MICRO-ORDINATEURS

L'ASSEMBLEUR FACILE DU 6809

François BERNARD

sylvine bouvry - j.p. darai

EYROLLES

L'ASSEMBLEUR FACILE DU 6809

«La loi du 11 mars 1957 n'autorisant, aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les «copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective» et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, «toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite» (alinéa 1er de l'article 40)».
«Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal».

© Éditions EYROLLES, 1984

L'ASSEMBLEUR FACILE DU 6809

par

François BERNARD

Collection animée
par Richard SCHOMBERG

61, boulevard Saint-Germain — 75005 Paris
1985

DANS LA MÊME COLLECTION

SCHOMBERG - Le Basic Universel.
SCHOMBERG - Micro-ordinateurs : Comment ça marche ?
HERNANDEZ - Pascal par l'exemple.
NOLLET - La conduite du ZX 81.
PELLIER - La conduite du TRS 80.
LADEVIE - Votre gestion avec BASIC sur micro-ordinateur.
QUEINNEC - Langage d'un autre type : LISP.
PELLIER - Programmez vos jeux d'action rapide sur TRS 80.
ASTIER - La conduite de l'APPLE II.
Tome 1 : le Basic de l'APPLE II.
Tome 2 : le système graphique et l'assembleur de l'APPLE II.
MONTEIL - L'assembleur facile du 6502 et du 6510.
LEPAPE - L'assembleur facile du Z 80.
OROS et PERBOST - ZX 81 à la conquête des jeux.
- CASSETTE - ZX 81 à la conquête des jeux.
PERBOST - CASSETTE N° 2 - 13 jeux 1 K.
DAX - CP/M et sa famille.
NOLLET - Langage machine, trucs et astuces sur ZX 81.
BICKING - La conduite du PC 1212 (ou TRS 80 pocket).
TEJA - Apprenez à parler à votre ordinateur.
MONTEIL - La conduite du VIC 20.
PLOUIN - La conduite de l'IBM-PC.
SAGUEZ - Télécommande avec votre micro-ordinateur.
PELLIER - Tout sur les disques du TRS 80 modèles I et III.
OROS et PERBOST - La conduite du FX-702 P.
GROS - La conduite du PC 1500.
HARWOOD - Jeux et applications pour ZX SPECTRUM.
HARTNELL - Le grand livre du ZX SPECTRUM.
HARTNELL et JONES - La conduite du ZX SPECTRUM.
VULDY - Graphisme 3 D sur votre micro-ordinateur.
BOUQUEROD - Des extensions à construire pour votre ZX 81.
PINSON - Le Basic en gestion sur Apple II.
WILLARD - La conduite du TI 99.
CEYRAT - Mon TI 99/4A.
DELANNOY - Les fichiers en Basic sur micro-ordinateur.
AUBERT - Pratiquez l'intelligence artificielle.
PERBOST et MASSE - VIC 20 à la conquête des jeux.
TERRAL - La conduite du T O7.
ASTIER - La conduite de l'ORIC-1.
MONTEIL - Premiers pas en LOGO.
MONTEIL - La conduite du COMMODORE 64.
Tome 1 : Basic, graphisme et son.
Tome 2 : Langage-machine entrées/sorties et périphériques.
OROS - La conduite de l'ATARI 400/800.
WILLARD - TI 99 à la conquête des jeux.
KRUTCH - Expériences d'intelligence artificielle en Basic.
ASTIER - ORIC-1 à la conquête des jeux.

5.4. Les instructions logiques 92
5.5. Les instructions sur le registre d'état 110
5.6. Les instructions de comparaison 114
5.7. Les instructions de branchement 117
5.8. Les instructions d'appel et de retour de sous-programme ... 124
5.9. Les instructions sur la pile 125
5.10. Les instructions spéciales 128

6. Les entrées-sorties 133

7. La mise au point d'un programme en assembleur 139

Annexe : Tableau récapitulatif des instructions du 6809 151

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.

Avant-propos

Beaucoup d'entre vous sont probablement peu familiarisés avec l'ASSEMBLEUR en général, et sans doute encore moins avec l'ASSEMBLEUR du 68Ø9. Par contre, certains doivent déjà avoir une habitude du BASIC.

Ce livre s'adresse donc à tous ceux, à toutes celles qui ont décidé d'aborder l'"Informatique Individuelle" un peu plus en profondeur afin de voir réellement "comment ça marche".

Dès lors que la décision est prise il reste deux solutions: soit vous vous plongez à corps perdu dans la mer des instructions au risque de vous y noyer, soit vous abordez tranquillement le problème en utilisant au maximum ce que vous connaissez déjà: le BASIC. Ensuite, comme rien ne vaut l'expérience personnelle, nous décrivons l'assembleur de l'un des microprocesseurs les plus utilisés actuellement et surtout l'un des plus performants: le 68Ø9. En plus de l'inévitable jeu d'instructions nous vous donnons quelques conseils sur la façon de bien programmer et de faire tourner vos programmes.

Afin de prendre un bon départ, des exemples de programmes sont largement développés et commentés.

Mais jugez-en vous-même en lisant ce livre...

Nous n'avons pas abordé le problème de la connexion entre les langages évolués (BASIC par exemple) et l'ASSEMBLEUR, étant donné que ce problème est spécifique à chaque type de machine. Nous vous renvoyons donc aux ouvrages "La Conduite du..." parus dans la même collection.

Table des matières

Avant-propos IX

1. Introduction à l'assembleur 1

2. Les systèmes numériques 9

- 2.1. Le binaire 9
- 2.2. L'hexadécimal 10
- 2.3. Le code BCD 12
- 2.4. Le code ASCII 13
- 2.5. Représentation des nombres négatifs 15

3. La syntaxe assembleur 68Ø9 17

- 3.1. Introduction 17
- 3.2. La syntaxe assembleur 18
- 3.3. L'assemblage 28
- 3.4. Des assembleurs plus performants 31

4. Description du 68Ø9 33

- 4.1. Les registres internes du 68Ø9 33
- 4.2. Les différents modes d'adressage du 68Ø9 44

5. Le jeu d'instructions du 68Ø9 57

- 5.1. Introduction 57
- 5.2. Les instructions de chargement 58
- 5.3. Les instructions arithmétiques 72

5311 Addition en binaire nbres signés.
explication du bit C Carry retenue débordante.
V Overflow dépassement de capacité

Pellier - Langage machine, trucs et astuces sur ZX SPECTRUM.
Saguez et Andrieux - Maîtrisez les interfaces de votre micro-ordinateur.
De Geeter - Forth pour micros.
Astier - ATMOS à la conquête des jeux.
Crowther et Hartley - MO5 et TO7 à la conquête des jeux.
Monteil - Introduction à l'IBM-PC Junior.
Oros - La conduite des ATARI XL - 600 XL, 800 XL, 1245 XLD.
Perbost - ZX SPECTRUM à la conquête des jeux.
Bernard - L'assembleur facile du 6809.
Garrec et Vignet - Des extensions à construire pour votre ORIC-ATMOS.

LOGILIVRES EYROLLES (logiciels sur cassettes)

Pellier - Kamikaze (jeu pour ZX Spectrum).
Pellier - Astéroïdes (jeu pour ZX Spectrum).
Pellier - Othello/Isola (jeux pour ZX Spectrum).
Pellier - Éditeur/Assembleur pour ZX Spectrum.
Perbost et Masse - VIC 20 version de base à la conquête des jeux.
Haddadi - Calcul des structures sur PC 1500/PC 2.
Vanryb et Politis - LYNX. Dictée musicale. Générateur de caractères.
Rosenthal - Résistance des matériaux sur ORIC-1 et ORIC-ATMOS.

1

Introduction à l'assembleur

La plupart de ceux qui vont lire ce livre possèdent certainement, ou bien ont la possibilité d'utiliser, une de ces machines que l'on nomme "ordinateur individuel", et sont à ce titre au moins un peu familiarisés avec le BASIC.

Pour ce premier contact avec l'ASSEMBLEUR, qui risque d'occuper une bonne partie de vos loisirs pendant des jours, des mois, pourquoi pas des années, nous avons choisi de partir de quelque chose que vous connaissez déjà : le BASIC.

Nous rentrerons par la suite un peu plus dans les détails afin de vous dire finalement quel intérêt il peut y avoir de pouvoir programmer son "micro" en assembleur.

