les instructions de chargement 16 bits

De même qu'il existait des instructions de chargement de registre 16 bits LDD, LDX et LDS, il existe les instructions de chargement mémoire correspondantes qui sont donc STD, STX et STS.

On aura donc :

D→M,M+1 (instruction STD)
SP→M,M+1 (instruction STS)
etc...

Comme dans le cas des instructions de types "LOAD" l'octet de poids fort du registre concerne est chargé en premier (case-mémoire M) et l'octet de poids faible est chargé en second (case-mémoire M+1).

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| STD <$LL | direct | DD | N,Z,V=Ø |
| STD $LL,X | indexé | ED | N,Z,V=Ø |
| STD $HHLL | étendu | FD | N,Z,V=Ø |
| STS <$LL | direct | 9F | N,Z,V=Ø |

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| STS $LL,X | indexé | AF | N,Z,V=Ø |
| STS $HHLL | étendu | BF | N,Z,V=Ø |
| STX <$LL | direct | DF | N,Z,V=Ø |
| STX $LL,X | indexé | EF | N,Z,V=Ø |
| STX $HHLL | étendu | FF | N,Z,V=Ø |

Nous allons illustrer les instructions LDD et STD à l'aide d'un exemple.

Soit le petit programme très simple suivant:

```
        ORG     $4800
        EXC     PROG
PROG    LDD     #$1000
        STD     $4900
        RTS
```

Après avoir tapé par exemple CLEAR 1000,15000, entrez ce programme à l'aide de l'Éditeur-Assembleur.

L'instruction LDD #$1000 permet de charger le registre D à l'aide de la valeur $1000. A, qui est l'octet de poids fort contient alors la valeur $1Ø tandis que B, qui est l'octet de poids faible, contient la valeur $ØØ.

Grâce à l'instruction STD $4900, il est possible de sauvegarder le contenu du registre D en mémoire à l'adresse $4900.

En : $4900 = 18688 on trouve donc 16 = $1Ø
et en : $4901 = 18689 on trouve donc Ø = $ØØ

Apès avoir fait exécuter ce petit programme, vous pourrez vérifier le contenu des cases-mémoire d'adresses 18688 et 18689 en tapant sous Basic :

```
    PRINT PEEK(18688)
et  PRINT PEEK(18689)
```

Le contenu des registres internes du 68Ø3 après exécution de l'instruction LDD est le suivant:

A = $1Ø
B = $ØØ
X = $HHLL
SP = $HHLL
PC = $HHLL
CCR = 11XXØØØX

### 5.2.3. Les instructions de transfert de registres

Ces instructions permettent de transférer le contenu d'un registre dans un autre.

Le 68Ø3 possède dans cette catégorie les instructions suivantes:

TXS (transfert du registre X dans le registre SP: X−1→SP)
TSX (transfert du registre SP dans le registre X: SP+1 → X)
TAB (transfert du registre A dans le registre B: A → B)
TBA (transfert du registre B dans le registre B: B → A)
TAP (transfert du registre A dans le registre CCR: A → CCR)
TPA (transfert du registre CCR dans le registre A: CCR → A)

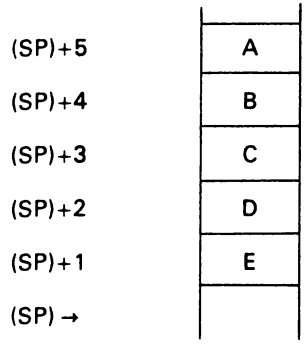
TXS et TSX ne sont pas exactement des instructions de transfert puisque dans un cas, X−1 est transféré dans SP, dans l'autre SP+1 est transféré dans X. Ce point est une particularité intéressante du 68Ø3 comme nous allons le voir un peu plus loin.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| TSX | implicite | 3Ø | néant |
| TXS | implicite | 35 | néant |
| TAB | implicite | 16 | N,Z,V=Ø |
| TBA | implicite | 17 | N,Z,V=Ø |
| TAP | implicite | Ø6 | tous |
| TPA | implicite | Ø7 | néant |

L'instruction TAP permet de programmer le contenu du registre CCR. Il est ainsi possible de restaurer ainsi une ancienne valeur préalablement sauvegardée par une instruction TPA.

Les instructions TXS et TSX sont très utiles pour rechercher des tableaux de données en mémoire.

Considérons la pile contenant les 5 données A, B, C, D et E. L'allure de cette pile est la suivante:




Le pointeur de pile pointe toujours sur la première case disponible équivalent au sommet de la pile.

Les données A, B, C, D et E sont donc situées respectivement aux adresses (SP)+5, (SP)+4, (SP)+3, (SP)+2 et (SP)+1, (SP) désignant le contenu du registre pointeur de pile.

Une instruction TSX transfère donc dans le registre d'index X l'adresse même de la donnée E. En incrémentant ensuite le registre X, il est possible d'accéder successivement aux données D, C, B et A. (Nous reviendrons sur les instructions d'incrémentation un peu plus loin dans ce chapitre).

Inversement, après avoir stocké en mémoire un tableau de données à l'aide d'un adressage indexé, il est possible de conserver celui-ci dans la pile grâce à une instruction TXS.

Le petit programme suivant illustre ces deux instructions :

```
        ;PROGRAMME DE DEMONSTRATION
        ;DES INSTRUCTIONS TSX ET TXS
        ORG    $4800
        EXC    PROG
        ;SAUVEGARDE DES DONNEES
PROG    LDAA   #$01          ;01 DANS $4900
        LDX    #$4900
        STAA   $00,X
        LDAA   #$02          ;02 DANS $4901
        INX
        STAA   $00,X
        LDAA   #$03          ;03 DANS $4902
        INX
        STAA   $00,X
        TXS                  ;X-1 -> SP
        ;RESTAURATION DES DONNEES
        TSX                  ;SP+1-> X
        LDAA   $00,X         ;03 DANS $4920
        STAA   $4920
        DEX
        LDAA   $00,X         ;02 DANS $4921
        STAA   $4921
        DEX
        LDAA   $00,X         ;01 DANS $4922
        STAA   $4922
        BRA    PROG
```

Après exécution de ce programme, on obtient en mémoire les valeurs suivantes :

```
PEEK (18688) = 1
PEEK (18689) = 2
PEEK (18690) = 3
PEEK (18720) = 3
PEEK (18721) = 2
PEEK (18722) = 1
```

Ce petit programme n'a bien sûr qu'une valeur éducative. Nous verrons plus loin qu'il est possible de sauvegarder des données directement à l'aide d'instructions travaillant sur la pile.

Nous venons de décrire le fonctionnement des instructions les plus employées dans tout programme "tournant sur 6803". Inutile de dire

qu'il est nécessaire que vous en ayez saisi, sinon toutes les subtilités, du moins les bases de fonctionnement.

## 5.3. LES INSTRUCTIONS ARITHMÉTIQUES

Nous appelerons instructions arithmétiques les instructions suivantes :

— les instructions d'addition (8 bits et 16 bits) : ADCA, ADCB, ADDA, ADDB, ADDD, ABA, ABX ;

— les instructions de soustraction (8 bits et 16 bits) : SBCA, SBCB, SUBA, SUBB, SUBD, SBA ;

— l'instruction MUL (multiplication) ;

— les instructions d'incrémentation : INC, INCA, INCB, INX, INS ;

— les instructions de décrémentation : DEC, DECA, DECB, DEX, DES ;

— les instructions de décalage arithmétique : ASR, ASRA, ASRB, ASL, ASLA, ASLB, ASLD ;

— les instructions de négation : NEG, NEGA, NEGB ;

— l'instruction DAA (nous expliciterons cette instruction un peu plus loin).

### 5.3.1. Les instructions d'addition

#### 5.3.1.1. Notion d'addition sur les nombres binaires

Soit à additionner les deux nombres hexadécimaux suivants : $Ø5 et $17.

On sait que ces deux nombres représentent respectivement Ø5 et 23 en décimal.

Leur représentation binaire est :

$Ø5 = ØØØØ Ø1Ø1 $17 = ØØØ1 Ø111

Ø 5 1 7

Une addition en binaire est en tous points identique à une addition en décimal sauf que les chiffres utilisés, au lieu d'être Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9 sont Ø et 1.

On a donc:

Ø+Ø = Ø
Ø+1 = 1
1+Ø = 1
1+1 = 1Ø

L'addition de $Ø5 et $17 donnera:

| $Ø5 + $17 | Ø5 + 23 | ØØØØØ1Ø1 + ØØØ1Ø111 |
|---|---|---|
| = $1C | = 28 | = ØØØ11100 (1 C) |

Cela n'a rien de très compliqué.

Ceci dit, l'arithmétique binaire possède quelques subtilités, mais nous allons tout d'abord faire un retour sur le registre de condition CCR.

#### 5.3.1.2. Retour sur le registre CCR

Nous allons en particulier étudier les 3 bits H, V et C, le bit N ayant déjà été étudié.

Nous avons vu précédemment le rôle succint de ces indicateurs.

C = indicateur de retenue,

H = indicateur de demi-retenue,

V = indicateur de débordement.

*a) Le bit C*

Supposons que nous ayons à effectuer la somme de deux nombres $8A et $D5.

En représentation binaire on a:

$8A = 10001010
$D5 = 11010101

La somme de ces deux nombres donne:

```
   $8A                    10001010
 + $D5                  + 11010101
-------------------------------------
 =$15F                  =(1)01011111
```

on voit que le résultat est un nombre de 9 bits. Le bit de retenue C est positionné à 1 chaque fois qu'il y a une retenue sur la somme des bits de poids fort des deux nombres binaires considérés.

Donc pour $Ø5 + $17 = $1C on a C=Ø
et pour $8A + $D5 = $15F on a C=1.

*b) Le bit H*

Le bit H fonctionne exactement de la même façon que le bit C. Cependant, au lieu de détecter un dépassement de capacité au niveau du bit 7, il détecte un dépassement de capacité au niveau du bit 3. Ceci est utile lors d'opérations sur des nombres codés BCD. Nous reviendrons sur ce point avec l'étude de l'instruction DAA.

Par exemple, si nous voulons additionner les deux nombres $Ø9 et $17 ($Ø9 + $17 = $2Ø), le bit H sera positionné à 1.

Par contre, si nous voulons additionner les deux nombres $Ø2 et $17 ($Ø2 + $17 = $19), le bit H restera à Ø.

*c) Le bit V*

Il faut d'abord retenir une chose: le microprocesseur effectue de la même manière l'addition de deux nombres binaires qu'ils soient signés ou non (ceci est d'ailleurs également valable dans le cas de nombres codés BCD). C'est au programmeur d'en décider et de se fixer une convention.

Nous avons vu précédemment que si la somme de deux nombres binaires dépassait la capacité du microprocesseur il y avait positionnement à 1 du bit C. Cela se comprend aisément dans le cas de deux nombres compris entre Ø et 255, mais que se passe-t-il lorsque les deux nombres traités sont considérés comme signés par le programmeur?

Nous savons que le bit 7 est le bit de signe ; par définition, il y aura positionnement à 1 du bit indicateur de dépassement lorsque :

— soit il y a une retenue du bit 6 vers le bit 7, ceci sans retenue du type "carry" vue précédemment,

— soit il n'y a pas de retenue du bit 6 vers le bit 7, mais par contre il y a une retenue de type "carry".

Examinons quelques exemples : nous voulons additionner **$4B** et **$71** :

```
$4B = 01001011
$71 = 01110001

  $4B      01001011
+ $71    + 01110001
-----------------------
= $BC    = 10111100
```

On a dans ce cas V=1 (retenue du bit 6 vers le bit 7 sans carry) et C=Ø.

Si les deux nombres **$4B** et **$71** sont considérés comme signes, ils sont tous deux positifs (75 et 113) mais leur somme, qui est positive, dépasse la capacité du microprocesseur (188 est supérieur à 127) donne donc un résultat négatif.

Il est donc nécessaire d'indiquer que le résultat est erroné. C'est le rôle du bit V qui dans ce cas est positionné à 1.

Ici **$4B** + **$71** est égal à **$BC** alors que le résultat trouvé est égal à **$–42**.

Nous allons voir maintenant ce que donne l'addition des deux nombres **$–Ø1** et **$–Ø5**.

```
  $-01                  11111111
+ $-05                + 11111011
-----------------------------------
= $-06              =(1)11111010
```

Dans ce cas ci, le bit V est positionné à Ø (retenue du bit 6 vers le bit 7 et carry simultanée) et le bit C à 1.

Le résultat trouvé est égal à $–Ø6 ce qui est exactement le résultat escompté.

Nous voyons donc que dans certains cas, la somme de deux nombres signés est exacte alors que dans d'autres elle est erronée.

Vous pourrez vérifier de vous-même à l'aide d'exemples que lorsque V est positionné à Ø le résultat de l'addition de deux nombres signes est exact alors que quand il est positionné à 1, le résultat est faux et nécessite une correction adéquate afin de pouvoir être utilisé ultérieurement.

#### 5.3.1.3. Les instructions sur 8 bits

Il s'agit, nous l'avons vu, des instructions ADCA, ADCB, ADDA, ADDB et ABA.

Les deux premières qui signifient en anglais "ADD WITH CARRY" (addition avec retenue) permettent d'ajouter à l'accumulateur le contenu d'une case-mémoire spécifiée (ou bien une valeur binaire dans le cas d'un adressage immédiat) ainsi que le bit de retenue C, le résultat étant placé dans l'accumulateur A ou B suivant le cas.

On a donc l'opération suivante:

A + bit C + M→A

ou bien:

B + bit C + M→A

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ADCA #$LL | immédiat | 89 | H,N,Z,V,C |
| ADCA <$LL | direct | 99 | H,N,Z,V,C |
| ADCA $LL,X | indexé | A9 | H,N,Z,V,C |
| ADCA $HHLL | étendu | B9 | H,N,Z,V,C |
| ADCB #$LL | immédiat | C9 | H,N,Z,V,C |
| ADCB <$LL | direct | D9 | H,N,Z,V,C |
| ADCB$LLX | indexé | E9 | H,N,Z,V,C |
| ADCB $HHLL | étendu | F9 | H,N,Z,V,C |

Considérons l'addition des deux nombres hexadécimaux $4B et $71 ainsi que du bit de retenue égal à 1.

Nous allons réaliser cette opération à l'aide du programme suivant :

```
        ORG     $4000
        EXC     PROG
PROG    SEC             ;C=1
        LDAA    #$4B    ;CHARGE 1ERE VALEUR
        ADCA    #$71    ;ADDITIONNE 2EME VALEUR
        STAA    $4020   ;SAUVEGARDE EN MEMOIRE
        RTS
```

L'instruction SEC permet, comme nous le verrons plus loin, de positionner à 1 le bit de retenue C.

Après avoir fait exécuter ce programme, tapez sous interpréteur Basic :

```
PRINT PEEK(16416)
```

ce qui doit donner la valeur 189 = $BD = $4B + $71 + 1.

Si l'on avait positionné à Ø le bit C une instruction CLC (qui sera également vue plus loin), le résultat aurait été alors :

188 = $BC = $4B + $71

Les instructions ADDA et ADDB fonctionnent comme les instructions ADCA et ADCB sauf que le bit C n'est pas ici pris en considération.

On aura donc :

A + M → A

ou bien :

B + M → B

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ADDA #$LL | immédiat | 8B | H,N,Z,V,C |
| ADDA <$LL | direct | 9B | H,N,Z,V,C |
| ADDA $LL,X | indexé | AB | H,N,Z,V,C |

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ADDA $HHLL | étendu | BB | H,N,Z,V,C |
| ADDB #$LL | immédiat | CB | H,N,Z,V,C |
| ADDB <$LL | direct | DB | H,N,Z,V,C |
| ADDB $LL,C | indexé | EB | H,N,Z,V,C |
| ADDB $HHLL | étendu | FB | H,N,Z,V,C |

L'instruction ABA permet d'additionner le contenu des accumulateurs A et B, le résultat étant stocké dans A.

On a donc:

$A + B \rightarrow A$

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ABA | implicite | 1B | H,N,Z,V,C |

### 5.3.1.4. L'instruction DAA

Cette instruction est utilisée dans le cas d'opérations portant sur des nombres codes BCD. Elle permet, comme son nom l'indique, d'effectuer un ajustement décimal sur l'accumulateur A.

Cet ajustement fonctionne de la manière suivante:

— si le contenu des 4 bits de poids faible de A est supérieur à 9 ou si l'indicateur de demi-retenue H est égal à 1, alors la valeur 6 est additionnée aux 4 bits de poids faible de A.

— si, après l'opération ci-dessus effectuée, le contenu des 4 bits de poids fort de A est supérieur à 9 ou si l'indicateur de demi-retenue H est égal à 1, alors la valeur 6 est additionnée aux 4 bits de poids fort de A.