Soit le petit programme BASIC suivant :

```
1Ø INPUT A
2Ø IF A<Ø THEN GOTO 5Ø
3Ø B=-A
4Ø GOTO 6Ø
5Ø B=A
6Ø PRINT B
7Ø 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 :

```
        LDA     VAR       ; CHARGE A
        BLT     NEG       ; SI NEGATIF, B=A
        NEGA              ; SI POSITIF, B=-A
NEG     STA     AFFICH ,  ; RANGE B 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 "LDA" 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 "LDA". Dans cette zone se trouvent les mnémoniques ou ensembles de lettres correspondant à des instructions exécutables par le microprocesseur. Notons que ces mnémoniques doivent être traduits sous forme binaire afin de pouvoir être compris par la machine (nous reviendrons sur ce point un petit 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 dans l'Accumulateur, qui est une case-mémoire particulière située à l'intérieur du 68Ø9, 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 AFFICH. 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'Accumulateur 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 la scrutation du clavier afin d'y introduire une donnée et la ranger à l'adresse VAR.

— l'autre permettant d'afficher sur l'écran cathodique ou sur l'imprimante le contenu de la mémoire AFFICH.

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, la plupart des micro-ordinateurs individuels ne possèdent qu'un interpréteur qui, comme son nom l'indique, traduit tout d'abord chaque ligne de programme en une série d'instructions "machine" qui sont ensuite exécutées par le microprocesseur.

Il arrivera donc probablement un jour où, las d'attendre que l'ordinateur veuille bien sortir ses résultats, vous vous résoudrez à revenir aux sources et à vous plonger dans l'assembleur.

En pratique, vous l'utiliserez :

— chaque fois que le facteur temps d'exécution aura une grande importance : par exemple lorsqu'il y a de longs calculs à effectuer, lorsqu'on programme des jeux animés sur écran, ou lorsque l'on désire faire de l'acquisition de données.

— lorsque vous aurez besoin de fonctions spécifiques dont votre BASIC n'est pas doté ; exemple : tracer une droite reliant 2 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 à 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 5ØØ ØØØ additions par seconde (oui, oui, vous avez bien lu!!).

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. Tous les microprocesseurs du marché possèdent donc un jeu d'instructions plus ou moins étendu qui leur 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.

Avant de voir de quelle façon sont codées ces instructions, il nous faut tout 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" et 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 68Ø9 et de tous les microprocesseurs 8 bits du marché, est composée de mots de 8 bits ou octets qui ne sont en fait que les cases-mémoires dont nous avons parlé ci-dessus. Chacune d'elles reçoit une adresse comprise entre 0 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 68Ø9 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) ou de doubles octets (le jeu d'instructions du 68Ø9 comportant en effet plus de 256 éléments), chacun réalisant une fonction spécifique : on les nomme code-opérations.

Par exemple le mot 1ØØØ1Ø11 veut dire "additionne au contenu de l'Accumulateur A la valeur binaire suivant immédiatement le code-opération". Cette valeur binaire, codée sur un octet représente ce que l'on appelle l'opérande qui, selon les cas, nécessite Ø, 1 ou 2 mots-mémoire.

Par définition, un programme exécutable par un microprocesseur sera une suite 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.

Il apparaît comme évident qu'il n'est pas possible de développer de longs programmes en binaire car la probabilité d'erreur est très grande et la probabilité de retrouver une erreur très faible.

Une première solution consiste à coder les instructions non pas sur 8 bits (ØØØØØØØØ à 11111111) mais sur deux chiffres hexadécimaux (ØØ à FF) : nous reviendrons sur la numérotation hexadécimale dans le prochain chapitre qui traite des systèmes numériques. Les instructions et les opérandes sont alors introduites en hexadécimal mais nécessitent bien sûr d'être traduites en binaire avant d'être traitées. Cette tâche sera confiée à un programme qui aura dû lui-même être écrit préalablement. Ceci dit cette solution n'est pas idéale car il faut savoir à quel code-opération correspond telle instruction et inversement.

Exemple : 8B correspond à ADDA (addition sans retenu en adressage immédiat avec 8B=1ØØØ1Ø11).

C'est pourquoi les programmeurs ont créé l'Assembleur. Ses caractéristiques principales sont les suivantes.

A chaque instruction correspond un mnémonique de trois ou quatre lettres.

Exemple: L'addition s'écrit par exemple ADDA, ou bien ADCB, la soustraction s'écrit par exemple SUBA, ou bien SBCB, un appel de sous-programme s'écrit JSR.

Exemple: ADDA 64ØØØ veut dire additionner le contenu de la case-mémoire d'adresse 64ØØØ à l'Accumulateur A.

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

Exemple: A = Accumulateur
X = Registre d'index X

Il est bien sûr toujours possible de traduire un programme écrit en assembleur à la main en remplaçant les mnémoniques et les opérandes par les nombres hexadécimaux ou même les mots binaires correspondants mais le plus simple est d'avoir un programme qui effectue cette tâche automatiquement sans jamais faire d'erreur : c'est ce programme qui est appelé l'ASSEMBLEUR.

Le jeu d'instructions étant différent d'un type de microprocesseur à un autre, un ASSEMBLEUR sera spécifique à un microprocesseur bien particulier ; nous parlerons donc de l'ASSEMBLEUR 68Ø9.

Son rôle est de traduire le programme appelé source écrit en mnémoniques 68Ø9 en un programme directement exécutable par le microprocesseur que l'on appelle le code-objet.

L'assembleur peut accomplir diverses autres tâches que nous détaillerons plus loin dans cet ouvrage.

Nous venons de voir quelques-uns des avantages d'un assembleur mais voyons un peu les inconvénients qu'il présente.

Le problème est, nous l'avons déjà dit, que chaque microprocesseur possède son propre jeu d'instructions et donc un langage assembleur spécifique. Il n'est donc pas possible de faire tourner des programmes écrits pour le 68Ø9 sur un autre microprocesseur.

Ceci est un problème très important étant donné les coûts énormes de développement du logiciel comparés à ceux du matériel qui ne cessent de décroître.

D'autre part il faut bien avouer, vous avez dû le sentir en regardant le listing que nous avons donné un peu plus haut, qu'un programme en assembleur n'est pas aisément compréhensible à première vue.

Il faut en général se pencher de près sur le problème et quelquefois même cela ne suffit pas.

C'est pourquoi ont été développés des langages de haut niveau, (Basic, Pascal, Fortran, etc.), souvent orientés vers un certain domaine d'application et qui proposent des fonctions aisément compréhensibles, même à première vue.

Tous ces langages sont bien sûr des programmes qui ont été développés à partir d'un assembleur.

Dans les lignes ci-dessus que, nous l'espérons, vous n'avez pas trouvées trop longues, nous avons essayé de vous faire sentir la nécessité de posséder un assembleur, ceci dès que l'on veut sortir un peu des sentiers battus du BASIC qui, bien qu'également très intéressant, n'offre pas la même richesse que l'assembleur.

Dans le prochain chapitre, qui peut être considéré comme une annexe, nous allons décrire les différents systèmes numériques utilisés dans le 68Ø9 et dans les assembleurs destinés à ce microprocesseur.

2

Les systèmes numériques

2.1. LE BINAIRE

Nous l'avons déjà dit, les microprocesseurs ne comprennent que ce mode 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 × 1ØØØ) + (7 × 1ØØ) + (8 × 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 on aura :

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

ou encore :

3783 = (1 × 2^{11}) + (1 × 2^{10}) + (1 × 2^{9}) + (Ø × 2^{8}) + (1 × 2^{7}) + (1 × 2^{6}) +
(Ø × 2^{5}) + (Ø × 2^{4}) + (Ø × 2^{3}) + (1 × 2^{2}) + (1 × 2^{1}) + (1 × 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.

Par un petit calcul simple on pourrait montrer qu'avec 12 bits il est possible de coder des nombres compris entre Ø et 4Ø95 (=111111111111) c'est-à-dire 4Ø96 nombres différents.

Plus généralement un mot binaire de N bits permet de coder 2^N nombres qui sont compris entre Ø et 2^N-1.