Par exemple, considérons les deux nombres BCD 8 et 7.
On a:

8 = ØØØØ1ØØØ
7 = ØØØØØ111

8+7 = ØØØØ1111 ≠ 15 en BCD

Le nombre binaire obtenu est $ØF au lieu de 15 attendu en arithmétique BCD. Si l'on ajoute la valeur 6 au résultat ci-dessus on obtient la valeur ØØØ1Ø1Ø1 qui correspond exactement à 15 codé BCD.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| DAA | implicite | 19 | N,Z,V,C |

Le programme suivant permet d'effectuer une addition BCD sur 16 bits. Son fonctionnement est très simple et n'amène aucun commentaire.

```
        ORG    $4000
        EXC    PROG
        ;PROGRAMME D'ADDITION BCD 16 BITS
NAL     DFO    $12   ;OCTET DE POIDS FAIBLE A
NAH     DFO    $35   ;OCTET DE POIDS FORT A
NBL     DFO    $24   ;OCTET DE POIDS FAIBLE B
NBH     DFO    $45   ;OCTET DE POIDS FORT B
RL      DFO    $ØØ   ;OCTET DE POIDS FAIBLE RESULTAT
RH      DFO    $ØØ   ;OCTET DE POIDS FORT RESULTAT
PROG    LDAA   NAL   ;CALCULE SOMME OCTETS DE POIDS FAIBLE
        ADDA   NBL
        DAA          ;AJUSTEMENT DECIMAL
        STAA   RL    ;SAUVEGARDE POIDS FAIBLE RESULTAT
        LDAA   NAH
        ADCA   NBH
        DAA
        STAA   RH    ;SAUVEGARDE POIDS FORT RESULTAT
        RTS
```

#### 5.3.1.5. Les instructions sur 16 bits

Il s'agit des instructions ADDD et ABX.

L'instruction ADDD permet d'ajouter à l'ensemble formé par les accumulateurs A et B le contenu de la case-mémoire spécifiée et de la suivante (ou une valeur sur 16 bits dans le cas d'un adressage immédiat).

On aura donc :

D + M,M+1→D

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ADDD #$HHLL | immédiat | C3 | N,Z,V,C |
| ADDD <$LL | direct | D3 | N,Z,V,C |
| ADDD $LL,X | indexé | E3 | N,Z,V,C |
| ADDD $HHLL | étendu | F3 | N,Z,V,C |

Considérons l'exemple suivant :

```
        ORG    $4000
        EXC    PROG
        ;STOCKE 1ERE VALEUR
PROG    LDAA   #$10   ;OCTET DE POIDS FORT
        STAA   $4020
        LDAA   #$20   ;OCTET DE POIDS FAIBLE
        STAA   $4021
        ;CHARGE 2EME VALEUR
        LDD    #$2020
        ADDD   $4020
        ;SAUVEGARDE SOMME EN MEMOIRE
        STD    $4022
        RTS
```

Après avoir fait exécuter ce programme, vous pouvez vérifier sous Basic que :

(16416) = ($4020) = 16 = $10 = octet de poids fort de la 1re valeur
(16417) = ($4021) = 32 = $20 = octet de poids faible de la 1re valeur
(16418) = ($4022) = 48 = $30 = octet de poids fort du résultat
(16419) = ($4023) = 64 = $40 = octet de poids faible du résultat

L'instruction ABX permet d'ajouter à l'accumulateur B le contenu du registre X.

On a donc :

B + X → X

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ABX | implicite | 3A | néant |

### 5.3.2. Les instructions de soustraction

#### 5.3.2.1. Notion de soustraction sur les nombres binaires

Il faut tout d'abord savoir qu'une soustraction n'est rien d'autre qu'une addition particulière. En effet, on additionne au premier nombre l'inverse du second. C'est ainsi qu'en arithmétique décimale classique on a :

$$13 - 10 = 13 + (-10)$$

En arithmétique binaire on additionnera donc le complément à deux. Supposons que nous voulions effectuer la soustraction suivante :

$$\$20 - \$15 = \$20 + (\$-15)$$

donc :

$\$20 = \quad 00100000$
$\$15 = \quad 00010101$
$\$-15 = \quad 11101011$
$\$20-\$15 = (1)00001011 = \$10B$

retenue ↑

La différence $\$20-\$15$ est un nombre positif ($\$0B$) et la retenue est alors égale à 1.

Supposons maintenant que nous voulions faire $\$15-\$20$ :

$$\$-20 = 11100000$$

donc : $\$15-\$20 = 11110101 = \$F5 = \$-0B$

Cette fois-ci, la retenue est égale à zéro et le résultat est négatif. Une soustraction binaire s'effectue alors très simplement et on a les résultats suivants :

C = 1 : le résultat est positif

C = 0 : le résultat est négatif.

#### 5.3.2.2. Les instructions sur 8 bits

Il s'agit des instructions SBCA, SBCB, SUBA, SUBB et SBA.

Les instructions SBCA et SBCB sont l'analogue des instructions ADCA et ADCB et signifient donc "soustraction avec retenue".

Le contenu de la case-mémoire spécifiée (ou la valeur binaire dans le cas d'un adresage immédiat), ainsi que le bit C du registre CCR sont retranchés à l'accumulateur A ou B. Le résultat est placé dans l'accumulateur.

On a donc :

A – bit C – M→A

ou bien :

B – bit C – M→B

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| SBCA #$LL | immédiat | 82 | N,Z,V,C |
| SBCA <$LL | direct | 92 | N,Z,V,C |
| SBCA $LL,X | indexé | A2 | N,Z,V,C |
| SBCA $HHLL | étendu | B2 | N,Z,V,C |
| SBCB #$LL | immédiat | C2 | N,Z,V,C |
| SBCB <$LL | direct | D2 | N,Z,V,C |
| SBCB $LL,C | indexé | E2 | N,Z,V,C |
| SBCB $HHLL | étendu | F2 | N,Z,V,C |

De même les instructions SUBA et SUBB sont les analogues des instructions ADDA et ADDB. La soustraction s'effectue alors sans retenue.

On a donc :

A – M→A

ou bien :

B – M→B

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| SUBA #$LL | immédiat | 8Ø | N,Z,V,C |
| SUBA <$LL | direct | 9Ø | N,Z,V,C |
| SUBA $LL,X | indexé | AØ | N,Z,V,C |
| SUBA $HHLL | étendu | BØ | N,Z,V,C |
| SUBB #$LL | immédiat | CØ | N,Z,V,C |
| SUBB <$LL | direct | DØ | N,Z,V,C |
| SUBB $LL,X | indexé | EØ | N,Z,V,C |
| SUBB $HHLL | étendu | FØ | N,Z,V,C |

L'instruction SBA est l'analogue de l'instruction ABA et permet donc de retrancher le contenu de B à celui de A.

On a donc :

A – B → A

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| SBA | implicite | 1Ø | N,Z,V,C |

#### 5.3.2.3. Les instructions sur 16 bits

Il s'agit de l'instruction SUBD.

On a donc :

D – M,M+1 → D

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| SUBD #$HHLL | immédiat | 83 | N,Z,V,C |
| SUBD <$LL | direct | 93 | N,Z,V,C |
| SUBD $LL,X | indexé | A3 | N,Z,V,C |
| SUBD $HHLL | étendu | B3 | N,Z,V,C |

### 5.3.3. Les instructions d'incrémentation

Nous regroupons sous cette appellation les instructions INC, INCA, INCB, INX et INS. L'instruction INC effectue une incrémentation en mémoire tandis que les instructions INCA, INCB, INX et INS permettent respectivement d'incrémenter le contenu des registres A, B, X et S. Il s'agit pour ces quatre dernières d'un adressage implicite.

#### 5.3.3.1. Notion d'incrémentation

Il s'agit en fait de quelque chose de très simple : l'incrémentation consiste à ajouter 1 au contenu de la case-mémoire désignée par l'opérande.

*Exemple* :

$Ø5 donne $Ø6

### 5.3.3.2. Les instructions INC, INCA, INCB, INX et INS

On aura donc :

M + 1→M (cas de l'instruction INC)

A + 1→A (cas de l'instruction INCA)

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| INC $LL,X | indexé | 6C | N,Z,V |
| INC $HHLL | étendu | 7C | N,Z,V |
| INCA | implicite | 4C | N,Z,V |
| INCB | implicite | 5C | N,Z,V |
| INX | implicite | Ø8 | Z |
| INS | implicite | 31 | néant |

## 5.3.4. Les instructions de décrémentation

Nous regrouperons sous cette appellation les instructions DEC, DECA, DECB, DEX et DES.

### 5.3.4.1. Notion de décrémentation

Le fonctionnement est exactement le même que pour une incrémentation sauf que l'on retranche 1 au contenu de la case-mémoire désignée par l'opérande.

*Exemple* :

$Ø5 donne $Ø4

### 5.3.4.2. Les instructions DEC, DECA, DECB, DEX et DES

On aura donc :

M – 1→M (cas de l'instruction DEC)

B – 1→B (cas de l'instruction DECB)

etc...

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| DEC $LL,X | indexé | 6A | N,Z,V |
| DEC $HHLL | étendu | 7A | N,Z,V |
| DECA | implicite | 4A | N,Z,V |
| DECB | implicite | 5A | N,Z,V |
| DEX | implicite | Ø9 | Z |
| DES | implicite | 34 | néant |

Le programme ci-dessous illustre le fonctionnement de l'instruction DEX et permet de réaliser ce que l'on appelle un délai logiciel.

En fait cela consiste à faire exécuter au microprocesseur un certain nombre de fois la même boucle :

```
        ORG     $4ØØØ
        EXC     PROG
PROG    LDX     #$FFFF    ;NOMBRE DE BOUCLES
LOOP    DEX               ;TERMINE?
        BNE     LOOP      ;NON, RECOMMENCE
        RTS
```

### 5.3.5. L'instruction MUL

Cette instruction permet d'effectuer la multiplication des contenus des registres A et B considérés comme des nombres non signés. Le résultat est stocké dans le registre D formé par la réunion de ces deux registres.

En effet, la multiplication de 2 nombres de 8 bits donne un résultat sur 16 bits.

La présence de cette instruction est une particularité du 68Ø3 rarement trouvée sur les microprocesseurs 8 bits du marché.

Elle permet une exécution beaucoup plus rapide des calculs arithmétiques la mettant en œuvre.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| MUL | implicite | 3D | C |

*Exemple*: Soit à effectuer la multiplication des deux nombres:

65 = $41 et 34 = $22

Le petit programme suivant permet de calculer ce produit:

```
        ORG     $4000
        EXC     PROG
PROG    LDAA    #$41        ;A=MULTIPLICATEUR
        LDAB    #$22        ;B=MULTIPLICANDE
        MUL
        STD     $4020       ;SAUVEGARDE RESULTAT
        RTS
```

Aux adresses 16416 = $4Ø2Ø et 16417 = $4Ø21 on trouve respectivement les octets de poids fort (8 = $Ø8) et de poids faible (162 = $A2) du résultat.

### 5.3.6. Les instructions de négation

Il s'agit des instructions NEG, NEGA et NEGB. Elles donnent le complément à deux du contenu de la case-mémoire spécifiée ou des registres A et B.

On a donc:

$\overline{A} + 1 \rightarrow A$ soit $Ø - A \rightarrow A$

ou bien:

$\overline{B} + 1 \rightarrow B$ soit $Ø - B \rightarrow B$

ou bien :

$\bar{M} + 1 \rightarrow M$ soit $\emptyset - M \rightarrow M$

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| NEG $LL,X | indexé | 6Ø | N,Z,V,C |
| NEG $HHLL | étendu | 7Ø | N,Z,V,C |
| NEGA | implicite | 4Ø | N,Z,V,C |
| NEGB | implicite | 5Ø | N,Z,V,C |

### 5.3.7. Les instructions de décalage arithmétique

Il s'agit des instructions ASL, ASLA, ASLB, ASLD, ASR, ASRA et ASRB.

— ASL : Arithmetic Shift Left : décalage arithmétique vers la gauche ;

— ASR : Arithmetic Shift Right : décalage arithmétique vers la droite.

#### 5.3.7.1. Fonctionnement des instructions de décalage arithmétique

*a) ASL*

Chaque bit est décalé d'un rang vers la gauche. Le bit Ø est remplacé par un "Ø" et le bit 7 devient le bit de retenue (bit C). On notera que le bit de signe (bit 7) est conservé dans la retenue d'où le nom de décalage "arithmétique".

Nous verrons plus loin en effet que le bit de signe n'est pas conservé dans le cas d'un décalage logique.

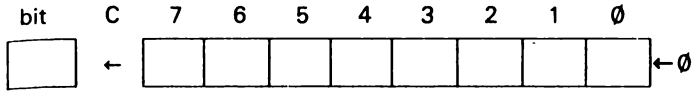
*Exemple* : Soit l'instruction ASLA (adressage implicite).

On suppose que A = $9D initialement.

$9D = 10011101

Après décalage cela nous donne $3A = ØØ111Ø1Ø et C=1.

D'une manière générale, nous avons donc le schéma suivant :




*b) ASR*

Chaque bit est ici décalé d'un rang vers la droite. Le bit Ø devient le bit de retenue (bit C) tandis que le bit de signe (bit 7) est recopié à la place qu'il occupait précédemment.

Là encore, l'appellation "décalage arithmétique" est justifiée par le fait que le bit de signe est conservé.

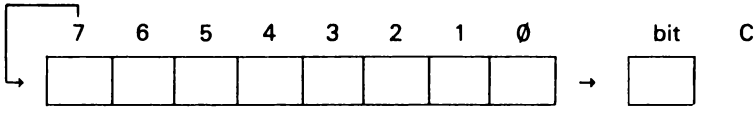
*Exemple* : Soit l'instruction ASRA.

On suppose que A = $9D initialement.

Après décalage cela nous donne :

$CE = 11001110 et C=1

D'une manière générale, nous avons donc le schéma suivant :




### 5.3.7.2. Les instructions ASL, ASLA, ASLB, ASLD

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ASL $LL,X | indexé | 68 | N,Z,V,C |
| ASL $HHLL | étendu | 78 | N,Z,V,C |
| ASLA | implicite | 48 | N,Z,V,C |
| ASLB | implicite | 58 | N,Z,V,C |
| ASLD | implicite | Ø5 | N,Z,V,C |

Notons que le décalage à gauche sur le registre D fonctionne exactement comme les décalages sur A ou B, mais cette fois-ci sur 16 bits.

### 5.3.7.3. Les instructions ASR, ASRA et ASRB

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ASR $LL,X | indexé | 67 | N,Z,V,C |
| ASR $HHLL | étendu | 77 | N,Z,V,C |
| ASRA | implicite | 47 | N,Z,V,C |
| ASRB | implicite | 57 | N,Z,V,C |

*Exemple*: Soit le programme suivant permettant de décaler vers la droite le contenu de l'Accumulateur A = $9D.

```
        ORG     $4000
        EXC     PROG
PROG    LDAA    #$9D
        ASRA
        STAA    $4020
        RTS
```

Le résultat obtenu à l'adresse 16416 = $4020 est égal à 206 = $CE.

## 5.4. LES INSTRUCTIONS LOGIQUES

Nous regrouperons sous cette appellation les instructions suivantes:

— le "ET" logique: ANDA, ANDB

— le "OU" logique: ORAA, ORAB

— le "OU" exclusif: EORA, EORB

— les instructions de complémentation: COM, COMA, COMB

— les instructions de décalage logique: LSL, LSLA, LSLB, LSLD, LSR, LSRA, LSRB, LSRD

— les instructions de rotation: ROL, ROLA, ROLB, ROR, RORA, RORB

— les instructions de test de bits: BITA, BITB, TST, TSTA, TSTB.

## 5.41. Le "ET" logique

### 5.4.1.1. Notion de "ET" logique

L'opération "ET" appartient à une catégorie un peu particulière qui est l'ALGEBRE DE BOOLE. Dans l'algèbre booléenne, le système numérique utilisé est le binaire. Une opération s'effectue entre deux chiffres binaires et donne comme résultat un chiffre binaire unique. Les opérations booléennes présentes dans le 68Ø3 sont le "ET", le "OU", le "OU exclusif" et le "NON".

Le "ET" logique: on considère deux chiffres A et B binaires. L'opération "ET" (notée souvent Λ) est définie de la manière suivante:

si A = B = 1 alors A Λ B = 1
sinon A Λ B = Ø

On peut définir une table de vérité pour cette opération.

| B \ A | Ø | 1 |
|---|---|---|
| Ø | Ø | Ø |
| 1 | Ø | 1 |

Les chiffres inscrits dans les 4 cases centrales donnent la valeur de A Λ B pour chaque combinaison de A et B.

Extension de la notion de "ET" logique à un octet:

Le "ET" s'effectue bit par bit.

*Exemple*: Soit à effectuer:

$12∧$35

on a:

$12 = ØØØ1ØØ1Ø
$35 = ØØ11Ø1Ø1

$12∧$35 = ØØØ1ØØØØ = $1Ø

#### 5.4.1.2. Les instructions ANDA et ANDB

Ces instructions effectuent le "ET" logique entre le contenu d'une case-mémoire déterminée (ou une valeur binaire dans le cas d'un adressage immédiat) et l'un des accumulateurs A ou B, le résultat final étant stocké dans ce dernier.