Nous avons vu dans le chapitre précédent que les codes-opérations du 68Ø9 étaient donnés sous forme d'un ou deux octets. Il peut donc exister au maximum 2^{16} = 65535 codes-opérations différents ; en fait tous les codes ne sont pas attribués et le jeu d'instruction du 68Ø9 est plus réduit.

Mais, nous direz-vous, pourquoi avoir choisi le binaire ? C'est bien simple : en raison de sa grande facilité de mise en œuvre.

Il est beaucoup plus facile de réaliser des circuits pouvant prendre deux états différents (état "Ø" ou "1", ØV ou 5V, "OUI" ou "NON") même s'ils doivent être en grand nombre (il faut huit chiffres binaires pour coder les nombres compris entre Ø et 255), que de réaliser des circuits pouvant prendre 1Ø états différents (cas du décimal).

2.2. L'HÉXADÉCIMAL

Nous avons jusqu'à présent parlé des systèmes de base 1Ø (le décimal) et de base 2 (le binaire). Pourquoi n'existerait-il pas de système de

base 16 d'autant plus que, nous allons le voir, un tel système facilite grandement l'écriture des octets (en se plaçant cette fois-ci du côté de l'utilisateur et non pas du côté de la machine). De même que le décimal comporte 1Ø chiffres (Ø à 9), l'hexadécimal comporte 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

Voyons maintenant comment il est possible de convertir un mot binaire en un nombre hexadécimal et inversement.

C'est en fait extrêmement simple : le mot binaire est séparé, de la droite vers la gauche, en groupes de 4 bits, chacun de ces derniers étant converti en son équivalent hexadécimal.

Considérons tout d'abord l'octet suivant :

$$\underbrace{\text{Ø11Ø}}_{6} \quad \underbrace{\text{1ØØ1}}_{9}$$

donc Ø11Ø1ØØ1 = 69 en hexadécimal = 1Ø5 en décimal.

Supposons maintenant que le mot binaire ne contienne pas un nombre entier de fois 4 bits.

Soit par exemple le mot suivant :

(ØØ)11 ØØ1Ø Ø11Ø 11Ø1
3 2 6 D

On opère maintenant exactement de la même façon que précédemment en remplaçant les bits manquant à gauche par des zéros.

Cela nous donne ici :

326D en hexadécimal

Grâce à cette notation, un mot-mémoire est codé à l'aide de deux caractères. Une adresse de 16 bits nécessitera de même quatre caractères hexadécimaux.

La plupart des codes-opérations sont compris entre ØØ et FF.

2.3. 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 :

$$36 = \underbrace{\emptyset\emptyset 11}_{3} \quad \underbrace{\emptyset 11\emptyset}_{6}$$

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

Ce code n'en reste pas moins très utilisé surtout dans le 68Ø9 qui permet d'effectuer des additions et des soustractions sur des nombres codés en BCD.

2.4. LE CODE ASCII

Jusqu'à présent nous avons vu différents moyens de codage des nombres. Mais il est nécessaire de savoir également comment coder les caractères alphanumériques (par exemple lors de leur entrée à partir d'un clavier).

C'est dans ce but qu'a été créé le code ASCII (en anglais "American Standard Code for Information Interchange") qui est actuellement le plus répandu. Il s'agit d'un code à 8 bits mais dont le bit de poids fort est utilisé pour la détection des erreurs : c'est le bit de parité. L'information n'est donc réellement contenue que dans 7 bits ce qui permet de coder 128 caractères différents.

Ces 128 codes permettront de représenter toutes les lettres de l'alphabet, les chiffres, ainsi que différents caractères spéciaux. Les codes restant sont utilisés comme des commandes.

Nous donnons ci-après la liste des différents codes ASCII ainsi que leur signification (nous avons supposé que le bit de parité était toujours à zéro pour simplifier).

Code	Caractère	Code	Caractère	Code	Caractère
ØØ	NUL : Null	3Ø	Ø	6Ø	-
Ø1	SOH : Start of Heading	31	1	61	a
Ø2	STX : Start of Text	32	2	62	b
Ø3	ETX : End of Text	33	3	63	c
Ø4	EOT : End of Transmission	34	4	64	d
Ø5	ENQ : Enquiry	35	5	65	e
Ø6	ACK : Acknowledge	36	6	66	f
Ø7	BEL : Bell	37	7	67	g
Ø8	BS : Backspace	38	8	68	h
Ø9	HT : Horizontal Tabulation	39	9	69	i
ØA	LF : Line Feed	3A	:	6A	j
ØB	VT : Vertical Tabulation	3B	;	6B	k
ØC	FF : Form Feed	3C	<	6C	l
ØD	CR : Carriage Return	3D	=	6D	m
ØE	SO : Shift out	3E	>	6E	n
ØF	SI : Shift in	3F	?	6F	o
1Ø	DLE : Data link Escape	4Ø	@	7Ø	p
11	DC1 : Device Control 1	41	A	71	q
12	DC2 : " 2	42	B	72	r
13	DC3 : " 3	43	C	73	s
14	DC4 : " 4	44	D	74	t
15	NAK : Negative Acknowledge	45	E	75	u
16	SYN : Synchronous Idle	46	F	76	v
17	ETB : End of Transmission Block	47	G	77	w
18	CAN : Cancel	48	H	78	x
19	EM : End of Medium	49	I	79	y
1A	SUB : Substitute	4A	J	7A	z
1B	ESC : Escape	4B	K	7B	{
1C	FS : File Separator	4C	L	7C	¦
1D	GS : Group Separator	4D	M	7D	}
1E	RS : Record Separator	4E	N	7E	~
1F	US : Unit Separator	4F	O	7F	DEL : Delete
2Ø	SP : Space	5Ø	P		
21	!	51	Q		
22	"	52	R		
23	#	53	S		
24	$	54	T		
25	%	55	U		
26	&	56	V		
27	'	57	W		
28	(	58	X		
29	)	59	Y		
2A	*	5A	Z		
2B	+	5B	[		
2C	,	5C	\		
2D	-	5D	]		
2E	.	5E	Λ		
2F	/	5F	←		

2.5. 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 ?

Nous avons vu qu'avec un octet il était possible de coder les nombres Ø à 255 en décimal ou ØØ à FF en hexadécimal.

Considérons l'opération Ø–1 = –1 en décimal (elle consiste à retrancher 1 de Ø) et tentons de la réaliser en binaire :

```
   Ø      ØØØØØØØØ
 – 1    – ØØØØØØØ1
 ----   -----------
 = –1   = 11111111 = FF
```

Donc –1 sera représenté par FF.

Recommençons et soustrayons 1 à –1 :

```
 – 1      11111111
 – 1    – ØØØØØØØ1
 ----   -----------
 = –2   = 1111111Ø = FE
```

Donc :

–2 = FE en hexadécimal

Ceci dit 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 de −2. On a:

```
 2 = ØØØØØØ1Ø
     111111Ø1 par inversion des bits
-2 = 1111111Ø = FE
```

Le résultat trouvé est bien le même que précédemment. Nous pouvons vérifier que l'opération +2 −2 donne bien zéro (sur 8 bits). Le tableau suivant donne la valeur décimale associée à chacun des octets compris entre ØØ et FF.

	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

Par cette méthode il est donc possible de coder les nombres décimaux compris entre −128 et +127. Le bit de poids fort (bit 7) de chaque octet est:

— égal à Ø si le nombre est positif,

— égal à 1 si le nombre est négatif.

Notons que cette notation en complément à deux n'est pas obligatoire. C'est au programmeur de décider si les nombres qu'il utilise sont compris entre Ø et 255 ou bien entre −128 et +127.

Dans le prochain chapitre nous rentrons cette fois-ci dans le vif du sujet en décrivant le microprocesseur qui nous intéresse ici: le 68Ø9, ou plutôt son assembleur.

3

La syntaxe assembleur 68Ø9

3.1. INTRODUCTION

Eh oui, comme toute "langue vivante", l'assembleur possède ses propres règles de "grammaire" et d'"orthographe". Et gare à vous si, dans un désir d'indépendance tout à fait mal à propos, vous décidez de ne pas vous y plier !