On a donc :

$A \wedge M \rightarrow A$ (cas de l'instruction ANDA)

ou bien :

$B \wedge M \rightarrow B$ (cas de l'instruction ANDB)

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ANDA #$LL | immédiat | 84 | N,Z,V=Ø |
| ANDA <$LL | direct | 94 | N,Z,V=Ø |
| ANDA $LL,X | indexé | A4 | N,Z,V=Ø |
| ANDA $HHLL | étendu | B4 | N,Z,V=Ø |
| ANDB #$LL | immédiat | C4 | N,Z,V=Ø |
| ANDB <$LL | direct | D4 | N,Z,V=Ø |
| ANDB $LL,X | indexé | E4 | N,Z,V=Ø |
| ANDB $HHLL | étendu | F4 | N,Z,V=Ø |

### 5.4.2. Le "OU" logique

#### 5.4.2.1. Notion de "OU" logique

L'opération "OU" (notée souvent V) est définie de la manière suivante :

Si :

$A = B = Ø$ alors $A \vee B = Ø$

sinon :

$A \vee B = 1$

La table de vérité de cette opération est la suivante:

| B \ A | Ø | 1 |
|---|---|---|
| Ø | Ø | 1 |
| 1 | 1 | 1 |

Comme dans le cas de l'opération "ET", le "OU" entre 2 octets s'effectue bit par bit.

*Exemple*:

Soit à effectuer $12 V $35

$12 = ØØØ1ØØ1Ø
$35 = ØØ11Ø1Ø1

$12V$35 = ØØ11Ø111 = $37

### 5.4.2.2. Les instructions ORAA et ORAB

Ces instructions effectuent le "OU" logique entre le contenu d'une case-mémoire donnée (ou une valeur binaire dans le cas d'un adressage immédiat) et l'un des accumulateurs A ou B.

On a donc:

A∨M→A (cas de l'instruction ORAA)
B∨M→B (cas de l'instruction ORAB)

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ORAA #$LL | immédiat | 8A | N,Z,V=Ø |
| ORAA <$LL | direct | 9A | N,Z,V=Ø |
| ORAA $LL,X | indexé | AA | N,Z,V=Ø |
| ORAA $HHLL | étendu | BA | N,Z,V=Ø |
| ORAB #$LL | immédiat | CA | N,Z,V=Ø |
| ORAB <$LL | direct | DA | N,Z,V=Ø |
| ORAB $LL,X | indexé | EA | N,Z,V=Ø |
| ORAB $HHLL | étendu | FA | N,Z,V=Ø |

### 5.4.3. Le "OU" exclusif

#### 5.4.3.1. Notion de "OU" exclusif

L'opération "OU exclusif" (notée souvent $\forall$) est définie de la manière suivante :

si A = B alors A $\forall$ B = Ø
si A ≠ B alors A $\forall$ B = 1

En d'autres termes, le résultat d'un "OU exclusif" entre deux chiffres binaires est 1 si un et un seul de ces deux chiffres est égal à 1.

La table de vérité de cette opération est la suivante :

| B \ A | Ø | 1 |
|---|---|---|
| Ø | Ø | 1 |
| 1 | 1 | Ø |

Le "OU exclusif" sur un octet s'effectue bit par bit comme dans le cas du "ET" et du "OU".

*Exemple* :

Soit à effectuer \$12$\forall$\$35

\$12 = ØØØ1ØØ1Ø
\$35 = ØØ11Ø1Ø1

\$12$\forall$\$35 = ØØ1ØØ111 = \$27

#### 5.4.3.2. Les instructions EORA et EORB

Ces opérations effectuent le "OU exclusif" entre le contenu d'une case-mémoire spécifiée (ou une valeur binaire dans le cas d'un adressage immédiat) et l'un des accumulateurs A et B.

On a donc :

A$\forall$M→A (cas de l'instruction EORA)

ou bien :

B$\forall$M→B (cas de l'instruction EORB)

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| EORA #$LL | immédiat | 88 | N,Z,V=Ø |
| EORA <$LL | direct | 98 | N,Z,V=Ø |
| EORA $LL,X | indexé | A8 | N,Z,V=Ø |
| EORA $HHLL | étendu | B8 | N,Z,V=Ø |
| EORB #$LL | immédiat | C8 | N,Z,V=Ø |
| EORB <$LL | direct | D8 | N,Z,V=Ø |
| EORB $LL,X | indexé | E8 | N,Z,V=Ø |
| EORB $HHLL | étendu | F8 | N,Z,V=Ø |

## 5.4.4. Les instructions de complémentation

### 5.4.4.1. Notion de complémentation

L'opération "complémentation" est définie de la manière suivante :
si A = 1 alors $\overline{A} = Ø$ ($\overline{A}$ est le complément de A)
si A = Ø alors $\overline{A} = 1$

*Exemple* : Soit à effectuer $12.

$12 = ØØØ1ØØ1Ø alors $\overline{\$12}$ = 111Ø11Ø1

### 5.4.4.2. Les instructions COM, COMA et COMB

Ces instructions permettent de complémenter soit les registres A, B, soit le contenu de la case-mémoire d'adresse spécifiée.

On a donc :

$\overline{A} \rightarrow A$ (cas de l'instruction COMA)
$\overline{B} \rightarrow B$ (cas de l'instruction COMB)
$\overline{M} \rightarrow M$ (cas de l'instruction COM)

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| COMA | implicite | 43 | N,Z,C=1,V=Ø |
| COMB | implicite | 53 | N,Z,C=1,V=Ø |
| COM $LL,X | indexé | 63 | N,Z,C=1,V=Ø |
| COM $HHLL | étendu | 73 | N,Z,C=1,V=Ø |

### 5.4.5. Les instructions de décalage logique

Il s'agit des instructions suivantes :

— les instructions de décalage à gauche : LSL, LSLA, LSLB, LSLD,

— les instructions de décalage à droite : LSR, LSRA, LSRB, LSRD.

#### 5.4.5.1. Fonctionnement des instructions de décalage logique

*a) LSL*

Chaque bit est décalé d'un rang vers la gauche. Le bit 7 se trouve propagé dans le bit C du registre CCR tandis que le bit Ø est remplacé par un Ø. Il est à remarquer que cette instruction est en tous points similaire à l'instruction ASL décrite dans le paragraphe concernant les instructions arithmétiques.

En effet, un décalage logique vers la gauche conserve le bit de signe par l'intermédiaire du bit C.

*b) LSR*

Ici le décalage se fait d'un rang vers la droite. Le bit Ø se trouve propagé dans le bit C tandis que le bit 7 est remplacé par un zéro. Contrairement au cas de l'instruction ASR, le bit de signe n'est pas conservé d'où le nom de décalage logique.

*Exemple* : Soit l'instruction LSRA (adressage implicite).

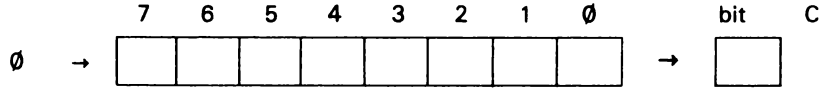
On suppose que A = $9D initialement.

$9D = 10011101

Après décalage cela nous donne :

$4E = 01001110 et C=1

D'une manière générale, nous avons donc le schéma suivant :



### 5.4.5.2. Les instructions LSL, LSLA, LSLB, LSLD

Le décalage s'effectue soit sur l'un des accumulateurs A et B soit sur le contenu d'une case-mémoire d'adresse spécifiée, soit sur la réunion des registres A et B (registre D).

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| LSLA | implicite | 48 | N,Z,V,C |
| LSLB | implicite | 58 | N,Z,V,C |
| LSLD | implicite | Ø5 | N,Z,V,C |
| LSL $LL,X | indexé | 68 | N,Z,V,C |
| LSL $HHLL | étendu | 78 | N,Z,V,C |

### 5.4.5.3. Les instructions LSR, LSRA, LSRB, LSRD

Le décalage s'effectue soit sur l'un des accumulateurs A et B ou sur le contenu d'une case-mémoire d'adresse spécifiée, soit sur le registre D.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| LSRA | implicite | 44 | Z,V,C,N=Ø |
| LSRB | implicite | 54 | Z,V,C,N=Ø |
| LSRD | implicite | Ø4 | Z,V,C,N=Ø |
| LSR $LL,X | indexé | 64 | Z,V,C,N=Ø |
| LSR $HHLL | étendu | 74 | Z,V,C,N=Ø |

## 5.4.6. Les instructions de rotation

Il s'agit des instructions suivantes:

— les instructions de rotation vers la gauche: ROLA, ROLB, ROL

— les instructions de rotation vers la droite: RORA, RORB, ROR

### 5.4.6.1. Fonctionnement des instructions de rotation

*a) ROL*

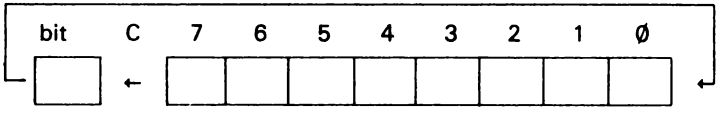
Chaque bit est décalé d'un rang vers la gauche. Le bit C devient le bit Ø tandis que le bit 7 se trouve propagé dans la retenue. Il y a donc rotation suivante:

C→Ø→1→2→3→4→5→6→7→C

*Exemple*: Soit l'instruction ROLA avec A=$9D et C=Ø initialement.

```
$9D = 10011101 qui donne apres rotation
$3A = 00111010 et C=1
```

Plus généralement nous avons donc le schéma suivant:



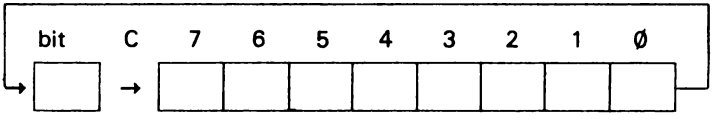

*b) ROR*

Le fonctionnement est identique à celui de l'instruction ROL sauf que la rotation se fait maintenant vers la droite.

*Exemple*: Soit l'instruction RORA avec A=$9D et C=1. Cela nous donne alors:

```
$CE = 11001110 et C=1
```

Plus généralement nous avons le schéma suivant:




### 5.4.6.2. Les instructons ROL, ROLA et ROLB

La rotation s'effectue soit sur l'un des accumulateurs A et B ou sur le contenu d'une case-mémoire d'adresse spécifiée.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| ROLA | implicite | 49 | N,Z,V,C |
| ROLB | implicite | 59 | N,Z,V,C |
| ROL $LL,X | indexé | 69 | N,Z,V,C |
| ROL $HHLL | étendu | 79 | N,Z,V,C |

#### 5.4.6.3. Les instructions ROR, RORA et RORB

La rotation s'effectue soit sur l'un des accumulateurs A et B soit sur le contenu d'une case-mémoire d'adresse spécifiée.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| RORA | implicite | 46 | N,Z,V,C |
| RORB | implicite | 56 | N,Z,V,C |
| ROR $LL,X | indexé | 66 | N,Z,V,C |
| ROR $HHLL | étendu | 76 | N,Z,V,C |

### 5.4.7. Les instructions de test de bits

Il s'agit des instructions suivantes :

— les instructions BITA et BITB,

— les instructions TST, TSTA et TSTB

#### 5.4.7.1. Fonctionnement des instructions "BIT"

Le 68Ø3 effectue un "ET" logique entre l'accumulateur et le contenu de la case-mémoire spécifiée (ou la valeur binaire dans le cas d'un adressage immédiat).

Nous allons envisager en détail la manière dont sont affectés les bits N et Z du registre de condition CC.

*a) Le bit Z*

Le bit Z est positionné à 1 ou Ø de la même façon que dans les instructions que nous avons rencontrées jusqu'à présent.

Si A est l'accumulateur et M la case-mémoire désignée par l'opérande on a :

$Z = 1$ si $A \wedge M = \emptyset$
$Z = \emptyset$ si $A \wedge M = \emptyset$

*b) Le bit N*

Le bit N est positionné à 1 ou Ø selon que le résultat de l'instruction est positif ou négatif.

#### 5.4.7.2. Les instructions BITA et BITB

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| BITA #$LL | immédiat | 85 | N,Z,V=Ø |
| BITA <$LL | direct | 95 | N,Z,V=Ø |
| BITA $LL,X | indexé | A5 | N,Z,V=Ø |
| BITA $HHLL | étendu | B5 | N,Z,V=Ø |
| BITB #$LL | immédiat | C5 | N,Z,V=Ø |
| BITB <$LL | direct | D5 | N,Z,V=Ø |
| BITB $LL,X | indexé | E5 | N,Z,V=Ø |
| BITB $HHLL | étendu | F5 | N,Z,V=Ø |

### 5.4.8. Les instructions TST, TSTA et TSTB

Ces instructions permettent tout simplement de tester la valeur contenue soit dans un des deux accumulateurs soit dans une case-mémoire donnée.

Les indicateurs N et Z sont positionnés suivant la valeur testée. Les bits V et C sont mis à zéro.

*Exemple*: Soit l'instruction TSTA avec A=$B8=1Ø1Ø1ØØØ.

L'accumulateur n'est pas modifié après une telle instruction mais le bit Z est positionné à Ø ($B8 est différent de zéro) et le bit N est positionné à 1 ($B8 est négatif puisque supérieur à 128).

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| TSTA | implicite | 4D | N,Z,V=C=Ø |
| TSTB | implicite | 5D | N,Z,V=C=Ø |
| TST $LL,X | indexé | 6D | N,Z,V=C=Ø |
| TST $HHLL | étendu | 7D | N,Z,V=C=Ø |

Nous allons maintenant donner un exemple de programme utilisant certaines des instructions de type logique rencontrées dans ce paragraphe et permettant de convertir deux chiffres BCD en la valeur binaire correspondante.

```
        ORG     $4000
        EXC     PROG
        ;PROG. CONVERSION BCD->BINAIRE
PROG    LDAA    CHIF    ;CHARGE NOMBRE BCD
        ANDA    #$0F    ;CHIFFRE POIDS FAIBLE
        STAA    VAR1    ;SAUVEGARDE TEMP.
        LDAA    CHIF
        ANDA    #$F0    ;CHIFFRE POIDS FORT
        LSRA            ;A=8 FOIS MSB
        STAA    VAR2    ;SAUVEGARDE TEMP.
        LSRA            ;A=4 FOIS MSB
        LSRA            ;A=2 FOIS MSB
        ADDA    VAR2    ;A=10 FOIS MSB
        ADDA    VAR1    ;RESULTAT
        STAA    RES
        RTS
CHIF    DFO     $00
VAR1    DFO     $00
VAR2    DFO     $00
RES     DFO     $00
```

Si nous prenons par exemple:

23 = 00100011 en BCD

alors il nous faut calculer 23=2×10+3.

Le problème est donc d'obtenir le "2", le "3" et de multiplier ensuite 2 par 10. Or on sait que 10=8+2 ce qui permet d'obtenir facilement le résultat escompté à l'aide de décalages adéquats.

Le nombre BCD est stocké en mémoire sous le nom CHIF.

Les variables temporaires VAR1 et VAR2 permettent de stocker respectivement le chiffre de poids faible (ici 3) et 8 fois le chiffre de poids fort. Le résultat final est stocké sous le nom VAR.

## 5.5. LES INSTRUCTIONS SUR LE REGISTRE D'ÉTAT

Nous regrouperons sous cette appellation les instructions suivantes :

- CLC, SEC
- CLI, SEI
- CLV, SEV

Les instructions CLC, CLI et CLV permettent respectivement de positionner à zéro les bits C, I et V du registre de condition CCR.

Au contraire, les instructions SEC, SEI et SEV permettent de positionner à un ces mêmes bits.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| CLC | implicite | ØC | C=Ø |
| SEC | implicite | ØD | C=1 |
| CLI | implicite | ØE | I=Ø |
| SEI | implicite | ØF | I=1 |
| CLV | implicite | ØA | V=Ø |
| SEV | implicite | ØB | V=1 |

Les instructions SEC et CLC ont été rencontrées précédemment avec les instructions arithmétiques.

Les instructions CLI et SEI permettent respectivement d'autoriser ou d'interdire les interruptions masquables de type IRQ. Nous reviendrons sur la notion d'interruption à la fin de ce chapitre.

## 5.6. LES INSTRUCTIONS DE COMPARAISON

Nous regrouperons sous cette appellation les instructions CMPA, CMPB, CBA, CPX.

Les trois premières opèrent sur les accumulateurs A et B et portent donc sur des mots de 8 bits. Au contraire, la dernière porte sur le registre X qui possède 16 bits.

Le fonctionnement de ces instructions est le même pour chacune d'entre elles. Afin d'effectuer une comparaison, le contenu de la case-mémoire spécifiée ou la valeur binaire (dans le cas d'un adressage immédiat) est retranché au registre considéré. Les indicateurs N, Z, V et C du registre de condition sont positionnés suivant le résultat de cette soustraction mais les contenus de la case-mémoire et des registres ne sont pas affectés.

On effectuera donc de façon interne l'opération suivante:

A−M pour l'instruction CMPA,

X−M,M+1 pour l'instruction CPX.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| CMPA ≠$LL | immédiat | 81 | N,Z,V,C |
| CMPA <$LL | direct | 91 | N,Z,V,C |
| CMPA $LL,X | indexé | A1 | N,Z,V,C |
| CMPA $HHLL | étendu | B1 | N,Z,V,C |
| CMPB ≠$LL | immédiat | C1 | N,Z,V,C |
| CMPB <$LL | direct | D1 | N,Z,V,C |
| CMPB $LL,X | indexé | E1 | N,Z,V,C |
| CMPB $HHLL | étendu | F1 | N,Z,V,C |
| CBA | implicite | 11 | N,Z,V,C |
| CPX ≠$LL | immédiat | 8C | N,Z,V,C |
| CPX <$LL | direct | 9C | N,Z,V,C |
| CPX $LL,X | indexé | AC | N,Z,V,C |
| CPX $HHLL | étendu | BC | N,Z,V,C |

Programme d'application: il s'agit d'un programme qui compare deux chaînes de caractères ASCII.

Dans notre cas, ces deux chaînes sont Alice 32 et Alice 32 et nous voulons vérifier qu'elles sont bien identiques.

Ce programme fonctionne très simplement: les caractères sont comparés un à un jusqu'au huitième.

Si les deux chaînes sont égales, la valeur 255 = $FF est stockée à l'adresse 16640 = $4100.

Au contraire, dans le cas de chaînes différentes, la valeur 0 est stockée à cette même adresse.

Si vous faites tourner ce programme, vous devez donc obtenir 255 en 16640. Vous pourrez vérifier par vous-même qu'en changeant la chaîne de caractères TEXT2 par Alice 90 le résultat sera égal à 0.

```
        ORG    $4000
        EXC    PROG
TEXT1  'ALICE 32
TEXT2  'ALICE 32
PROG    LDAB   #$00   ;IND. CHAINE DIFF.
        LDX    #$4000
SUIT    LDAA   $00,X  ; 1 ER CARACTERE CHAINE 1
        CMPA   $08,X  ;COMPARE 1 ER CARACTERE CHAINE
        BNE    FIN    ;DIFFERENT? TERMINE
        INX           ; NON, RECOMMENCE
        CPX    #$4008 ;DERNIER CARACTERE?
        BCS    SUIT   ;NON, CARACTERE SUIVANT
        LDAB   #$FF   ;CHAINES EGALES
FIN     STAB   $4100
        RTS
```

## 5.7. LES INSTRUCTIONS DE BRANCHEMENT

Il existe deux grandes catégories d'instructions de branchement:

— les instructions de branchement inconditionnel: BRA, BRN, BSR et JMP;

— les instructions de branchement conditionnel: (nous ne les listerons pas ici étant donné leur nombre important).

### 5.7.1. Les instructions de branchement inconditionnel

La principale instruction de ce type (et la plus connue) est l'instruction JMP qui est l'équivalent du GOTO en Basic (du moins en ce qui concerne les modes d'adressage direct et étendu), sauf qu'en assembleur l'étiquette a un nom (par exemple "BOUCLE"). Lorsque le microprocesseur rencontre le code-opération de l'instruction JMP, il charge son compteur ordinal avec l'adresse spécifiée dans l'opérande.

On a donc pour l'instruction JMP :

EA→PC

EA désignant une adresse effective sur 16 bits.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| JMP $LL,X | indexé | 6E | aucun |
| JMPA $HHLL | étendu | 7E | aucun |

Nous avons inclus sous cette rubrique les instructions BRA, BRN, BSR.

Ces instructions sont ce que l'on appelle des instructions de branchement relatif. A ce titre, l'adresse de branchement est calculée en ajoutant au registre PC un déplacement signé (valeur en complément à deux). Ce déplacement est codé sur 8 bits (pour accéder à un espace adressable de 256 octets).

L'instruction BRA signifie "branchement dans tous les cas" (branch always). Elle correspond à l'instruction JMP avec un adressage relatif.

On a donc :

PC + Déplacement→PC

L'instruction BRN signifie "jamais de branchement". Elle est l'équivalent de l'instruction NOP puisqu'elle n'effectue aucune action. Elle est utilisée uniquement pour disposer de programmes plus facilement lisibles.

L'instruction BSR signifie "branchement à un sous-programme".

Elle est donc l'équivalent de l'instruction JSR qui sera décrite plus loin, mais cette fois-ci avec un adressage relatif.

On a donc pour cette dernière instruction :

— pousse PC bas sur la pile ; SP−1 → SP

— pousse PC haut sur la pile ; SP−1 → SP

— PC + déplacement → PC

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| BRA $LL | relatif | 2Ø | aucun |
| BRN $LL | relatif | 21 | aucun |
| BSR $LL | relatif | 8D | aucun |

### 5.7.2. Les instructions de branchement conditionnel

Comme leur nom l'indique, ces instructions ont un mode d'exécution qui dépend d'une condition. En l'occurrence ici, c'est le contenu d'un bit (ou de plusieurs bits) du registre de condition qui importe. Toutes ces instructions ont un fonctionnement identique c'est pourquoi nous n'en expliciterons qu'une seule.

Considérons par exemple l'instruction BEQ. Elle signifie "branch if equal" (branchement si égalité à zéro). Il y aura donc branchement relatif si le bit Z du registre de condition est égal à 1 (caractéristique d'un résultat nul).

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| BCC $LL | relatif | 24 | aucun, condition C=Ø |
| BCS $LL | relatif | 25 | aucun, condition C=1 |
| BEQ $LL | relatif | 27 | aucun, condition Z=1 |
| BGE $LL | relatif | 2C | aucun, condition >=Ø |
| BGT $LL | relatif | 2E | aucun, condition >Ø |
| BHI $LL | relatif | 22 | aucun, condition supérieur à |
| BHS $LL | relatif | 24 | aucun, condition supérieur ou égal à |
| BLE $LL | relatif | 2F | aucun, condition <=Ø |
| BLO $LL | relatif | 25 | aucun, condition inférieur à |

| *nstruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| LS $LL | relatif | 23 | aucun, condition inférieur ou égal à |
| LT $LL | relatif | 2D | aucun, condition <Ø |
| MI | relatif | 2B | aucun, condition N=1 |
| NE | relatif | 26 | aucun, condition Z=Ø |
| PL | relatif | 2A | aucun, condition N=Ø |
| VC | relatif | 28 | aucun, condition V=Ø |
| VS | relatif | 29 | aucun, condition V=1 |

Comme vous pouvez le constater le nombre de ces instructions est ‹trêmement important. De plus certaines d'entre elles semblent avoir es modes de fonctionnement similaires. C'est pourquoi nous allons ssayer de clarifier un peu tout cela dans les lignes qui suivent.

Pour cela nous allons regrouper ces instructions de branchement en ois catégories :

*a)* celles qui traduisent une condition simple (par exemple dépen- ant de la valeur d'un bit du registre de condition).

*b)* celles qui opèrent sur des nombres signés.

*c)* celles qui opèrent sur des nombres non signés.

Notons que certaines instructions peuvent appartenir à plusieurs roupes simultanément. En effet, l'égalité par exemple concerne les ombres signés ou non signés de manière similaire.

Nous allons examiner en détail les instructions appartenant à hacun de ces trois groupes.

*a)* Il s'agit des instructions BEQ, BNE, BMI, BPL, BCS, BCC, BVS et BVC.

— les instructions BEQ (branchement si égalité) et BNE (branche- nent si non égalité) testent la valeur du bit Z et donc l'égalité ou non à éro.
Donc si Z=1 l'égalité à zéro est vérifiée.

— les instructions BMI (branchement si moins ou N=1) et BPL branchement si plus ou N=Ø) testent la valeur du bit N et donc le signe lu nombre considéré.

— les instructions BCC (branchement si C=Ø et BCS (branche- nent si C=1) testent la valeur du bit C.

— les instructions BVC (branchement si V=Ø) et BVS (branchement si V=1) permettent de tester la valeur du bit de dépassement V.

En théorie il est possible d'effectuer n'importe quel type de test à l'aide de ces instructions. Cependant leur mode d'application peut différer selon que l'on s'intéresse à des nombres signés ou non. C'est pourquoi il existe dans le 68Ø3 un certain nombre d'instructions qui facilitent le traitement de nombres de ces deux types, en utilisant par exemple certaines combinaisons de bits du registre de condition. Bien entendu ces combinaisons sont traitées de façon interne par le 68Ø3 et l'utilisateur n'a à se soucier que du résultat.

*b)* Les instructions portant sur des nombres signés sont les suivantes: BGT, BLE, BLT, BEQ, BNE.

— l'instruction BGT (branchement si supérieur à zéro) teste un inégalité stricte. En effet, la condition détectée est:

Z V (N XOR V)=Ø (XOR désignant le "OU exclusif")

Le test sur Z donne l'inégalité stricte. En effet, si Z=1 le résulat est nul.

Le test (N XOR V) permet de détecter si les deux bits N et V sont égaux.

Généralement les instructions de branchement relatif portant sur des nombres signés ou non ont lieu après une instruction de comparaison dans laquelle un nombre (B) est retranché d'un nombre (A). Certains bits du registre de condition sont alors testés par l'instruction de branchement relatif considérée.

*Exemple*:

Soit les deux nombres A=$37 e B=$A8. Nous voulons tester si A est strictement inférieur à B.

On a:

```
        $37 = 00110111
        $A8 = 10101000
       $-A8 = 01011000
-------------------------
 $37-  $A8 = 10001111
```

Nous avons alors Z=Ø, N=1 et V=1 ce qui donne Z V (N XOR V) = Ø.

Puisque nous étions en présence de nombres signés, A était positif et B négatif (donc A était supérieur à B).

Vous pourrez vérifier par vous-même à l'aide d'autres exemples que la condition Z V (N XOR V) = Ø est bien correcte pour A>B.

Le contraire de l'instruction BGT est l'instruction BLE qui teste donc l'inégalité large A<=B.

La condition nécessaire est donc Z V (N XOR V) = 1.

— l'instruction BGE fonctionne comme l'instruction BGT sauf qu'ici l'inégalité au sens large est testée (A>=B) et la condition qui doit être vérifiée est la suivante:

```
(N XOR V) =Ø
```

Le contraire de cette instruction est BLT qui teste l'inégalité stricte A<B et pour laquelle la condition nécessaire est:

```
(N XOR V)=1
```

— les instructions BEQ et BNE ont déjà été rencontrées ci-dessus.

*c)* Les instructions portant sur des nombres non signés sont les suivantes: BHI, BLS, BHS, BLO, BEQ, BNE.

Comme dans le cas de nombres signés il y a possibilité de détecter ici une égalité (instructions BHI et BLO) ou une inégalité au sens large (instructions BHS et BLS).

L'instruction BHI teste la condition (C V Z) = Ø qui est la condition nécessaire pour qu'un nombre non signé (A) soit supérieur à un autre (B).

Au contraire l'instruction BLS teste la condition opposée (A<=B) soit (C V Z) = 1.

L'instruction BHS teste si C=Ø qui est une condition suffisante pour tester une inégalité au sens large entre deux nombres non signés.

## 5.8. LES INSTRUCTIONS D'APPEL ET DE RETOUR DE SOUS-PROGRAMME

Il s'agit des instructions JSR et RTS.

L'instruction JSR est l'équivalent du GOSUB en Basic et est donc une instruction d'appel de sous-programme.

L'instruction RTS est l'équivalent du RETURN en Basic et est donc une instruction de retour de sous-programme.

Lorsque le microprocesseur rencontre l'instruction JSR, il charge le contenu du compteur ordinal dans la pile puis se branche à l'adresse spécifiée par l'opérande.

On a donc les opérations suivantes :

charge PC bas sur la pile ; SP–1→SP
charge PC haut sur la pile ; SP–1→SP
EA→PC

Lors d'une instruction JSR le pointeur de pile est donc décrémenté de deux unités.

Lorsque le microprocesseur rencontre l'instruction RTS, il va chercher l'adresse qui se trouve en haut de la pile et la charge ensuite dans le compteur ordinal.

On a donc les opérations suivantes :

SP+1→SP ; charge PC haut avec le sommet de la pile ;
SP+1→SP ; charge PC bas avec le sommet de la pile ;

Lors d'une instruction RTS, le pointeur de pile est donc incrémenté de deux unités ;

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| JSR <$LL | direct | 9D | aucun |
| JSR $LL,X | indexé | AD | aucun |
| JSR $HHLL | étendu | BD | aucun |
| RTS | implicite | 39 | aucun |

Nous allons illustrer le fonctionnement des instructions d'appel et de retour de sous-programme par un retour sur le programme de génération de son donné dans le chapitre consacré à l'"Introduction au 68Ø3".

Nous donnons ci-dessous le listing d'un programme permettant de réaliser la même fonction mais utilisant, pour la génération du son proprement dit, un sous-programme.

```
        ORG    $4ØØØ
        EXC    PROG
        ;PROGRAMME DE GENERATION DE SON
PROG    LDAB   #$FF   ;PARAMETRE DE MODULATION
        JSR    SON    ;GENERE SON
        RTS
SON     LDAA   #$ØØ   ;MISE A ZERO DE VAL
        STAA   VAL
INV     LDAA   VAL    ;INVERSION DE VAL
        EORA   #$FF
        STAA   VAL
        STAA   $BFFF  ;ENVOIE DANS GENERATEUR
        TBA           ;B MIS DANS A
LOOP    DECA          ;BOUCLE D'ATTENTE TERMINEE?
        BNE    LOOP   ;NON,CONTINUE
        DECB          ;B DIMINUE D'UNE UNITE
        BNE    INV    ;SI ()Ø,RECOMMENCE
        RTS
VAL     DFO    $ØØ
```

## 5.9. LES INSTRUCTIONS SUR LA PILE

Elles sont au nombre de six et comprennent les instructions d'empilement et de dépilement.

— empilement PSHA, PSHB et PSHX

— dépilement PULA, PULB et PULX

### 5.9.1. Fonctionnement des instructions d'empilement

Ces instructions permettent de sauvegarder le contenu d'un registre interne du 68Ø3 au sommet de la pile.

Grâce à ces instructions les registres A, B ou X peuvent être stockés sur la pile.

Donc les instructions PSHA, PSHB et PSHX permettent respectivement de charger le contenu des registres A, B ou X sur le sommet de la pile.

### 5.9.2. Fonctionnement des instructions de dépilement

Ces instructions permettent de charger un registre interne du 68Ø3 à l'aide d'octets situés dans la pile.

Grâce à ces instructions les registres A, B ou X peuvent être chargés à partir de la pile.

### 5.9.3. Les instructions PSHA, PSHB, PSHX, PULA, PULB, PULX

Les opérations suivantes sont effectuées lors de l'exécution des instructions d'empilement :

pousse A sur le sommet de la pile, SP–1→SP
(cas de l'instruction PSHA)

pousse X bas sur le sommet de la pile, SP–1→SP
pousse X haut sur le sommet de la pile, SP–1→SP
(cas de l'instruction PSHX)

Dans le cas des opérations de dépilement on aura :

SP+1→SP, charge A avec le sommet de la pile
(cas de l'instruction PULA)

SP+1→SP, charge X haut avec le sommet de la pile
SP+1→SP, charge X bas avec le sommet de la pile
(cas de l'instruction PULX)

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| PSHA | immédiat | 36 | aucun |
| PSHB | immédiat | 37 | aucun |
| PSHX | immédiat | 3C | aucun |
| PULA | immédiat | 32 | aucun |
| PULB | immédiat | 33 | aucun |
| PULX | immédiat | 38 | aucun |

*Exemple d'application*: Les instructions d'empilement ou de dépilement sont très utiles quand il s'agit de stocker des données temporairement.

Comme nous l'avons dit lors de la description du registre pointeur de pile, la première donnée rentrée est alors la dernière sortie.

Le programme suivant permet de "sauvegarder le contexte", c'est-à-dire le contenu de tous les registres internes du microprocesseur.

Ceci peut être utile lors de l'exécution d'une routine d'interruption (les interruptions seront décrites en fin de chapitre).

```
        ORG    $4000
        EXC    PROG
        ;ROUTINE DE SAUVEGARDE REGISTRES
PROG    PSHA   ;REGISTRE A
        PSHB   ;REGISTRE B
        PSHX   ;REGISTRE X
        TPA
        PSHA   ;REGISTRE CCR
        TSX
        DEX
        PSHX   ;REGISTRE SP
```

Au contraire la routine suivante permet de "restaurer le contexte", c'est-à-dire de recharger une ancienne configuration des registres internes du 6803.

```
        ORG    $4000
        EXC    PROG
        ;ROUTINE DE RESTAURATION REGISTRES
PROG    PULX
```

```
INX
TXS     ;REGISTRE SP
PULA
TAP     ;REGISTRE CCR
PULX    ;REGISTRE X
PULB    ;REGISTRE B
PULA    ;REGISTRE A
```

Les deux routines ci-dessus tiennent compte des particularités des instructions TSX et TXS décrites précédemment, ce qui explique la présence des instructions d'incrémentation et de décrémentation.