Ceci dit, ces règles sont très simples et il ne vous faudra qu'un peu d'habitude pour les connaître. Mieux même, ces programmes que jusqu'à maintenant vous considériez être du chinois en seront réduits à n'être finalement que... de l'anglais : on tombe vraiment dans la facilité ! Nous pourrons distinguer deux sortes de règles dans la programmation en assembleur :

— *les règles absolues* : si par malheur vous les transgressez, vous vous ferez froidement "jeter" par la machine avec en guise de compliment de petits mots d'amitié du type "ILLEGAL FORMAT", "MISSING OPERAND", le tout dans le plus pur anglais de Shakespeare ;

— *les règles conseillées* : ce sont celles qui sont laissées à l'appréciation du client. En gros, si vous ne les suivez pas, vous risquez :

* d'une part de passer beaucoup plus de temps que nécessaire pour faire tourner votre programme,

* d'autre part de devoir vous reporter au chapitre consacré aux codes d'erreurs par suite d'une étourderie ce qui est très désagréable.

Nous allons maintenant passer en revue les différentes règles à observer dans le cas d'un assembleur 68Ø9 classique et nous indiquerons au passage ce qu'il *faut* et ce qu'il *ne faut pas* faire.

3.2. LA SYNTAXE ASSEMBLEUR

Une ligne en langage d'assemblage est subdivisée en un certain nombre de parties appelées "champs". 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.

3.2.1. Le champ étiquette (Label)

Vous qui connaissez le BASIC, vous devez savoir ce qu'est une étiquette : celle-ci représente une adresse de branchement.

Exemple :

GO TO 1ØØ

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 (souvent 6) dont le premier doit être une lettre.

Exemple :

BOUCLE
BRANCH
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 #3Ø      ; EFFECTUE LA SOMME
\_____/
 champ
 label
```

Ce qu'il faut faire :

Les noms donnés aux étiquettes *doivent* (c'est un conseil d'ami) être des noms compréhensibles au commun des mortels. Cela permettra à d'autres que vous-même de lire vos programmes ; et puis imaginez que vous vouliez améliorer le programme que vous avez écrit l'année précédente si vous faites...

Ce qu'il ne faut pas faire :

Utiliser les "%", "*" et autres "$" pour définir une étiquette. D'autre part l'assembleur risque de ne pas être content et de vous lancer un "ILLEGAL LABEL" en guise de reproche. D'autre part avouez que de tels signes (bien qu'utilisés très largement en micro-informatique) ne sont pas très explicites.

3.2.2. Le champ opération

Le champ opération est situé juste après le champ étiquette duquel il est séparé par un symbole spécial appelé DELIMITEUR. Le plus commun est tout simplement un espace (touche "SPACE"). Dans ce champ on trouve le mnémonique de l'instruction.

```
SOMME          ADDA #3Ø        ; EFFECTUE LA SOMME
     \________/\______/
       champ     champ
    délimiteur opération
```

3.2.3. Le champ opérande

Ce champ est celui qui risque de poser le plus de problèmes au programmeur novice en assembleur. Il est généralement séparé du champ opération par un délimiteur.

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.

Nous reparlerons de ceci dans le prochain chapitre qui sera en partie consacré aux différents modes d'adressage du 68Ø9.

Dans le champ opérande on peut trouver :

— des nombres,
— des noms de variables,
— des étiquettes,
— des expressions arithmétiques ou logiques.

a) Les nombres

La plupart des assembleurs acceptent des nombres sous forme décimale, hexadécimale, binaire, le tout étant de le préciser.

Il existe donc des caractères qui permettent de renseigner l'assembleur sur le système numérique dans lequel ils sont représentés.

— *les nombres décimaux.* En général le nombre 8Ø (en base 1Ø) s'écrira 8Ø. Il n'y a pas besoin ici de caractère spécial (on appelle cela l'option de défaut). Les nombres décimaux pourront être positifs ou négatifs.

Exemple : –3
+75
Ø

— *les nombres hexadécimaux.* Le caractère utilisé est généralement un "**$**" (dollar), parfois un "H" comme hexadécimal.

Exemple : $8Ø (=128)

ou bien : 8ØH

— *les nombres binaires.* Les caractères utilisés peuvent être le "%" ou bien "B". L'octet Ø1ØØ11Ø1 s'écrira alors :

ou bien : %Ø1ØØ11Ø1
Ø1ØØ11Ø1B

— *les caractères ASCII.* Il s'agit d'un unique caractère ASCII précédé d'une apostrophe.

Exemple :

'A

L'assembleur remplace ce caractère par le code ASCII correspondant.

— *les déplacements par rapport au registre PC* (compteur ordinal dont nous parlerons plus loin).

Un déplacement de 7 octets par rapport au PC s'écrira :

*+7

Le déplacement peut être positif ou négatif.

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'étiquettes

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

Exemple :

JMP BOUCLE

d) Les expressions arithmétiques et logiques

Certains assembleurs acceptent comme opérande des expressions arithmétiques et logiques mettant en jeu des nombres, des noms de variables, des noms d'étiquettes, etc...

Exemple : ADCA VALEUR+1
JMP BOUCLE−5

Ces expressions arithmétiques utilisent les opérateurs + (addition), − (soustraction), * (multiplication), / (division entière).

Attention : Ce type d'expressions est une grande source d'erreurs surtout lorsqu'elles sont compliquées, donc essayez de les utiliser le moins fréquemment possible.

3.2.4. 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 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 généralement 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 CARRY", "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 petit 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.

3.2.5. Les pseudo-instructions

Nous appelons pseudo-instructions les instructions qui ne font pas partie de la bibliothèque du 68Ø9 et qui sont juste destinées à donner des informations (ou directives) à l'assembleur.

Nous allons en répertorier quelques-unes sachant qu'elles peuvent différer d'un assembleur à l'autre.

a) La directive origine

Elle permet de définir l'adresse de départ d'un programme ou d'un sous-programme.

Supposons que nous voulions faire commencer notre programme à l'adresse $Ø2ØØ, nous écrirons alors :

ORG $Ø2ØØ

Après assemblage l'adresse de la première instruction sera donc $Ø2ØØ.

b) La directive de fin

De même qu'il est nécessaire de fournir à l'assembleur une indication concernant l'adresse de début du programme, il faut qu'il connaisse également l'endroit où il se termine. C'est le rôle de la directive END

notée généralement END nn. Elle se place à la dernière ligne du programme.

Prenons par exemple le petit programme suivant :

```
DEBUT  LDA    $78
       ADCA   # Ø5
–
–
       END    DEBUT
```

Lors de l'assemblage, la pseudo-instruction "END DEBUT" indique à l'assembleur que la première instruction à exécuter sera située à l'adresse DEBUT. Alors que la directive ORG donne l'adresse de chargement mémoire du code-objet, la directive END donne l'adresse de lancement du programme (ces deux adresses ne sont pas forcément égales). La définition du code-objet sera donnée un peu plus loin.

c) La directive Equate

Nous avons vu précédemment que l'on pouvait donner des noms à des variables représentant soit des adresses soit des données. Afin que l'assembleur puisse générer le code-objet, il faut préalablement définir ces variables.

Suivant l'assembleur utilisé on écrira :

```
COMPT  EQU $Ø5
FIN    EQU DEBUT+$6Ø
```

Cette dernière ligne suppose bien sûr que la variable DEBUT ait été préalablement définie.

En général, on placera les directives "EQU" au début du programme. Cela accroît la lisibilité et surtout cela permet de les changer facilement.

d) 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, des tableaux, des chaînes de caractères ASCII. Nous allons définir ici un certain nombre de directives qui remplissent ces rôles.

— *la directive FCB*. Elle permet de réserver un octet en mémoire.

Exemple:

VAL FCB $3Ø

Lorsque l'assembleur rencontre cette ligne, il place la valeur $3Ø dans la première case-mémoire et lui assigne la variable VAL.

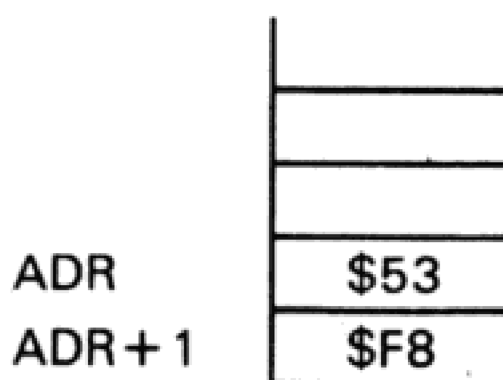
Le pointeur d'adresse est alors incrémenté de 1.

— *la directive FDB*. Elle permet de réserver un double octet en mémoire. Ici le pointeur d'adresse est incrémenté de 2 et l'octet de poids fort du mot de 16 bits est stocké en premier.

Exemple:

ADR FDB $53F8

L'état de la mémoire sera le suivant:



La variable ADR est assignée à $53F8.

— *la directive FCC*. Cette directive permet de réserver un bloc mémoire pour stocker une chaîne de caractères ASCII.

Exemple:

DONNEE FCC /NUMERO?/

Les caractères "N", "U", "M", "E", "R", "O", "?" sont stockés en mémoire grâce à leurs codes ASCII.

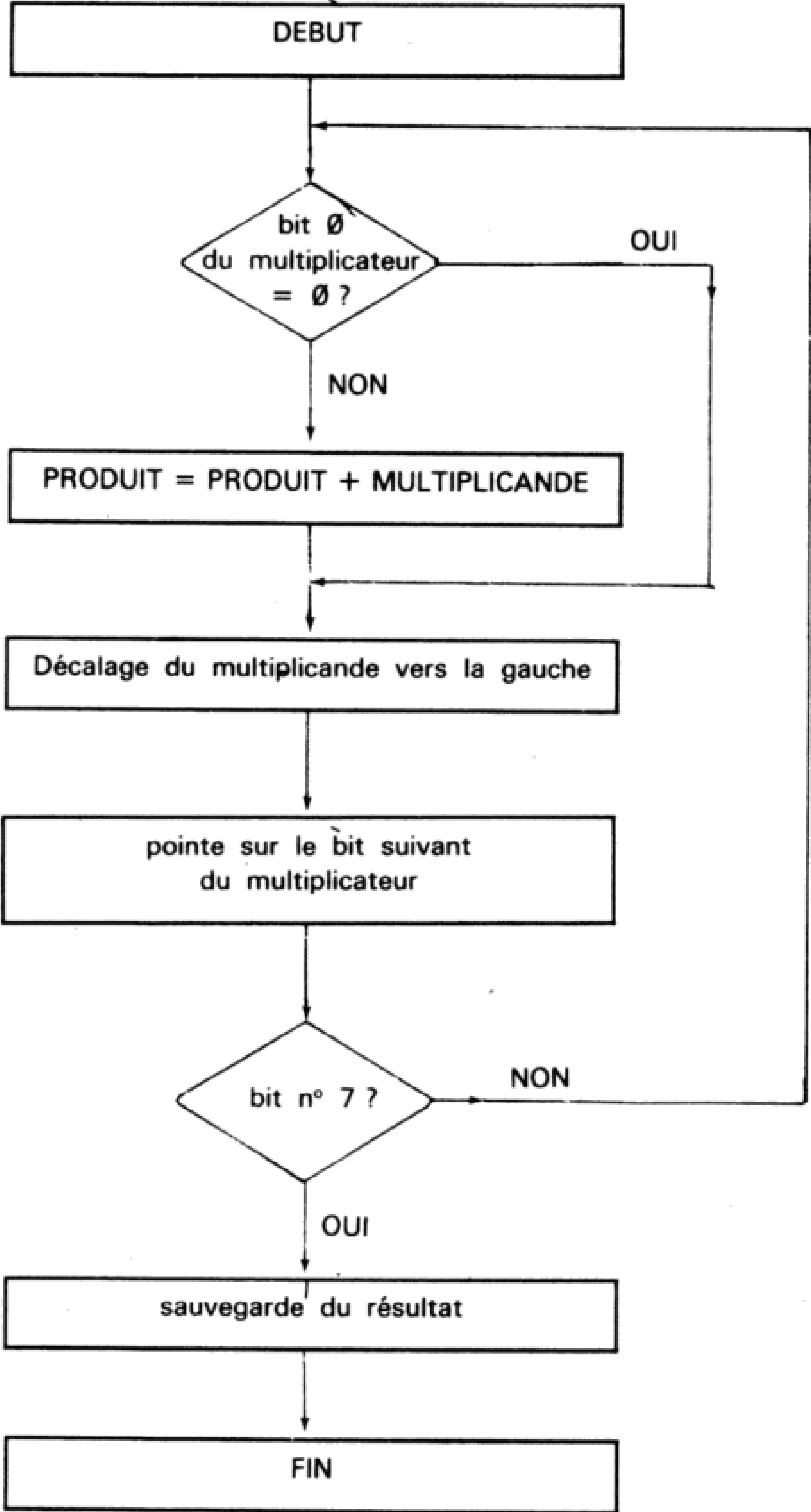
La variable DONNEE est assignée à l'adresse du premier caractère, soit "N".

Nous allons nous arrêter là dans notre énumération. Nous vous donnons ci-après le listing d'un programme source (donc non assemblé) dans lequel nous avons employé volontairement le maximum de pseudo-instructions. Il est à noter que nous avons indiqué jusqu'à présent les notations les plus courantes que l'on trouve dans les assembleurs 68Ø9. Sur certaines machines la syntaxe peut différer mais les principes de fonctionnement restent les mêmes. Pour plus de précisions vous pourrez vous reporter à la notice d'utilisation fournie avec l'assembleur que vous possédez.

Le programme ci-dessous simule le fonctionnement de l'instruction MUL et exécute donc une multiplication binaire.

```
       ORG    $Ø2ØØ
PRODB  EQU    $ØØ     ;PARTIE BASSE RESULTAT
PRODH  EQU    $Ø1     ;PARTIE HAUTE RESULTAT
VAR    EQU    $Ø2     ;INTERMEDIAIRE CALCUL
MDE    EQU    $Ø5     ;MULTIPLICANDE
MTR    FCB    $12     ;MULTIPLICATEUR
DEBUT  LDB    #$Ø8    ;INITIALISE BOUCLE
       LDA    #$ØØ    ;ANNULE RESULTAT
       TFR    A,DP    ;REGISTRE PAGE DIRECTE
       STA    <PRODB  ;PARTIE BASSE RESULTAT
       STA    <PRODH  ;PARTIE HAUTE RESULTAT
       STA    <VAR    ;
DECAL  LSR    MTR     ;BITØ =Ø
       BCC    SUITE   ;OUI,DECALE MULTIPLICANDE
       LDA    <PRODB  ;NON,ADD.  MULTIPLICANDE
       ADDA   <MDE    ;PARTIE BASSE
       STA    <PRODB  ;SAUVEGARDE PARTIE BASSE
       LDA    <PRODH  ;PARTIE HAUTE
       ADCA   <VAR    ;
       STA    <PRODH  ;SAUVEGARDE PARTIE HAUTE
SUITE  ASL    <MDE    ;DECALE MULTIPLICANDE
       ROL    <VAR    ;SAUVE BIT 7 DANS VAR
       DECB           ;DERNIERE ITERATION?
       BNE    DECAL   ;NON, RECOMMENCE
       SWI            ;OUI, TERMINE
       END
```

L'organigramme est le suivant :



3.3. L'ASSEMBLAGE

3.3.1. Introduction

Nous venons de voir la structure d'un programme en assembleur après son introduction à partir du clavier. Ce programme (appelé programme-source) n'est, bien entendu, pas exécutable par le microprocesseur. Le rôle de l'assembleur est de générer à partir de ce code-source un code-objet exécutable par la machine.

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 et d'autre part les adresses d'implantation en mémoire des codes-opérations ;

— il détecte les erreurs éventuelles (erreurs de syntaxe, sur des variables, des étiquettes, etc.), que nous détaillerons un peu plus loin.

b) L'assembleur génère un code-objet destiné à être stocké sur cassette ou disquette et qui contient, outre le code-machine correspondant au programme, des informations concernant son adresse d'implantation en mémoire, son adresse de lancement.

En ce qui concerne notre exemple, l'allure du programme assemblé est la suivante :