## 5.10. LES INSTRUCTIONS SPÉCIALES

Il s'agit des instructions suivantes :

— l'instruction NOP
— l'instruction RTI
— l'instruction SWI
— l'instruction WAI

Nous allons envisager chacune d'entre elles chacune à son tour.

### 5.10.1. L'instruction NOP

Le fonctionnement de cette instruction est très facile à comprendre puisqu'elle ne fait rien, ou presque : elle se contente juste d'incrémenter le compteur ordinal.

Mais quelle est l'utilité d'une instruction qui ne fait rien ? En fait elle peut servir à beaucoup de choses et ceux d'entre vous qui ont utilisé des calculatrices programmables doivent le savoir :

— remplacer une instruction non utile par un NOP permet de ne pas avoir à réécrire tout le programme lors d'un assemblage à la main ou pendant la mise au point. Sans cela, il faudrait recalculer tous les branchements.

Cet inconvénient est limité dans le cas de l'Alice puisque cette machine dispose d'un Éditeur-Assembleur incorporé.

— provoquer un délai de durée fixe dans l'exécution d'un programme.

— mettre au point un programme partie par partie en remplaçant, par exemple, certains sous-programmes par des NOP.

Il est évident bien sûr que le programme définitif doit être débarrassé de ces instructions inutiles afin d'en diminuer le temps d'exécution.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| NOP | implicite | Ø1 | aucun |

### 5.10.2. L'instruction RTI

En pratique, vous n'aurez probablement jamais à l'utiliser au même titre que l'instruction WAI.

Nous allons tout de même décrire brièvement ce qu'est une interruption.

Nous avons vu qu'il existait une instruction d'appel de sous-programme JSR. Cette instruction permet donc de faire un appel de sous-programme à partir du logiciel.

Par définition, une interruption est un appel de sous-programme provoqué par le matériel (par opposition au logiciel dans le cas de JSR). Il y a donc possibilité, à l'aide d'un signal externe, de se brancher à un sous-programme spécialisé dont l'instruction de retour est RTI (retour d'interruption).

Il existe dans le 68Ø3 deux types d'interruptions matérielles.

— l'interruption NMI (de l'anglais "Non Maskable Interrupt" qui signifie "interruption non masquable).
Lorsqu'un signal actif est appliqué à la broche NMI du 68Ø3, celui-ci sauvegarde le contenu de tous ses registres internes et se branche automatiquement à une routine dont l'adresse de départ est donnée par le contenu des cases-mémoire d'adresses $FFFC et $FFFD.

Cette interruption ne peut être interdite d'où son nom.

— l'interruption masquable IRQ.

Contrairement à la précédente cette interruption peut ou non être autorisée. Ceci peut se faire grâce au bit I (bit 4) du registre de condition CC. Lorsque ce bit est positionné à 1, l'interruption IRQ est interdite. L'adresse de début de la routine de traitement de ce type d'interruptions est donnée par le contenu des adresses $FFF8 et $FFF9.

Les routines de traitement des interruptions se terminent toutes par une instruction RTI. Lorsque le 68Ø3 rencontre cette instruction, les registres internes sont chargés les uns après les autres dans l'ordre PC bas, PC haut, X bas, X haut, A, B, CCR.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| RTI | implicite | 3B | tous (registre CCR restauré) |

**Remarque**: l'accès aux interruptions nécessite d'accéder au "bus" de l'Alice situé à l'arrière de l'appareil et suppose donc la connaissance des différents signaux qui y véhiculent.

### 5.10.3. L'instruction SWI

Nous avons vu précédemment que certains signaux externes pouvaient provoquer une interruption et brancher le microprocesseur à un sous-programme spécialisé dont l'instruction de retour était RTI.

Il est possible de réaliser la même chose grâce à des instructions appelées interruptions logicielles. Lorsqu'un programme est exécuté et que le microprocesseur rencontre une de ces instructions, celui-ci sauvegarde l'état complet des registres internes dans la pile système suivant la procédure suivante:

Le bit du registre CCR est positionné à 1 afin d'interdire les interruptions matérielles de type IRQ.

| | | | | | | |
|---|---|---|---|---|---|---|
| X bas | → | (SP) | ; | SP–1 | → | SP |
| X haut | → | (SP) | ; | SP–1 | → | SP |
| A | → | (SP) | ; | SP–1 | → | SP |
| B | → | (SP) | ; | SP–1 | → | SP |
| CCR | → | (SP) | ; | SP–1 | → | SP |

(SP) désignant le contenu de la case-mémoire dont l'adresse est donnée par le registre SP.

Après la sauvegarde de ces registres, le PC est chargé avec le contenu des adresses $FFFA et $FFFB. L'adresse $FFFA contient l'octet de poids faible d'une adresse de branchement tandis que l'adresse $FFFB en contient l'octet de poids fort.

Selon la valeur du vecteur de branchement, il y aura appel du programme moniteur du système, du Basic, ou de tout autre programme. Cette instruction est très utilisée pour la mise au point des programmes car elle permet d'en arrêter le déroulement là où on le désire.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| SWI | implicite | 3F | I=1 |

### 5.10.4 L'instruction WAI

Cette instruction permet de synchroniser le 68Ø3 sur un événement extérieur. Ceci peut être utile par exemple dans le cas d'une application biprocesseur où les tâches sont partagées. L'un des deux microprocesseurs peut avoir à attendre que l'autre ait terminé l'exécution d'un programme donné avant de pouvoir continuer sa propre tâche.

Lorsque le 68Ø3 rencontre une instruction WAI, il s'arrête et attend qu'une interruption se produise. Lorsque celle-ci intervient, les registres internes du 68Ø3 sont sauvegardés sur la pile et il y a exécution de la routine d'interruption de type IRQ se terminant bien sûr par un RTI.

| *Instruction* | *Mode d'adressage* | *Code Opération* | *Indicateurs affectés* |
|---|---|---|---|
| WAI | implicite | 3E | aucun |

# 6

# Les sous-programmes système

Nous avons vu précédemment (chapitre 3) un aperçu de la configuration matérielle de l'Alice. Il est nécessaire maintenant de voir, sans rentrer bien sûr dans les détails, comment le système fonctionne du point de vue logiciel. Ce dernier est, nous l'avons dit, contenu dans une mémoire morte de 16 Koctets. Cette dernière contient l'interpréteur Basic (table des instructions et commandes, routines de calcul en virgule flottante, etc...), le système d'exploitation et bien sûr l'Éditeur-Assembleur qui est une des particularités les plus intéressantes de l'Alice.

Mais revenons sur le système d'exploitation. Les tâches accomplies par ce dernier sont très diverses.

— gestion de l'écran par l'intermédiaire du circuit contrôleur de type 9345,

— gestion des Entrées-Sorties (clavier, interface cassette, imprimante),

— éditeur de textes pour l'entrée des programmes,

— initialisation du système à la mise sous tension.

Afin de vous faire sentir comment interagissent le système d'exploitation et le Basic, prenons un exemple. Soit le programme suivant :

```
10 INPUT A
20 PRINT A
```

Lorsque l'interpréteur Basic rencontre le code (sur un octet) relatif à l'instruction INPUT, il se branche à une routine dont le rôle est d'attendre qu'un caractère soit rentré au clavier.

De même, lorsque l'interpréteur Basic rencontre le code relatif à l'instruction PRINT, il se branche à une routine chargée d'afficher A sous la forme d'une chaîne de caractères sur l'écran.

Les diverses routines d'Entrées-Sorties ainsi que bien d'autres sont accessibles à l'utilisateur. Elles sont appelées comme de "vulgaires" sous-programmes.

Par exemple la routine qui permet d'afficher un caractère sur l'écran est appelée à l'adresse hexadécimale $D40C.

L'utilisation des sous-programmes de la ROM moniteur se fait en un certain nombre d'étapes qui sont en général les suivantes :

— on charge les registres de communication (A, B, X) avec les paramètres à passer à la routine considérée ;

— on appelle la routine proprement dite par une instruction JSR ;

— on gère éventuellement les données retournées par cette routine (par exemple code ASCII d'un caractère frappé au clavier).

Nous allons maintenant décrire les différentes routines mises à la disposition du programmeur en assembleur.

### 1) Initialisation de l'affichage en 40 colonnes

Cette routine, située à l'adresse $D400 permet d'initialiser le contrôleur d'écran de manière à ce qu'il fonctionne en mode 40 colonnes, avec écran noir.

L'utilisateur de cette routine pourra se faire aisément à l'aide de la routine assembleur suivante (l'adresse de départ est ici $5800 ce qui convient dans le cas d'un Alice 90. Les possesseurs d'Alice 32 pourront

assembler le programme à partir de l'adresse $4000 par exemple).

```
1                          ORG    $5800
2                          EXC    INIT
3 5800 BDD400 INIT         JSR    $D400
```

Notons que le contenu des registres A, B et X du 6803 est affecté par cette routine.

**2) Affichage d'un caractère semi-graphique sur l'écran**

Cette routine permet d'afficher un caractère graphique sur la totalité de l'écran. Elle est située à l'adresse $D406.

La routine assembleur suivante permet d'afficher des pavés jaunes sur l'écran :

```
1                          ORG    $5800
2                          EXC    INIT
3 5800 C691     INIT       LDAB   #$91
4 5802 BDD406              JSR    $D406
5 5805 7E5800              JMP    INIT
```

Le caractère à afficher est déterminé par l'intermédiaire de l'Accumulateur B ; celui-ci est donc chargé par le code semi-graphique désiré avant l'appel de la routine.

Le manuel d'utilisation de l'Alice donne la manière de calculer la valeur des codes semi-graphiques en fonction de la forme et de la couleur désirée (p. 171 du manuel de l'Alice 90).

Dans notre cas :

$91 (hexadécimal) = 145 = 129+16

Ceci nous donne bien le caractère suivant de couleur jaune.

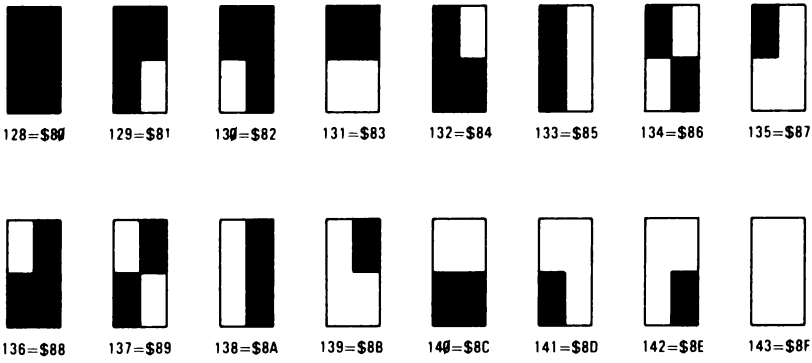
La petite routine assembleur suivante permet de visualiser successivement les différents caractères semi-graphiques dans les différentes couleurs disponibles.

```
ASSEMBLEUR ALICE REV 1.00*
COPYRIGHT MATRA, 1984

   1                        ORG    $5800
   2                        EXC    INIT
   3 5800 C680    INIT      LDAB   #$80     ;PREMIER CODE
   4 5802 F75818            STAB   CODE     ;SAUVEGARDE
   5 5805 BDD406  AFF       JSR    $D406    ;AFFICHE CE CODE
   6 5808 CEFFFF            LDX    #$FFFF   ;PROVOQUE UN DELAI
   7 580B 09      LOOP      DEX             ;DELAI TERMINE ?
   8 580C 26FD              BNE    LOOP     ;NON, ATTEND
   9 580E F65818            LDAB   CODE     ;OUI, PROCHAIN CODE
  10 5811 5C                INCB
  11 5812 F75818            STAB   CODE
  12 5815 7E5805            JMP    AFF      ;RECOMMENCE PROCEDURE
  13 5818 00      CODE      FCB    $00
  14
  15
  16
  17

0 ERREUR(S) PASSE 1
0 ERREUR(S) PASSE 2
SYMBOLES :
--------
INIT =5800 AFF  =5805 LOOP =580B CODE =5818
```

Le premier code à afficher est le code **$80** qui correspond à un rectangle noir. Chaque fois que cette routine est exécutée, ce code est incrémenté. Les 16 caractères semi-graphiques sont donc passés en revue chacun à son tour, ceci pour les couleurs suivantes :

| | |
|---|---|
| jaune | (nombre associé = 16) |
| bleu-roi | (nombre associé = 32) |
| rouge | (nombre associé = 48) |
| ivoire | (nombre associé = 64) |
| bleu pâle | (nombre associé = 80) |
| mauve | (nombre associé = 96) |
| orange | (nombre associé = 112) |

Nous rappelons en effet ci-dessous la manière dont sont obtenus les différents codes semi-graphiques.




Le code semi-graphique résultant est obtenu en faisant la somme du code ci-dessus avec le nombre associé à la couleur considérée.

C'est ainsi que pour un rectangle rouge, le code sera égal à :

143 + 48 = 191 = **$BF**

Les registres A, B et X sont affectés pour l'appel de cette routine.

### 3) Déplacement de l'écran vers le haut

La routine d'adresse **$D409** permet de déplacer ce qui est affiché sur l'écran d'une rangée vers le haut chaque fois qu'elle est appelée. (Il s'agit donc de ce que l'on appelle en anglais un scrolling).

La petite routine assembleur ci-dessous illustre très simplement son fonctionnement :

```
1                       ORG   $5800
2                       EXC   INIT
3  5800 BDD409  INIT    JSR   $D409   ;DECALAGE D'UN RANG
4  5803 CE2000          LDX   #$2000  ;DELAI
5  5806 09      LOOP    DEX           ;DELAI TERMINE ?
6  5807 26FD            BNE   LOOP    ;NON, ATTEND
7  5809 7E5800          JMP   INIT    ;OUI, RECOMMENCE
```

Nota : Seul le registre X est modifié par l'appel de cette routine.

**4) Écriture d'un caractère sur l'écran**

Cette routine, d'adresse $D4ØC, permet d'afficher un caractère à la position désirée sur l'écran. Le caractère en question est représenté par son code ASCII.

Le fonctionnement de cette routine est le suivant:

— le code ASCII du caractère à visualiser doit être chargé dans l'Accumulateur A.

— le registre d'index X doit être chargé avec un mot de 16 bits dont les 8 bits de poids fort représentent le numéro de la rangée où sera affiché le caractère et dont les 8 bits de poids faible représentent le numéro de la colonne où ce caractère sera affiché.

*Exemple*: La routine assembleur suivante permet d'afficher le mot "Alice" au milieu de l'écran (rangée n° 1Ø, colonne n° 16 à 21).

```
ASSEMBLEUR ALICE REV 1.ØØ*
COPYRIGHT MATRA,1984

 1                      ORG    $58ØØ
 2                      EXC    INIT
 3 58ØØ 8641    INIT    LDAA   #$41    ; LETTRE A
 4 58Ø2 CEØA1Ø          LDX    #$ØA1Ø  ; LIGNE 1Ø,COLONNE 16
 5 58Ø5 BDD4ØC          JSR    $D4ØC
 6 58Ø8 864C            LDAA   #$4C    ; LETTRE L
 7 58ØA CEØA11          LDX    #$ØA11  ; COLONNE SUIVANTE
 8 58ØD BDD4ØC          JSR    $D4ØC
 9 5810 8649            LDAA   #$49    ; LETTRE I
1Ø 5812 CEØA12          LDX    #$ØA12
11 5815 BDD4ØC          JSR    $D4ØC
12 5818 8643            LDAA   #$43    ; LETTRE C
13 581A CEØA13          LDX    #$ØA13
14 581D BDD4ØC          JSR    $D4ØC
15 5820 8645            LDAA   #$45    ; LETTRE E
16 5822 CEØA14          LDX    #$ØA14
17 5825 BDD4ØC          JSR    $D4ØC
18 5828 7E58ØØ          JMP    INIT
19
2Ø

Ø ERREUR(S) PASSE 1
Ø ERREUR(S) PASSE 2
SYMBOLES :
--------
INIT =58ØØ
```

Notons que cette routine (adresse $D4ØC) modifie le contenu des registres A et B.

**5) Lecture d'un caractère sur l'écran**

A l'inverse de la routine précédente qui permettait d'afficher un caractère sur l'écran, la routine considérée ici, d'adresse $D4ØF, permet de lire le caractère affiché à une position donnée de l'écran. L'exemple ci-dessous illustre le fonctionnement de cette routine.