```
                0010            ORG     $0200
                0020    ;
                0030    PRODB   EQU     $00     ;PARTIE BASSE RESULTAT
                0040    PRODH   EQU     $01     ;PARTIE HAUTE RESULTAT
                0050    VAR     EQU     $02     ;INTERMEDIAIRE CALCUL
                0060    MDE     EQU     $05     ;MULTIPLICANDE
0200 12         0070    MTR     FCB     $12     ;MULTIPLICATEUR
                0080    ;
                0090    ;
0201 C6 08      0100    DEBUT   LDB     #$08    ;INITIALISE BOUCLE
0203 86 00      0110            LDA     #$00    ;ANNULE RESULTAT
0205 1F 8B      0120            TFR     A,DP    ;REGISTRE PAGE DIRECTE
0207 97 00      0130            STA     <PRODB  ;PARTIE BASSE RESULTAT
0209 97 01      0140            STA     <PRODH  ;PARTIE HAUTE RESULTAT
020B 97 02      0150            STA     <VAR    ;
020D 74 00 02   0160    DECAL   LSR     MTR     ;BIT0 =0
0210 24 0C      0170            BCC     SUITE   ;OUI,DECALE MULTIPLICANDE
0212 96 00      0180            LDA     <PRODB  ;NON,ADD.  MULTIPLICANDE
0214 9B 05      0190            ADDA    <MDE    ;PARTIE BASSE
0216 97 00      0200            STA     <PRODB  ;SAUVEGARDE PARTIE BASSE
0218 96 01      0210            LDA     <PRODH  ;PARTIE HAUTE
021A 99 02      0220            ADCA    <VAR    ;
021C 97 01      0230            STA     <PRODH  ;SAUVEGARDE PARTIE HAUTE
021E 08 05      0240    SUITE   ASL     <MDE    ;DECALE MULTIPLICANDE
0220 09 02      0250            ROL     <VAR    ;SAUVE BIT 7 DANS VAR
0222 5A         0260            DECB            ;DERNIERE ITERATION?
0223 26 E8      0270            BNE     DECAL   ;NON,RECOMMENCE
0225 3F         0280            SWI             ;OUI,TERMINE
0226            0290            END

SYMBOL TABLE:
PRODB=0000      PRODH=0001      VAR=0002
MDE=0005        MTR=0200        DEBUT=0201
DECAL=020D      SUITE=021E
```

3.3.2. L'assemblage

Nous allons tout d'abord décrire brièvement le fonctionnement d'un assembleur. L'assemblage s'effectue généralement en deux fois c'est-à-dire en deux lectures du programme que l'on appellera "passes".

Il faut tout d'abord dire qu'une ligne assembleur comprend des symboles connus (mnémoniques) et des symboles inconnus (noms de variables et d'étiquettes).

Durant la première passe, l'assembleur examine d'une part chaque instruction dont il détermine la longueur (1 octet ou plus) d'autre part

chaque symbole qu'il stocke 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 recherche dans la table et lui affecte l'adresse ou la donnée stockée lors de la première passe.

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

a) Les erreurs de syntaxe

Nous nous contenterons de donner quelques exemples :

— Si l'assembleur rencontre un mnémonique qui n'existe pas dans la bibliothèque du 68Ø9 il générera probablement un message du type "UNDEFINED OPERATION CODE".

— De même, si l'opérande n'est pas correcte vous vous ferez gratifier d'un "ILLEGAL FORMAT".

Ces messages d'erreur sont donnés à titre indicatif et peuvent varier d'un assembleur à l'autre. Très souvent, une lettre est affectée à chaque type d'erreur et placée dans la marge en face de la ligne erronée. Il faudra donc vous plonger dans le manuel d'utilisation de votre assembleur.

b) Les erreurs de second niveau

Celles-ci sont généralement plus difficiles à corriger. Ce sont par exemple :

— Le cas où une étiquette est absente : "MISSING LABEL".

— Le cas où deux valeurs sont assignées au même nombre : "DOUBLE DEFINITION".

— Le cas où une variable utilisée n'a pas été définie préalablement : "UNDEFINED NAME".

c) Les erreurs de conception

Il s'agit des erreurs qui ne font pas partie des deux groupes précédents et donc que l'assembleur ne détectera pas. Elles sont liées à la conception du programme où à la mauvaise traduction de l'organigramme en programme-source. Nous reviendrons sur ce point vers la fin de cet ouvrage dans un chapitre consacré à la mise au point d'un programme assembleur.

3.4. DES ASSEMBLEURS PLUS PERFORMANTS

Jusqu'à présent nous avons toujours considéré le cas d'un assembleur de type classique. Mais il en existe de plus performants.

3.4.1. Les assembleurs-éditeurs

Pratiquement tous les assembleurs disponibles dans le commerce pour des ordinateurs individuels disposent d'un éditeur de texte. En effet, avant d'être assemblé, le programme doit avoir été introduit à partir du clavier et doit donc pouvoir être modifié à volonté. Notons que les éditeurs de texte performants font souvent cruellement défaut dans les interpréteurs BASIC des ordinateurs individuels.

3.4.2. Les macro-assembleurs

Il s'agit d'assembleurs qui, en plus des fonctions classiques que nous avons vues précédemment, donnent la possibilité de définir des ordres MACRO. Supposez que dans un programme certaines lignes ou certains groupes de lignes se répètent un certain nombre de fois : il peut être intéressant d'affecter un nom à ces lignes. Chaque fois que l'assembleur rencontrera ce nom il le remplacera par la séquence d'instructions correspondante. Notons que certains macro-assembleurs puissants

permettent de répéter des séquences d'instructions en modifiant certains paramètres. Il ne faut pas confondre une macro-instruction avec un sous-programme car, contrairement au cas de ce dernier, il n'y a aucun branchement. Les macro-instructions permettent d'alléger l'écriture du programme source uniquement alors que les sous-programmes allègent à la fois le programme-source et le code-objet.

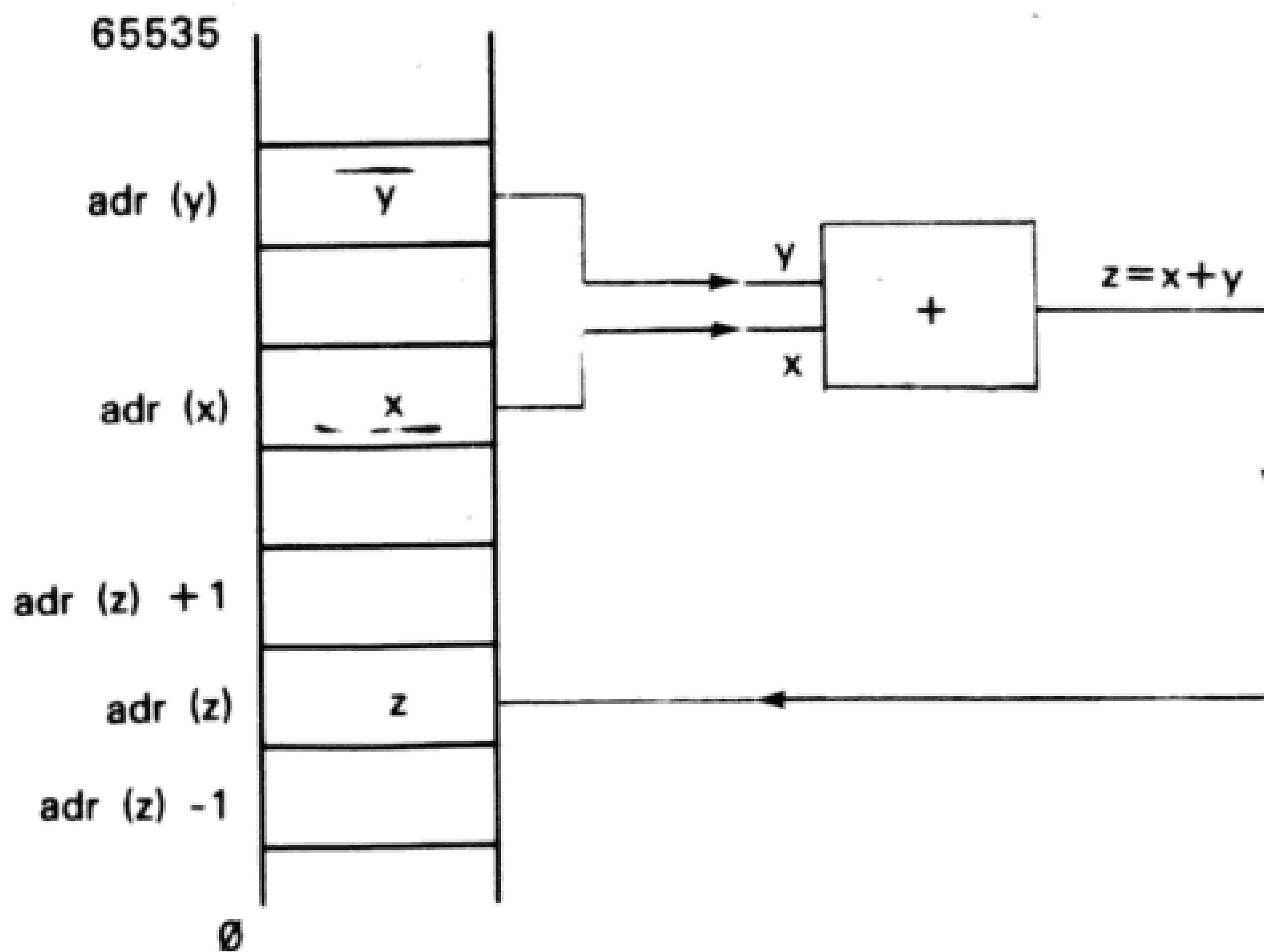
4
Description du 68Ø9

4.1. LES REGISTRES INTERNES DU 68Ø9

4.1.1. Les accumulateurs

Avant de vous présenter les différents registres du 68Ø9 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 :

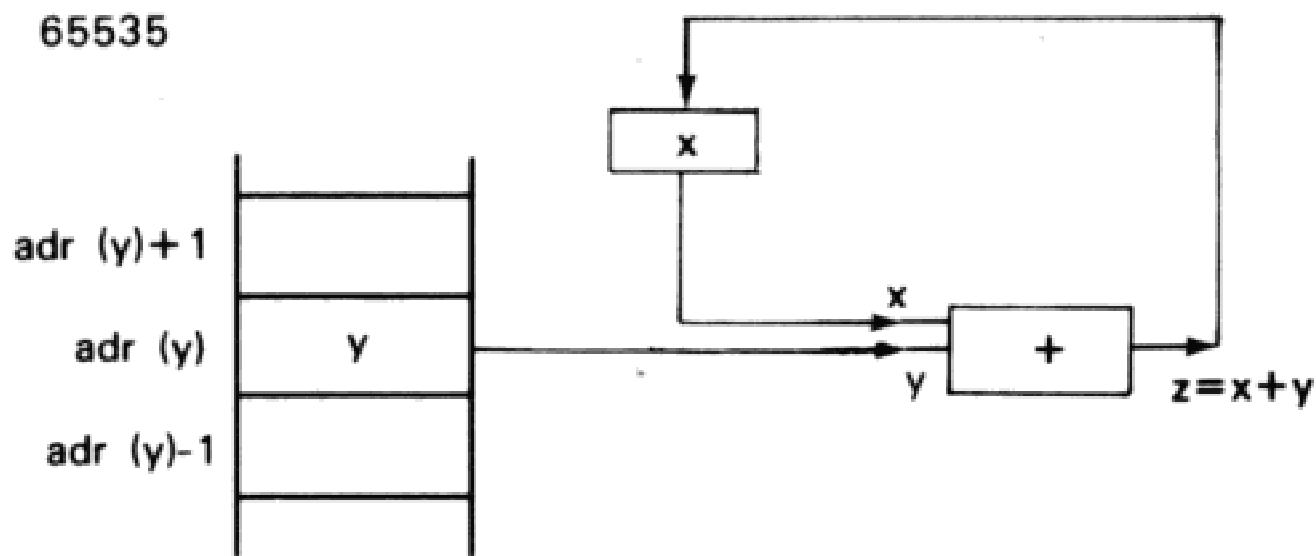


Nous avons déjà dit qu'un microprocesseur classique (et en particulier le 68Ø9) pouvait adresser 64 K octets de mémoire (ou plus exactement $65536 = 2^{16}$ octets). Afin de pouvoir sélectionner une case-mémoire et une seule, on définit une adresse sur 16 bits (ce qui fait donc 2 mots de 8 bits). Nous avons vu d'autre part qu'une instruction (ici addition) pouvait être codée par un mot de 8 bits (en fait plusieurs octets sont souvent nécessaires) 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 68Ø9 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 de la plupart des microprocesseurs existant actuellement sur le marché et notamment du 68Ø9.

Dans son cas il existe précisément deux accumulateurs A et B (8 bits) qui jouent exactement le même rôle. (Il existe des instructions qui travaillent sur ces deux registres.)

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 68Ø9. Il est à noter que dans certains microprocesseurs existant sur le marché, certains registres sont réservés pour un usage particulier (par exemple les accumulateurs sont réservés pour le stockage temporaire de résultats, les registres d'index sont utilisés pour les modes d'adressage comportant une indexation). Ceci permet d'avoir des microprocesseurs souvent un peu plus efficaces (du point de vue rapidité) mais ne facilite pas la tâche des programmeurs qui ont à connaître un jeu d'instructions parfois impressionnant.

Le 68Ø9, au contraire, ne comporte que 59 instructions différentes mais qui, compte tenu des différents registres concernés et modes d'adressage, portent ce nombre à 1 4Ø4.

En effet la plupart des instructions opèrent similairement sur les Accumulateurs ou les autres registres internes du 68Ø9 (par exemple registres d'index, pointeurs de pile, etc...) que nous allons décrire dans les pages à venir. Notons que les deux accumulateurs A et B peuvent être réunis pour former un unique registre de 16 bits. Le 68Ø9 possède d'ailleurs des instructions spécialisées dans le traitement d'informations 16 bits.

4.1.2. Les registres d'index X et Y

A côté des accumulateurs A et B il existe deux registres X et Y appelés registres d'index. Leur première particularité est de posséder 16 bits chacun, contrairement aux Accumulateurs qui n'en possèdent que huit. Ces registres sont accessibles à l'utilisateur au même titre que ces derniers et ils peuvent être mis en jeu directement dans les instructions de chargement, de comparaison par exemple. Bien qu'ils soient destinés plus particulièrement à une utilisation dans le cadre de modes d'adressage indexés (nous reviendrons sur ce point dans le prochain paragraphe de ce chapitre), ils peuvent être considérés comme des registres d'usage général comprenant 16 bits, c'est-à-dire par exemple :

— comme registres de stockage de résultats intermédiaires,

— comme compteurs de boucle.

Prenons par exemple le programme Basic suivant :