Après avoir affiché le mot Alice comme précédemment, nous voulons le recopier une ligne en dessous. Le programme assembleur réalisant cette opération sera le suivant:

Avant le "JMP INIT" du programme précédent, on insère les lignes suivantes:

```
 3 5800 CE0A10  LDX   #$0A10 ;LIT PREMIERE LETTRE
 4 5803 BDD40F  JSR   $D40F
 5 5806 CE0B10  LDX   #$0B10 ;RECOPIE LIGNE 11
 6 5809 BDD40C  JSR   $D40C
 7 580C CE0A11  LDX   #$0A11 ;LETTRE SUIVANTE
 8 580F BDD40F  JSR   $D40F
 9 5812 CE0B11  LDX   #$0B11
10 5815 BDD40C  JSR   $D40C
11 5818 CE0A12  LDX   #$0A12 ;LETTRE SUIVANTE
12 581B BDD40F  JSR   $D40F
13 581E CE0B12  LDX   #$0B12
14 5821 BDD40C  JSR   $D40C
15 5824 CE0A13  LDX   #$0A13 ;LETTRE SUIVANTE
16 5827 BDD40F  JSR   $D40F
17 582A CE0B13  LDX   #$0B13
18 582D BDD40C  JSR   $D40C
19 5830 CE0A14  LDX   #$0A14 ;LETTRE SUIVANTE
20 5833 BDD40F  JSR   $D40F
21 5836 CE0B14  LDX   #$0B14
22 5839 BDD40C  JSR   $D40C
```

Le fonctionnement de cette routine est donc le suivant: le registre d'index X est chargé avec la position désirée sur l'écran, comme décrit

précédemment (octet de poids fort = n° de rangée, octet de poids faible = n° de colonne).

Après l'appel de la routine d'adresse $D4ØF, le registre A est chargé avec le code ASCII du caractère occupant la position spécifiée sur l'écran.

Notons que le contenu du registre B est modifié lors de l'appel de cette routine.

### 6) Affichage d'un même caractère sur une ligne de l'écran

Comme son nom l'indique, cette routine permet d'afficher un caractère quelconque sur la totalité d'une ligne de l'écran.

Ce caractère devra préalablement avoir été stocké dans les registres internes du 9345.

Ceci peut se réaliser aisément conjointement avec la routine d'adresse $D4ØC, vue précédemment, et qui permet l'affichage d'un caractère sur l'écran.

Le petit programme suivant illustre le fonctionnement de ces deux routines et permet l'affichage du caractère semi-graphique de code $91 sur la ligne 16.

```
1                              ORG    $5800
2                              EXC    INIT
3  5800 8691        INIT       LDAA   #$91
4  5802 CE1000                 LDX    #$1000
5  5805 BDD40C                 JSR    $D40C
6  5808 BDD41B                 JSR    $D41B
7  580B 7E5800                 JMP    INIT
```

Notons que cette routine modifie le contenu des accumulateurs A et B.

### 7) Effacement d'une ligne de caractères sur l'écran

Tout comme nous avons vu qu'il existait une routine capable d'afficher un caractère identique sur une ligne de l'écran, il existe une routine permettant d'effacer une ligne. Elle réside à l'adresse $D429.

Le registre X doit être chargé avec le numéro de la rangée (octet de poids fort) avant l'appel de cette routine.

Le petit programme assembleur suivant illustre son fonctionnement, il permet d'allumer et d'éteindre successivement une rangée de caractères semi-graphiques jaunes à la ligne 16 de l'écran.

```
 1                        ORG   $5800
 2                        EXC   INIT
 3 5800 8691    INIT      LDAA  #$91    ;AFFICHE CARACTERE
 4 5802 CE1000            LDX   #$1000
 5 5805 BDD40C            JSR   $D40C   ; UNE FOIS
 6 5808 BDD41B            JSR   $D41B   ; SUR TOUTE LA LIGNE
 7 580B CEFFFF            LDX   #$FFFF  ; DELAI
 8 580E 09      LOOP1     DEX
 9 580F 26FD              BNE   LOOP1
10 5811 CE1000            LDX   #$1000  ; ETEINT CETTE LIGNE
11 5814 BDD429            JSR   $D429
12 5817 CEFFFF            LDX   #$FFFF  ; DELAI
13 581A 09      LOOP2     DEX
14 581B 26FD              BNE   LOOP2
15 581D 7E5800            JMP   INIT    ; RECOMMENCE
```

Notons que les accumulateurs A et B sont affectés par l'appel de cette routine.

### 8) Initialisation de l'écran

Une routine d'adresse $D42C permet d'initialiser le contrôleur d'écran pour fonctionner, selon le cas, en mode 32, 40 ou 80 colonnes.

Le fonctionnement de cette routine est le suivant :

On charge tout d'abord la case-mémoire d'adresse $301A avec la valeur 0, 1 ou 2.

On a respectivement :

0 → initialisation sur 80 colonnes
1 → initialisation sur 40colonnes
2 → initialisation sur 32 colonnes

Après avoir chargé cette case-mémoire avec la valeur désirée, la routine concernée ici est appelée pour une instruction JSR.

Le contenu des registres A, B et X est modifié par l'appel de cette routine.

*Exemple*: Pour un affichage sur 40 colonnes, on écrira:

```
1                       ORG    $5800
2                       EXC    INIT
3 5800 8601    INIT     LDAA   #$01
4 5802 BDD42C           JSR    $D42C
5 5805 7E5805  LOOP     JMP    LOOP
```

La ligne 5 de ce programme permet juste de visualiser le fonctionnement de cette routine et n'apporte rien au programme en lui-même.

### 9) Initialisation du système

La routine d'adresse $D42F permet de réinitialiser l'Alice, ceci sans modifier le contenu de la RAM utilisateur. Il y a alors retour au Basic. On écrira:

```
JSR   $D42F
```

### 10) Scrutation du clavier

Nous avons vu des routines permettant d'afficher des caractères sur l'écran et représentant donc des sorties.

Notre description ne serait pas complète sans une routine permettant une entrée à partir du clavier.

Cette routine réside à l'adresse $F883.

Après l'appel de cette routine, l'Accumulateur contient le code ASCII de la touche enfoncée (ou 0 si aucune touche n'est enfoncée).

Le petit programme suivant permet d'aller scruter le clavier et d'afficher sur l'écran le caractère tapé au clavier.

```
1                       ORG    $5800
2                       EXC    INIT
3 5800 BDF883  INIT     JSR    $F883   ;SRUTE LE CLAVIER
4 5803 8100             CMPA   #$00    ;CARACTERE DISPONIBLE
5 5805 27F9             BEQ    INIT    ;NON, RECOMMENCE
6 5807 CE0503           LDX    #$0503  ;OUI, AFFICHE LE
7 580A BDD40C           JSR    $D40C
8 580D 7E5800           JMP    INIT
```

### 11) Émission de notes musicales

La routine d'adresse $FFAB permet d'envoyer une note de fréquence et de durée programmables dans le haut-parleur de votre téléviseur.

Le fonctionnement de cette routine est le suivant :

— l'accumulateur A doit être chargé avec un octet ($Ø1 à $FF) représentant la fréquence de la note,

— l'accumulateur B doit être chargé avec un octet ($Ø1 à $FF) représentant la durée de la note.

— la routine d'adresse $FFAB est ensuite appelée.

Le petit programme suivant permet de jouer un air qui vous semblera familier.

```
ASSEMBLEUR ALICE REV 1.00*
COPYRIGHT MATRA, 1984

  1                          ORG   $5600
  2                          EXC   INIT
  3 5800 8666    INIT        LDAA  #$66
  4 5802 BD5838              JSR   NOTE
  5 5805 8666                LDAA  #$66
  6 5807 BD5838              JSR   NOTE
  7 580A 8666                LDAA  #$66
  8 580C BD5838              JSR   NOTE
  9 580F 8677                LDAA  #$77
 10 5811 BD5838              JSR   NOTE
 11 5814 8687                LDAA  #$87
 12 5816 BD5838              JSR   NOTE
 13 5819 8677                LDAA  #$77
 14 581B BD5838              JSR   NOTE
 15 581E 8666                LDAA  #$66
 16 5820 BD5838              JSR   NOTE
 17 5823 8687                LDAA  #$87
 18 5825 BD5838              JSR   NOTE
 19 5828 8677                LDAA  #$77
 20 582A BD5838              JSR   NOTE
 21 582D 8677                LDAA  #$77
```

```
22 582F BD5838          JSR   NOTE
23 5832 8666            LDAA  #$66
24 5834 BD5838          JSR   NOTE
25 5837 39              RTS
26 5838 C605    NOTE    LDAB  #$05
27 583A BDFFAB          JSR   $FFAB
28 583D CEFFFF          LDX   #$FFFF
29 5840 09      LOOP    DEX
30 5841 26FD            BNE   LOOP
31 5843 39              RTS
32

0 ERREUR(S) PASSE 1
0 ERREUR(S) PASSE 2
SYMBOLES :
---------
INIT =5800 NOTE =5838 LOOP =5840
```

Le tableau des fréquences en fonction de la valeur prise par l'Accumulateur est le même que dans le cas de programmes Basic et de l'utilisation de l'instruction SOUND. Ce tableau est donné dans le manuel d'utilisation de l'Alice.

### 12) Émission d'un caractère sur écran ou imprimante

Cette routine permet d'envoyer un caractère soit sur l'écran soit sur une imprimante connectée à l'Alice.

Cette routine est appelée à l'adresse $F9C6.

Son fonctionnement est le suivant :

— on charge tout d'abord la case-mémoire d'adresse $E8 avec Ø ou 1 (Ø indique que le caractère sera envoyé sur l'écran tandis que 1 indique que le caractère sera envoyé sur l'imprimante) ;

— on charge la case-mémoire d'adresse $328Ø avec le numéro de la rangée où devra s'effectuer l'affichage (cas d'un affichage sur écran uniquement) ;

— on charge la case-mémoire d'adresse $3281 avec le numéro de la colonne où devra s'effectuer l'affichage (cas de l'écran uniquement);

— on charge l'accumulateur A avec le code ASCII du caractère que l'on désire afficher;

— on appelle la routine d'adresse $F9C6 par une instruction d'appel de sous-programme.

*Exemple 1*: On désire afficher la lettre T sur l'écran à la colonne 16, rangée 16. Le programme assembleur correspondant s'écrira:

```
 1                          ORG   $5800
 2                          EXC   INIT
 3 5800 8600    INIT        LDAA  #$00   ;AFFICHAGE ECRAN
 4 5802 B700E8              STAA  $E8
 5 5805 8610                LDAA  #$10   ;RANGEE 10
 6 5807 B73280              STAA  $3280
 7 580A 8610                LDAA  #$10   ;COLONNE 10
 8 580C B73281              STAA  $3281
 9 580F 8654                LDAA  #$54   ;LETTRE T
10 5811 BDF9C6              JSR   $F9C6
11 5814 7E5814 LOOP         JMP   LOOP
```

*Exemple 2*: On désire maintenant afficher le même caractère T sur imprimante. Cette lettre devra être validée par un retour chariot (CHR$(13) en Basic, $OD en assembleur). Le programme assembleur correspondant s'écrira:

```
1                          ORG   $5800
2                          EXC   INIT
3 5800 8601    INIT        LDAA  #$01   ;AFFICHAGE ECRAN
4 5802 B700E8              STAA  $E8
5 5805 8654                LDAA  #$54   ;LETTRE T
6 5807 BDF9C6              JSR   $F9C6
7 580A 860D                LDAA  #$0D   ;RETOUR CHARIOT
8 580C BDF9C6              JSR   $F9C6
9 580F 7E580F LOOP         JMP   LOOP
```

**13) Émission d'une chaîne de caractère sur écran ou imprimante**

Cette routine permet d'afficher sur écran ou sur imprimante une chaîne de caractères quelconque. Elle fonctionne de manière similaire à

la routine précédente. Il est de plus nécessaire de préciser l'emplacement mémoire contenant les codes ASCII de la chaîne à afficher. Les opérations suivantes doivent donc être effectuées :

— on charge la case-mémoire d'adresse $E8 avec Ø ou 1 (Ø indique que le caractère sera envoyé sur l'écran tandis que 1 indique que le caractère sera envoyé sur l'imprimante), comme nous l'avons vu précédemment ;

— on charge les cases-mémoire d'adresses $328Ø et $3281 avec respectivement les numéros de rangée et colonne de l'endroit où devra s'effectuer l'affichage (cas de l'écran uniquement) ;

— on charge le registre d'index X avec l'adresse, décrémentée de une unité, du premier caractère constituant la chaîne de caractères ;

— on appelle alors la routine d'adresse $E7A8 par une instruction JSR.

Le petit programme suivant illustre le fonctionnement de cette routine lors d'un affichage sur écran :

```
 1                          ORG   $5800
 2                          EXC   INIT
 3 5800 8600    INIT        LDAA  #$00
 4 5802 B700E8              STAA  $E8
 5 5805 8610                LDAA  #$10      ; RANGEE 16
 6 5807 B73280              STAA  $3280
 7 580A 8605                LDAA  #$05      ; COLONNE 5
 8 580C B73281              STAA  $3281
 9 580F CE5817              LDX   #$5817
10 5812 BDE7A8              JSR   $E7A8
11 5815 7E5815 LOOP         JMP   LOOP
12 5818 434543 TEXT         'CECI EST UN ESSAI
```

Lors d'un affichage sur imprimante, on pourrait écrire:

```
 1                          ORG    $5800
 2                          EXC    INIT
 3 5800 8601     INIT       LDAA   #$01   ;AFFICHAGE
                                           IMPRIMANTE
 4 5802 B700E8              STAA   $E8
 5 5805 CE5812              LDX    #$5812
 6 5808 BDE7A8              JSR    $E7A8 ;AFFICHAGE CHAINE
 7 580B 860D                LDAA   #$0D  ;RETOUR CHARIOT
 8 580D BDF9C6              JSR    $F9C6
 9 5810 7E5810 LOOP         JMP    LOOP
10 5813 434543 TEXT         'CECI EST UN ESSAI
```

L'envoi du "retour chariot" (code $OD) est destiné à provoquer l'impression du texte "ceci est un essai" qui sinon resterait stocké dans le buffer (tampon) situé dans l'imprimante jusqu'à ce qu'il soit complètement rempli. (En effet, une imprimante contient généralement une mémoire volatile de faible capacité destinée à conserver l'équivalent d'une ligne de texte, par exemple).

Nous en avons maintenant terminé avec la description des routines moniteur les plus utiles contenues dans la ROM de l'Alice. Elles devraient vous aider à écrire facilement des programmes assembleur possédant des Entrées-Sorties clavier/écran intéressantes.

Il existe de plus un certain nombre de cases-mémoire utilisées par le moniteur ou l'interpréteur Basic. Nous n'allons bien sûr pas toutes les passer en revue mais certaines d'entre elles vous seront bien utiles.

- adresse du curseur: la case-mémoire d'adresse $328Ø donne le numéro de la rangée où se trouve le curseur (Ø à 24).

De même la case-mémoire d'adresse $3281 donne le numéro de la colonne où se trouve le curseur (Ø à 79).

- indicateur de répétition automatique: la case-mémoire d'adresse $3ØØ8 permet de déterminer si les touches ont une fonction d'auto-repeat. Normalement elle contient la valeur $ØØ et cette fonction est inhibée. Lorsque par contre elle contient la valeur $Ø1, les touches du clavier auront une fonction d'Auto-Repeat.

# 7

# Les possibilités sonores de l'Alice

## 7.1. ACCÈS A PARTIR DU BASIC

Nous avons vu dans le chapitre consacré au Basic de l'Alice que la seule instruction disponible pour rajouter du son à vos programmes était l'instruction SOUND.

La syntaxe de cette instruction est, nous le rappelons, la suivante :

SOUND F,D

F désignant la fréquence de la note désirée et D sa durée.

Ne possédant qu'un générateur unique sur 3 octaves, l'Alice ne comporte que des possibilités sonores très limitées surtout si l'on se contente de cette simple instruction SOUND.

C'est pourquoi nous allons décrire dans les pages qui vont suivre un ensemble de routines en assembleur permettant d'animer vos programmes, qu'ils soient en Basic ou en Assembleur.

A partir du Basic, elles pourront être appelées à l'aide d'une instruction EXEC, tandis que dans un programme Assembleur, elles pourront être des sous-programmes, tout simplement.

## 7.2. ACCÈS A PARTIR DE L'ASSEMBLEUR

### 1. Génération d'une note isolée

Nous allons commencer par la routine la plus simple qui consiste à produire une note isolée de fréquence et de durée déterminées.

Elle est l'analogue de l'instruction Basic SOUND et similaire à la routine moniteur d'adresse $FFAB décrite dans le chapitre précédent.

Le programme correspondant est le suivant :

```
ASSEMBLEUR ALICE REV 1.00*
COPYRIGHT MATRA, 1984

 1                      ORG    $4000
 2                      EXC    PROG
 3                      ;PROGRAMME DE GENERATION DE NOTE ISOLEE
 4 4000 FF      DUREE   DFO    $FF     ;DUREE DE LA NOTE
 5 4001 0070    FREQ    DFD    $0070   ;FREQUENCE DE LA NOTE
 6 4003 8600    PROG    LDAA   #$00
 7 4005 F64000          LDAB   DUREE   ;INITIALISE COMPTEUR DE DUREE
 8 4008 88FF    LOOP1   EORA   #$FF
 9 400A B7BFFF          STAA   $BFFF
10 400D FE4001          LDX    FREQ    ;INITIALISE COMPTEUR FREQUENCE
11 4010 09      LOOP2   DEX            ;TERMINE?
12 4011 26FD            BNE    LOOP2   ;NON, ATTEND AVANT INVERSION
13 4013 5A              DECB           ;DUREE ECOULEE?
14 4014 26F2            BNE    LOOP1   ;NON, CONTINUE
15 4016 39              RTS
16

0 ERREUR(S) PASSE 1
0 ERREUR(S) PASSE 2
SYMBOLES :
--------
DUREE=4000 FREQ =4001 PROG =4003 LOOP1=4008 LOOP2=4010
```

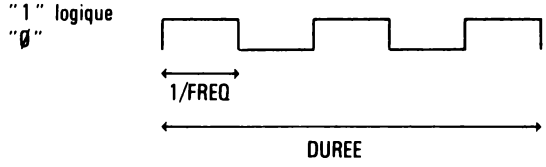
Il permet de générer dans le haut-parleur de votre téléviseur une note musicale de fréquence et de durée programmables.

La fréquence est déterminée par le contenu des deux cases-mémoire d'adresses $4001 et $4002. La durée est déterminée par le contenu de la case-mémoire d'adresse $4000. Bien sûr ces adresses, de même que l'adresse d'implantation en mémoire du programme ci-dessus, peuvent être modifiées à volonté.

Le fonctionnement de ce programme est le suivant :

Dans l'accumulateur, on stocke une valeur qui va être envoyée à l'adresse $BFFF correspondant au générateur de son.

En effet pour produire une note musicale, il suffit d'envoyer à cette adresse un signal carré d'allure suivante :




Un "1" logique est envoyé à cette adresse puis un "Ø" et ainsi de suite. Le temps pendant lequel le signal reste à "Ø" ou à "1" est égal à une demi-période du signal et correspond à 1/FREQ, FREQ étant caractéristique de la fréquence du signal et programmable dans le programme ci-dessus.

La durée du signal (DUREE) correspond au nombre de demi-périodes successives.

Examinons maintenant de plus près les instructions que constituent notre programme.

Le passage du signal de "1" à "Ø" et de "Ø" à "1" est réalisé à l'aide de l'instruction EORA ("OU" exclusif).

Le registre d'index est chargé périodiquement avec une valeur binaire de 16 bits caractérisant la fréquence du signal. La touche LOOP2 est donc parcourue un nombre de fois égal au contenu de ce registre X.

L'Accumulateur B, quant à lui, est chargé avec la variable DURÉE. La boucle LOOP1 permet donc de compter le nombre de demi-périodes du signal.

## 2. Génération d'un signal modulé en fréquence

Grâce au programme ci-dessous, nous nous proposons de générer un signal dont la fréquence varie continuement en fonction du temps de manière à imiter une sirène.

Le programme correspondant est le suivant :

```
ASSEMBLEUR ALICE REV 1.00*
COPYRIGHT MATRA, 1984

 1                      ORG    $4000
 2                      EXC    PROG
 3                      ;PROGRAMME DE GENERATION DE SON AVEC
 4                      ;MODULATION DE FREQUENCE
 5  4000 00      DUREE DFO    $00   ;DUREE DE LA NOTE ELEMENTAIRE
 6  4001 00      FREQ  DFO    $00   ;FREQUENCE DE LA NOTE
 7  4002 05      INT   DFO    $05   ;INTERVALLE DE FREQUENCE ENTRE 2 NOTES
 8  4003 00      VAL   DFO    $00   ;VALEUR ENVOYEE DANS LE GENERATEUR DE SON
 9  4004 B64001  PROG  LDAA   FREQ  ;PREMIERE FREQUENCE
10  4007 BB4002        ADDA   INT
11  400A B74001        STAA   FREQ
12  400D BD4012        JSR    SON   ;ENVOIE UN SON ELEMENTAIRE
13  4010 20F2          BRA    PROG
14  4012 C6FF    SON   LDAB   #$FF    ;INITIALISE COMPTEUR DE DUREE
15  4014 F04001        SUBB   FREQ
16  4017 F74000        STAB   DUREE
17  401A B64003 LOOP1  LDAA   VAL     ;INVERSE LE CRENEAU
18  401D 88FF          EORA   #$FF
19  401F B74003        STAA   VAL
20  4022 B7BFFF        STAA   $BFFF  ;ENVOIE VALEUR DANS LE GENERATEUR DE SON
21  4025 F64001        LDAB   FREQ   ;INITIALISE COMPTEUR FREQUENCE
22  4028 5A      LOOP2 DECB           ;TERMINE?
23  4029 26FD          BNE    LOOP2 ;NON, ATTEND AVANT INVERSION
24  402B 7A4000        DEC    DUREE ;DUREE ECOULEE?
25  402E 26EA          BNE    LOOP1 ;NON, CONTINUE
26  4030 39            RTS
27

0 ERREUR(S) PASSE 1
0 ERREUR(S) PASSE 2
SYMBOLES :
---------
DUREE=4000 FREQ =4001 INT  =4002 VAL  =4003 PROG =4004
  =4007 SON  =4012 LOOP1=401A LOOP2=4028
```

Les variables utilisées par ce programme sont les suivantes :

— DUREE détermine la durée d'une note élémentaire, le signal résultant étant constitué d'une suite de notes élémentaires de fréquences différentes.

— FREQ détermine la fréquence de la note élémentaire.

— INT détermine l'intervalle de fréquence qu'il existe entre deux notes élémentaires.

— VAL contient la valeur "Ø" ou "1" qui sera envoyé dans le générateur de son d'adresse $BFFF.

Le fonctionnement général du programme est donc le suivant :

— on considère une première fréquence égale à FREQ + INT et on envoie une note élémentaire de durée ($FF – FREQ). En soustrayant chaque valeur de FREQ à $FF, il est possible d'obtenir des notes élémentaires de durée constante et donc un signal résultant plus réaliste. Les notes élémentaires sont générées par le sous-programme SON.

— la fréquence est ensuite incrémenté (FREQ+INT+INT) de manière à générer la note élémentaire suivante.

— le programme continue de cette manière indéfiniment.

Les paramètres INT et de durée réelle (caractérisée par le $FF de l'instruction LDAB #$FF) peuvent être aisément modifiées à votre gré. Nous en avons terminé avec ce court chapitre. Comme nous l'avons dit, les possibilités sonores de l'Alice sont limitées mais les routines décrites ci-dessus permettront tout de même d'"égailler" avantageusement vos programmes.

# 8

# Les possibilités graphiques de l'Alice

## 8.1. INTRODUCTION

Dans ce chapitre, nous allons décrire de manière détaillée les différents modes d'affichage de l'Alice.

C'est sur ce sujet que les versions "32" et "9Ø" de l'Alice diffèrent le plus fondamentalement de l'"ancien" Alice 4K. Ceci est rendu possible par l'utilisation d'un contrôleur d'écran plus performant et que nous avons déjà mentionné dans le chapitre consacré à l'"Architecture du Système".

Son rôle est, nous l'avons dit, de lire de façon permanente les informations se trouvant dans la mémoire écran. Pour chacune des cases qu'elle comporte, il transforme le code qu'elle contient en un ensemble d'informations électriques permettant d'afficher caractères ou graphiques sur votre téléviseur.

La mémoire-écran de l'Alice possède 8 Koctets et est séparée de la mémoire utilisateur. Le microprocesseur n'y a donc pas accès directement et c'est le contrôleur d'écran qui se charge de sa gestion.

Ces 8 Koctets sont suffisants pour permettre d'afficher des caractères alphanumériques, semi-graphiques ou même des graphiques haute-résolution, le tout en couleurs.

## 8.2. LES ÉCRANS DE L'ALICE

### 8.2.1. Affichage sur 16 lignes de 32 colonnes

Lors de la mise sous tension, l'Alice est mis automatiquement en mode 16 lignes de 32 colonnes.

Il permet alors d'afficher des caractères alphanumériques ou semi-graphiques sur l'écran.

Ce mode d'affichage se présente sous la forme d'un cadre de fond vert entouré de noir et peut être réinitialisé à l'aide de l'instruction CLS 32.

La couleur du fond de l'écran peut aisément être modifiée à l'aide de l'instruction CLS suivie d'un numéro de couleur.

Nous rappelons ci-dessous les différentes couleurs disponibles sur l'Alice.

| *Code* | *Couleur* |
|---|---|
| Ø | noir |
| 1 | vert |
| 2 | jaune |
| 3 | bleu roi |
| 4 | rouge |
| 5 | ivoire |
| 6 | bleu pâle |
| 7 | mauve |
| 8 | orange |

C'est ainsi que CLS 1 correspond au mode d'affichage disponible lors de la mise sous tension.

Les caractères alphanumériques affichés sont de type majuscule et peuvent être affichés en vidéo normale ou inverse.

La commutation entre ces deux modes se fait par pression simultanée sur les touches SHIFT et Ø.

Chaque caractère alphanumérique est défini par une matrice de points de taille 8×1Ø.

L'obtention de graphismes basse-résolution peut se faire de deux façons :

— affichage de caractères semi-graphiques,

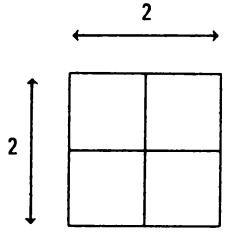
— utilisation des instructions SET, RESET et POINT.

Nous allons passer en revue ces deux méthodes :

*a) Affichage de caractères semi-graphiques*

Le clavier de l'Alice possède un certain nombre de touches permettant l'affichage de caractères semi-graphiques, au même titre que des caractères alphanumériques classiques. Ils sont au nombre de 16 et correspondent à toutes les combinaisons possibles de remplissage d'un "pavé" de taille 2×2.

Chaque caractère de l'écran 16×32 peut être divisé en 4 cases,




chacune d'entre elles pouvant être individuellement "éteinte" ou "allumée". L'ensemble des caractères semi-graphiques de l'Alice a déjà été décrit dans le chapitre consacré aux routines moniteurs.

L'affichage de ces caractères pourra être réalisé à l'aide d'instructions PRINT classiques. Leur couleur est alors uniforme sur la totalité de l'écran (noire sur fond de couleur spécifiée par l'instruction CLS).

Cependant, à chacun de ces caractères correspond un code qui peut être utilisé conjointement à une instruction de type CHR$.

Ces codes ont également été donnés dans le chapitre consacré aux routines moniteur.

Rappelons tout de même que le code de chaque caractère est composé de deux parties, l'une spécifiant son type (1 parmi 16 caractères semi-graphiques disponibles), l'autre spécifiant sa couleur (1 parmi 8 couleurs différentes, en plus du noir).

C'est ainsi que l'instruction :

PRINT CHR$(191) permet d'obtenir un carré.

En effet :

191 = 143 + 48

143 : code correspondant à toutes les cases du pavé 2x2 vides

48 : code correspondant à une couleur rouge

L'avantage de cette méthode est qu'elle permet de "dissimuler" des graphismes plus ou moins complexes à l'intérieur de chaînes de caractères dites "chaînes graphiques".

Pour produire un dessin sur l'écran, il suffira d'afficher une ou plusieurs chaînes à l'aide de l'instruction PRINT.

La construction de la chaîne pourra se faire caractère par caractère pour concaténations successives.

*Exemple* :

```
5 PRINT TAB(10)
10 PRINT TAB(10);CHR$(138);
20 FOR I=1 TO 10
30 PRINT CHR$(211);:NEXT I
40 PRINT CHR$(133)
50 FOR J=1 TO 6
60 PRINT TAB(10);CHR$(138);:GOSUB110;:PRINT CHR$(133):NEXT J
70 PRINT TAB(10);CHR$(138);
80 FOR I=1 TO 10
90 PRINT CHR$(220);:NEXT I
100 PRINT CHR$(133):END
110 FOR I=1 TO 10
120 PRINT CHR$(223);:NEXT I:RETURN
```

Le petit programme ci-dessus permet de dessiner un rectangle de couleur bleue sur l'écran.

L'utilisation de caractères semi-graphiques à partir de codes de type CHR$ peut sembler un peu fastidieuse quand il s'agit de réaliser des dessins complexes. C'est pourquoi existent les instructions SET, RESET et POINT que nous allons décrire ci-dessous.

**b) Utilisation des instructions SET, RESET et POINT**

Comme nous l'avons vu dans le chapitre consacré au Basic, ces instructions permettent d'allumer, d'éteindre ou de tester le contenu de points de l'écran.

Dans le cas d'un affichage de 16 lignes sur 32 colonnes, chaque caractère est divisé en une matrice 2×2 comme nous l'avons vu ci-dessus.

C'est ainsi que la résolution de l'affichage est égale à 64 dans le sens horizontal et 32 dans le sens vertical.

La syntaxe des instructions SET, RESET et POINT est la suivante :

— SET(X,Y,N)

X désignant l'abscisse du point que l'on désire allumer ($\emptyset \leqslant X \leqslant 63$), Y désignant son ordonnée ($\emptyset \leqslant Y \leqslant 31$) et N son code couleur.

Pour allumer le point de coordonnées X=1Ø et Y=15 et lui donner la couleur rouge, on écrira alors :

```
SET(1Ø,15,4)
```

après avoir fait un CLS Ø qui colorie tout l'écran en noir.

— RESET(X,Y)

X et Y désignant les coordonnées du point à éteindre.
Pour éteindre le point de coordonnées X=1Ø et Y=15, on écrira donc :

```
RESET(1Ø,15)
```

— POINT(X,Y)

X,Y désignant les coordonnées du point dont on veut connaître l'état. En effet, trois cas peuvent se présenter :

— soit il a été allumé par une instruction de type SET et possède une couleur donnée et dans ce cas l'instruction POINT renvoie le code couleur correspondant (Ø à 8) ;

— soit il est éteint et l'instruction POINT renvoie la valeur Ø ;

— soit il se trouve à un endroit où un caractère alphanumérique est affiché et dans ce cas, l'instruction POINT renvoie la valeur – 1.

*Exemple*: Après un CLS Ø, tapez SET (63,31,4) puis:
PRINT POINT (63,31)
vous devez obtenir la valeur 4.

Le programme suivant utilise l'instruction SET et permet de tracer un triangle sur l'écran.

```
5 CLS Ø:J=3Ø
1Ø FOR I=2Ø TO 4Ø
2Ø SET(I,3Ø,6):NEXT I
3Ø FOR I=2Ø TO 3Ø
4Ø SET(I,J,6):J=J-1:NEXT
5Ø FOR I=31 TO 41
6Ø J=J+1:SET(I,J,6):NEXT
```

### 8.2.2. Affichage sur 25 lignes de 4Ø colonnes

Ce mode d'affichage est pratiquement similaire au mode 16 lignes x 32 colonnes décrit dans les lignes précédentes.

Il permet d'afficher des caractères alphanumériques ou semi-graphiques sur l'écran et peut être réinitialisé à l'aide de l'instruction CLS 4Ø.

La couleur du fond de l'écran peut être modifiée à l'aide de l'instruction CLS suivie d'un numéro de couleur.

Le fonctionnement des instructions SET, RESET et POINT ne diffère que par les valeurs prises par les coordonnées X et Y.

En effet, le mode d'affichage de 25 lignes sur 4Ø colonnes permet d'obtenir une résolution de 8Ø en horizontal et de 5Ø en vertical, dans le cas d'un affichage graphique basse-résolution.

Nous n'allons pas plus nous attarder sur ce mode d'affichage.

### 8.2.3. Affichage sur 25 lignes de 8Ø colonnes

Ce mode d'affichage est une particularité intéressante pour un micro-ordinateur de la catégorie de l'Alice puisque les affichages "8Ø

colonnes" sont en général réservés à des machines plus "professionnelles".

Comme son nom l'indique, ce mode d'affichage permet d'afficher 25 lignes de 8Ø caractères. Afin d'obtenir une bonne lisibilité, il vous faudra disposer d'un téléviseur de bonne qualité ou mieux d'un moniteur couleur.

Ce mode d'affichage est plutôt destiné à des applications de type "traitement de texte" pour lesquelles 4Ø caractères par ligne ne sont pas suffisants.

Dans ce mode, l'Alice ne possède plus de jeu de caractères semi-graphiques. En remplacement, il permet d'afficher des caractères minuscules.

Les jeux de caractères majuscules et minuscules peuvent être commutés grâce à une pression simultanée sur les touches SHIFT et Ø. Ce mode d'affichage existe sous deux formes :

— affichage de caractères noirs sur fond vert et sélectionné par l'instruction CLS 8Ø ;

— affichage de caractères verts sur fond noir et sélectionné par l'instruction CLS 81.

Les instructions CLS Ø à CLS 8 ne fonctionnent pas dans ce mode. Cependant, les couleurs d'affichage des caractères, des graphismes (comme nous le verrons plus loin) et la couleur du fond de l'écran peuvent être modifiés à l'aide de l'instruction spécialisée SET*.

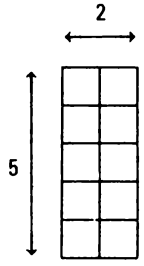
L'écran, en mode 8Ø colonnes est caractérisé par un rectangle (dans lequel seront affichés les caractères et graphismes) appelé "fond de l'écran" entouré d'un cadre ou "marge". Le fond de l'écran est caractérisé par la couleur d'"Intensité".

Les caractères alphanumériques sont affichés en prenant la couleur de marge. Ils sont définis par une matrice de points de taille 6×6, réduite par rapport aux cas des affichages sur 32 et 4Ø colonnes.

En mode 8Ø colonnes, l'écran peut être divisé en 2ØØØØ points élémentaires. Ceci permet d'obtenir un affichage moyenne résolution de 16Ø (résolution horizontale) sur 125 (résolution verticale).

Chacun de ces points peut être isolément allumé ou éteint et prend alors une couleur de "demi-intensité".

Ces points sont regroupés sous forme de matrices 2×5 (il existe donc 8Ø matrices selon l'axe horizontal et 25 matrices selon l'axe vertical).




Lorsqu'un point élémentaire est allumé, la matrice à laquelle il appartient est considérée comme un caractère et prend en conséquence la couleur de marge, sauf le point considéré qui, lui, prend l'une des trois couleurs :

— couleur de marge,
— couleur de demi-intensité,
— couleur d'intensité.

Ces couleurs peuvent être sélectionnées grâce à l'instruction SET* comme nous l'avons déjà dit.

La syntaxe de cette instruction est :

SET* M,I,D

M désignant la couleur de marge, I la couleur d'intensité et D la couleur de demi-intensité.

M, I et D sont données par un code couleur compris entre Ø et 8 et similaire à ce qui a été vu jusqu'à présent.

*Exemple* :

SET* 2,3,4

permet d'obtenir :

— des caractères jaunes (couleur de marge),
— un fond bleu roi (couleur d'intensité),
— une couleur de demi-intensité rouge.

L'affichage d'un point élémentaire se fait grâce à une instruction SET dont la syntaxe est :

SET(X,Y,N)

X désignant l'abscisse du point à allumer (Ø≤X≤159), Y son ordonnée (Ø≤Y≤124) et N permettant de choisir la couleur d'affichage de chaque point élémentaire.

N peut prendre les valeurs suivantes :

— Ø ce qui donne un point dont la couleur est la couleur de marge ;

— 1 ce qui donne un point dont la couleur est la couleur d'intensité ;

— 2 ce qui donne un point dont la couleur est la couleur de demi-intensité.

C'est ainsi que, après avoir tapé SET* 2,3,4 puis SET (1Ø,2Ø,2) on obtiendra un point rouge.

L'instruction RESET, dont la syntaxe est RESET(X,Y) permet d'éteindre un point élémentaire allumé préalablement à l'aide d'une instruction SET.

Nous avons donné dans le chapitre consacré au Basic de l'Alice un exemple d'utilisation des instructions SET et RESET dans un programme de tracé de courbe en haute-résolution.
Les lecteurs intéressés pourront donc se reporter à ce chapitre.

L'instruction POINT, quant à elle, fonctionne de manière similaire à celle rencontrée dans le cas des affichages sur 32 ou 4Ø colonnes.

Si le point est occupé par un caractère, elle renvoie la valeur — 1, sinon elle renvoie un code égal à Ø, 1 ou 2 selon que le point considéré a la couleur de marge, d'intensité ou de demi-intensité.

Nous en avons terminé avec ce chapitre qui vous permettra, nous l'espérons, de tirer au mieux parti des possibilités graphiques offertes par votre Alice. Les instructions rencontrées ici en permettent un accès aisé à partir de l'Interpréteur Basic.

Les lecteurs intéressés trouveront dans le chapitre 6 un certain nombre de routines moniteur permettant de produire des affichages alphanumériques ou semi-graphiques directement en Assembleur.

## TABLE DE CONVERSION HEXADÉCIMAL → DÉCIMAL

| | Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ø | Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1Ø | 11 | 12 | 13 | 14 | 15 |
| 1 | 16 | 17 | 18 | 19 | 2Ø | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 3Ø | 31 |
| 2 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 4Ø | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 3 | 48 | 49 | 5Ø | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 6Ø | 61 | 62 | 63 |
| 4 | 64 | 65 | 66 | 67 | 68 | 69 | 7Ø | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 5 | 8Ø | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 9Ø | 91 | 92 | 93 | 94 | 95 |
| 6 | 96 | 97 | 98 | 99 | 1ØØ | 1Ø1 | 1Ø2 | 1Ø3 | 1Ø4 | 1Ø5 | 1Ø6 | 1Ø7 | 1Ø8 | 1Ø9 | 11Ø | 111 |
| 7 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 12Ø | 121 | 122 | 123 | 124 | 125 | 126 | 127 |
| 8 | -128 | -127 | -126 | -125 | -124 | -123 | -122 | -121 | -12Ø | -119 | -118 | -117 | -116 | -115 | -114 | -113 |
| 9 | -112 | -111 | -11Ø | -1Ø9 | -1Ø8 | -1Ø7 | -1Ø6 | -1Ø5 | -1Ø4 | -1Ø3 | -1Ø2 | -1Ø1 | -1ØØ | -99 | -98 | -97 |
| A | -96 | -95 | -94 | -93 | -92 | -91 | -9Ø | -89 | -88 | -87 | -86 | -85 | -84 | -83 | -82 | -81 |
| B | -8Ø | -79 | -78 | -77 | -76 | -75 | -74 | -73 | -72 | -71 | -7Ø | -69 | -68 | -67 | -66 | -65 |
| C | -64 | -63 | -62 | -61 | -6Ø | -59 | -58 | -57 | -56 | -55 | -54 | -53 | -52 | -51 | -5Ø | -49 |
| D | -48 | -47 | -46 | -45 | -44 | -43 | -42 | -41 | -4Ø | -39 | -38 | -37 | -36 | -35 | -34 | -33 |
| E | -32 | -31 | -3Ø | -29 | -28 | -27 | -26 | -25 | -24 | -23 | -22 | -21 | -2Ø | -19 | -18 | -17 |
| F | -16 | -15 | -14 | -13 | -12 | -11 | -1Ø | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

## TABLEAU RÉCAPITULATIF DES INSTRUCTIONS DU 68Ø3

### Instructions portant sur les Accumulateurs et la mémoire

| *Instruction* | *Mode d'adressage* | | | | | *Indicateurs affectés* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mnémonique | IMM | DIR | IND | ETE | IMP | H | I | N | Z | V | C |
| ABA | | | | | 1B | ↕ | | ↕ | ↕ | ↕ | ↕ |
| ABX | | | | | 3A | | | | | | |
| ADCA | 89 | 99 | A9 | B9 | | ↕ | | ↕ | ↕ | ↕ | ↕ |
| ADCB | C9 | D9 | E9 | F9 | | ↕ | | ↕ | ↕ | | ↕ |
| ADDA | 8B | 9B | AB | BB | | ↕ | | ↕ | ↕ | ↕ | ↕ |
| ADDB | CB | DB | EB | FB | | ↕ | | ↕ | ↕ | ↕ | ↕ |
| ADDD | C3 | D3 | E3 | F3 | | | | ↕ | ↕ | ↕ | ↕ |
| ANDA | 84 | 94 | A4 | B4 | | | | ↕ | ↕ | Ø | |
| ANDB | C4 | D4 | E4 | F4 | | | | ↕ | ↕ | Ø | ↕ |
| ASL | | | 68 | 78 | | | | ↕ | ↕ | ↕ | ↕ |
| ASLA | | | | | 48 | | | ↕ | ↕ | ↕ | ↕ |
| ASLB | | | | | 58 | | | ↕ | ↕ | ↕ | ↕ |
| ASLD | | | | | Ø5 | | | ↕ | ↕ | ↕ | ↕ |
| ASR | | | 67 | 77 | | | | ↕ | ↕ | ↕ | ↕ |
| ASRA | | | | | 47 | | | ↕ | ↕ | ↕ | ↕ |

| *Instruction* | *Mode d'adressage* | | | | | *Indicateurs affectés* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mnémonique | IMM | DIR | IND | ETE | IMP | H | I | N | Z | V | C |
| ASRB | | | | | 57 | | | ↕ | ↕ | ↕ | ↕ |
| BITA | 85 | 95 | A5 | B5 | | | | ↕ | ↕ | ø | |
| BITB | C5 | D5 | E5 | F5 | | | | ↕ | ↕ | ø | |
| CBA | | | | | 11 | | | ↕ | ↕ | ↕ | ↕ |
| CLR | | | 6F | 7F | | | | ø | 1 | ø | ø |
| CLRA | | | | | 4F | | | ø | 1 | ø | ø |
| CLRB | | | | | 5F | | | ø | 1 | ø | ø |
| CMPA | 81 | 91 | A1 | B1 | | | | ↕ | ↕ | ↕ | ↕ |
| CMPB | C1 | D1 | E1 | F1 | | | | ↕ | ↕ | ↕ | ↕ |
| COM | | | 63 | 73 | | | | ↕ | ↕ | ø | 1 |
| COMA | | | | | 43 | | | ↕ | ↕ | ø | 1 |
| COMB | | | | | 53 | | | ↕ | ↕ | ø | 1 |
| DAA | | | | | 19 | | | ↕ | ↕ | ↕ | ↕ |
| DEC | | | 6A | 7A | | | | ↕ | ↕ | ↕ | |
| DECA | | | | | 4A | | | ↕ | ↕ | ↕ | |
| DECB | | | | | 5A | | | ↕ | ↕ | ↕ | |
| EORA | 88 | 98 | A8 | B8 | | | | ↕ | ↕ | ø | |
| EORB | C8 | D8 | E8 | F8 | | | | ↕ | ↕ | ø | |
| INC | | | 6C | 7C | | | | ↕ | ↕ | ↕ | |
| INCA | | | | | 4C | | | ↕ | ↕ | ↕ | |
| INCB | | | | | 5C | | | ↕ | ↕ | ↕ | |
| LDAA | 86 | 96 | A6 | B6 | | | | ↕ | ↕ | ø | |
| LDAB | C6 | D6 | E6 | F6 | | | | ↕ | ↕ | ø | |
| LDD | CC | DC | EC | FC | | | | ↕ | ↕ | ø | |
| LSL | | | 68 | 78 | | | | ↕ | ↕ | ↕ | ↕ |
| LSLA | | | | | 48 | | | ↕ | ↕ | ↕ | ↕ |
| LSLB | | | | | 58 | | | ↕ | ↕ | ↕ | ↕ |
| LSLD | | | | | ø5 | | | ↕ | ↕ | ↕ | ↕ |
| LSR | | | 64 | 74 | | | | ø | ↕ | ↕ | ↕ |
| LSRA | | | | | 44 | | | ø | ↕ | ↕ | ↕ |
| LSRB | | | | | 54 | | | ø | ↕ | ↕ | ↕ |
| LSRD | | | | | ø4 | | | ø | ↕ | ↕ | ↕ |
| MUL | | | | | 3D | | | | | | ↕ |
| NEG | | | 6ø | 7ø | | | | ↕ | ↕ | ↕ | ↕ |
| NEGA | | | | | 4ø | | | ↕ | ↕ | ↕ | ↕ |
| NEGB | | | | | 5ø | | | ↕ | ↕ | ↕ | ↕ |
| NOP | | | | | ø1 | | | | | | |
| ORAA | 8A | 9A | AA | BA | | | | ↕ | ↕ | ø | |
| ORAB | CA | DA | EA | FA | | | | ↕ | ↕ | ø | |
| PSHA | | | | | 36 | | | | | | |
| PSHB | | | | | 37 | | | | | | |
| PULA | | | | | 32 | | | | | | |
| PULB | | | | | 33 | | | | | | |
| ROL | | | 69 | 79 | | | | ↕ | ↕ | ↕ | ↕ |
| ROLA | | | | | 49 | | | ↕ | ↕ | ↕ | ↕ |
| ROLB | | | | | 59 | | | ↕ | ↕ | ↕ | ↕ |
| ROR | | | 66 | 76 | | | | ↕ | ↕ | ↕ | ↕ |

| *Instruction* | *Mode d'adressage* | | | | | *Indicateurs affectés* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mnémonique | IMM | DIR | IND | ETE | IMP | H | I | N | Z | V | C |
| RORA | | | | | 46 | | | ↕ | ↕ | ↕ | ↕ |
| RORB | | | | | 56 | | | ↕ | ↕ | ↕ | ↕ |
| SBA | | | | | 10 | | | ↕ | ↕ | ↕ | ↕ |
| SBCA | 82 | 92 | A2 | B2 | | | | ↕ | ↕ | ↕ | ↕ |
| SBCB | C2 | D2 | E2 | F2 | | | | ↕ | ↕ | ↕ | ↕ |
| STAA | | 97 | A7 | B7 | | | | ↕ | ↕ | 0 | |
| STAB | | D7 | E7 | F7 | | | | ↕ | ↕ | 0 | |
| STD | | DD | ED | FD | | | | ↕ | ↕ | 0 | |
| SUBA | 80 | 90 | A0 | B0 | | | | ↕ | ↕ | ↕ | ↕ |
| SUBB | C0 | D0 | E0 | F0 | | | | ↕ | ↕ | ↕ | ↕ |
| SUBD | 83 | 93 | A3 | B3 | | | | ↕ | ↕ | ↕ | ↕ |
| TAB | | | | | 16 | | | ↕ | ↕ | 0 | |
| TBA | | | | | 17 | | | ↕ | ↕ | 0 | |
| TST | | | 6D | 7D | | | | ↕ | ↕ | 0 | 0 |
| TSTA | | | | | 4D | | | ↕ | ↕ | 0 | 0 |
| TSTB | | | | | 5D | | | ↕ | ↕ | 0 | 0 |

## Instructions portant sur les registres d'index, pointeur de pile et CCR

| *Instruction* | *Mode d'adressage* | | | | | *Indicateurs affectés* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mnémonique | IMM | DIR | IND | ETE | IMP | H | I | N | Z | V | C |
| CPX | 8C | 9C | AC | BC | | | | ↕ | ↕ | ↕ | ↕ |
| DEX | | | | | 09 | | | | ↕ | | |
| DES | | | | | 34 | | | | | | |
| INX | | | | | 08 | | | | ↕ | | |
| INS | | | | | 31 | | | | | | |
| LDX | CE | DE | EE | FE | | | | ↕ | ↕ | 0 | |
| LDS | 8E | 9E | AE | BE | | | | ↕ | ↕ | 0 | |
| STX | | DF | EF | FF | | | | ↕ | ↕ | 0 | |
| STS | | 9F | AF | BF | | | | ↕ | ↕ | 0 | |
| TXS | | | | | 35 | | | | | | |
| TSX | | | | | 30 | | | | | | |
| ABX | | | | | 3A | | | | | | |
| PSHX | | | | | 3C | | | | | | |
| PULX | | | | | 38 | | | | | | |
| CLC | | | | | 0C | | | | | | 0 |
| CLI | | | | | 0E | | 0 | | | | |
| CLV | | | | | 0A | | | | | 0 | |
| SEC | | | | | 0D | | | | | | 1 |
| SEI | | | | | 0F | | 1 | | | | |
| SEV | | | | | 0B | | | | | 1 | |
| TAP | | | | | 06 | ↕ | ↕ | ↕ | ↕ | ↕ | ↕ |
| TPA | | | | | 07 | | | | | | |

## Instructions de saut

| *Instruction* | *Mode d'adressage* | | | | | *Indicateurs affectés* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mnémonique | DIR | REL | IND | ETE | IMP | H | I | N | Z | V | C |
| BRA | | 2Ø | | | | | | | | | |
| BRN | | 21 | | | | | | | | | |
| BCC | | 24 | | | | | | | | | |
| BCS | | 25 | | | | | | | | | |
| BEQ | | 27 | | | | | | | | | |
| BGE | | 2C | | | | | | | | | |
| BGT | | 2E | | | | | | | | | |
| BHI | | 22 | | | | | | | | | |
| BHS | | 24 | | | | | | | | | |
| BLE | | 2F | | | | | | | | | |
| BLO | | 25 | | | | | | | | | |
| BLS | | 23 | | | | | | | | | |
| BLT | | 2D | | | | | | | | | |
| BMI | | 2B | | | | | | | | | |
| BNE | | 26 | | | | | | | | | |
| BVC | | 28 | | | | | | | | | |
| BVS | | 29 | | | | | | | | | |
| BPL | | 2A | | | | | | | | | |
| BSR | | 8D | | | | | | | | | |
| JMP | | | 6E | 7E | | | | | | | |
| JSR | 9D | | AD | BD | | | | | | | |
| NOP | | | | | Ø1 | | | | | | |
| RTI | | | | | 3B | ↕ | ↕ | ↕ | ↕ | ↕ | ↕ |
| RTS | | | | | 39 | | | | | | |
| SWI | | | | | 3F | | 1 | | | | |
| WAI | | | | | 3E | | | | | | |

## NOTATIONS

IMM = mode d'adressage immédiat.

DIR = mode d'adressage direct.

IND = mode d'adressage indexé

ETE = mode d'adressage étendu.

IMP = mode d'adressage implicite.

REL = mode d'adressage relatif.

↕ = indicateur mis à 1 ou à Ø selon le résultat de l'instruction considérée.

1 = indicateur mis à 1 après exécution de l'instruction considérée.

Ø = indicateur mis à Ø après exécution de l'instruction considérée.