```
1Ø FOR I=1 TO 2Ø
2Ø PRINT I
3Ø NEXT I
4Ø END
```

Chaque fois que l'interpréteur Basic passe à la ligne 1Ø, la valeur de I est incrémentée et le programme affiche les valeurs suivantes :

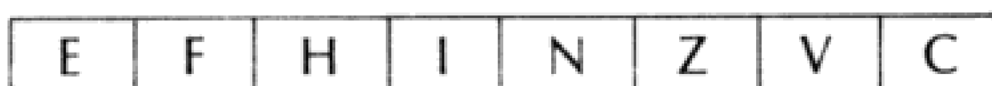
```
1
2
3
.
.
.
2Ø
```

On peut faire de même en assembleur : on charge un registre avec une valeur initiale (1 par exemple) et on l'incrémente à chaque tour de boucle. Le programme peut s'arrêter si l'on compare à chaque fois la valeur du compteur ainsi formé avec une valeur déterminée (2Ø par exemple).

Notons que le 68Ø9 possède des modes d'adressage très utiles pour ce type d'applications et qui sont les adressages auto-incrémentés et auto-décrémentés. Mais nous rentrerons dans les détails dans le prochain chapitre.

4.1.3. Le registre de condition CC

Ce registre est un peu particulier. Il comporte 8 bits qui sont les suivants :



E	F	H	I	N	Z	V	C

E = bit 7 = indicateur de sauvegarde totale du contexte (en anglais ENTIRE FLAG).

F = bit 6 = masque d'interruption rapide (FIRQ).

H = bit 5 = indicateur de demi-retenue (en anglais HALF-CARRY).
I = bit 4 = masque d'interruption (IRQ).
N = bit 3 = indicateur de résultat négatif.
Z = bit 2 = indicateur de zéro.
V = bit 1 = indicateur de débordement (OVERFLOW).
C = bit Ø = 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 (0 ou 1).

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

Le microprocesseur possède des instructions particulières permettant de modifier ou de lire la valeur prise par certains indicateurs du registre de condition CC. D'autre part, certaines instructions du 68Ø9 ont un déroulement qui dépend de la valeur prise par ces indicateurs (instructions de branchement conditionnel).

Nous ne rentrerons pas davantage dans les détails pour le moment car nous pensons que rien ne vaut un exemple pour bien saisir le fonctionnement de ce registre. Nous reviendrons donc sur ce point dans le chapitre consacré au jeu d'instructions du 68Ø9.

4.1.4. Le compteur ordinal : registre PC

Ce nom doit vous sembler pour le moins barbare mais vous allez vous rendre compte bien vite qu'en fait il s'agit de quelque chose de très simple.

Il faut savoir qu'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).

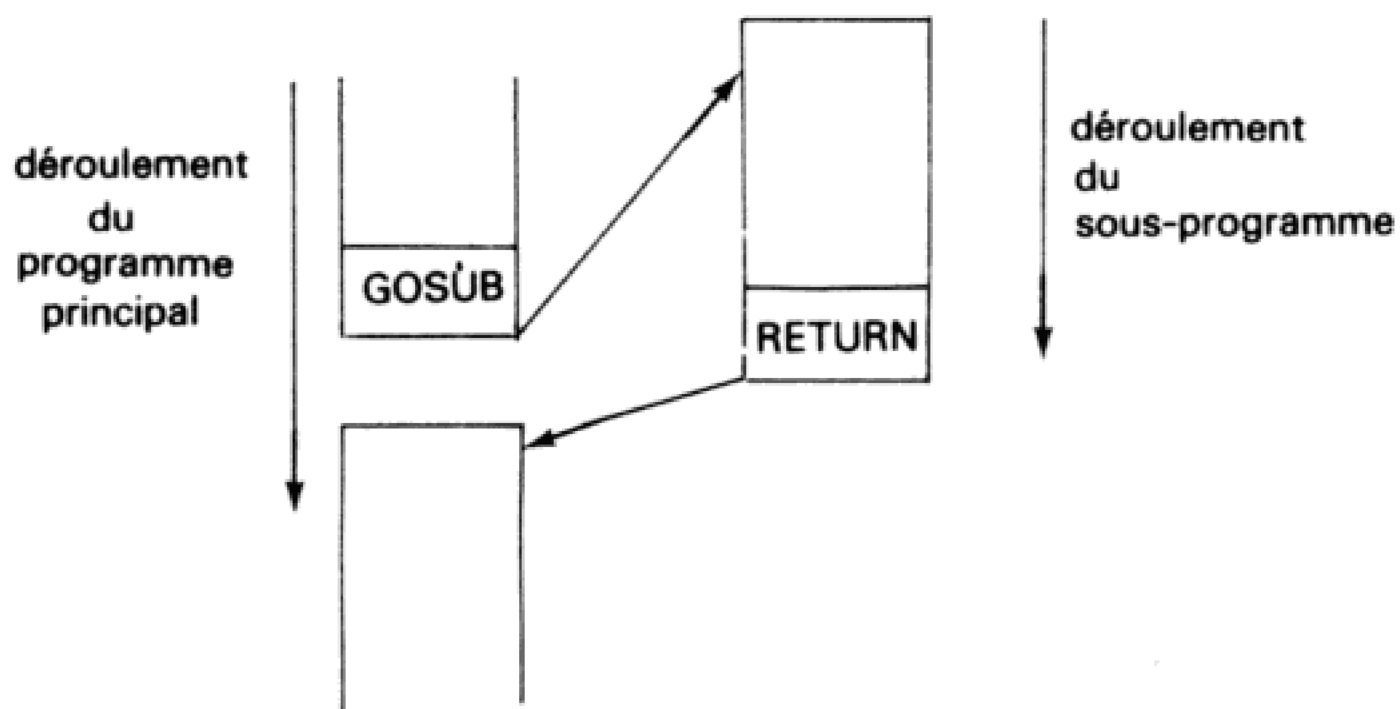
Afin que le programme se déroule correctement, il apparaît comme évident que le microprocesseur, lorsqu'il est en train d'exécuter une instruction, doit connaître l'adresse de la suivante.

Dans ce but il existe dans le microprocesseur un registre spécial appelé 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.

4.1.5. Les pointeurs de pile : registres S et U

a) Notion de pile

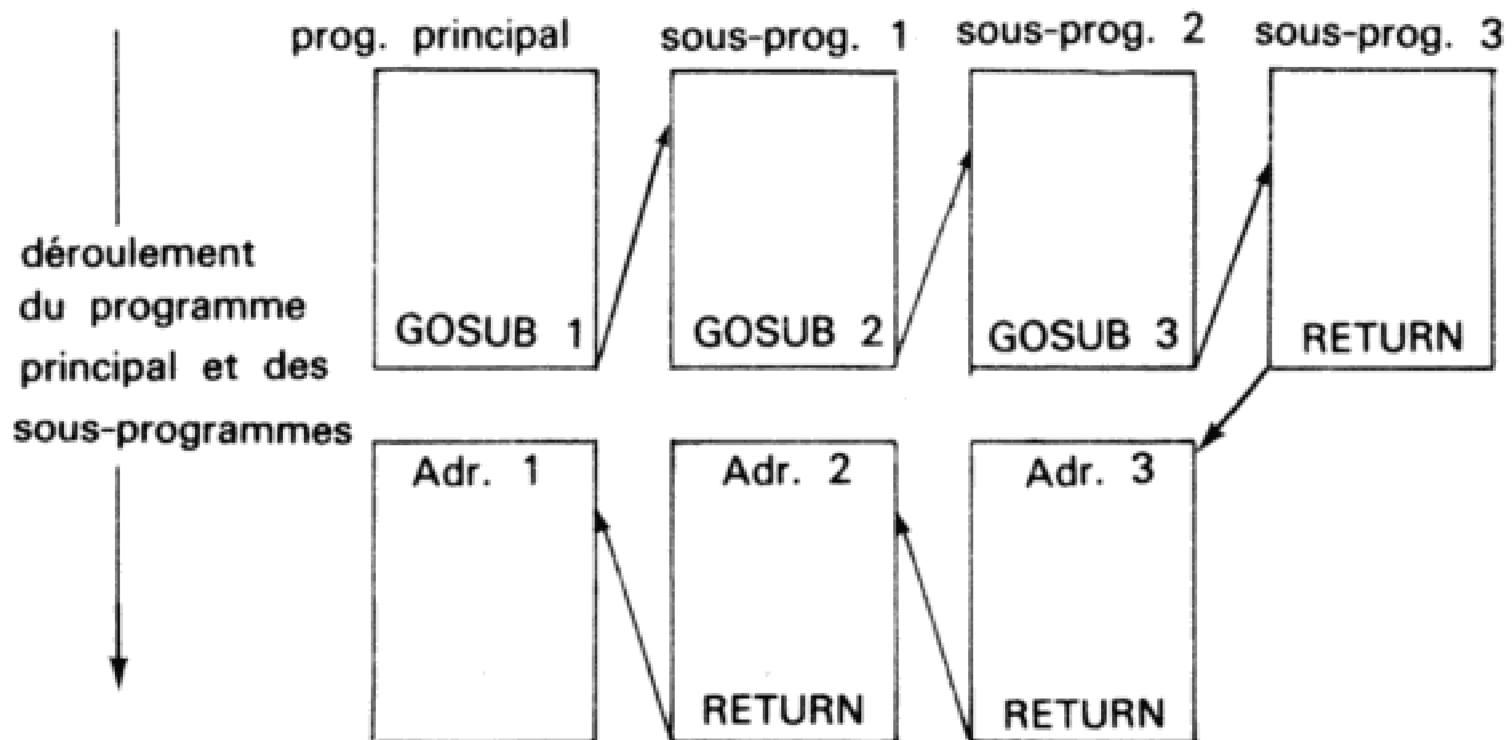
Revenons sur le rôle du compteur ordinal (PC) : nous avons vu qu'il était nécessaire que le microprocesseur sache à tout instant l'adresse de la prochaine instruction à exécuter. Examinons ce qui se passe lorsque le programme principal fait appel à un sous-programme.



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) mais le problème est le suivant : 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 (à moins que l'on ne tienne absolument à faire de la "casse"). Voilà qui nous amène directement à parler de la notion de pile dans un microprocesseur.

Il existe dans le cas du 68Ø9 deux zones mémoire qui fonctionnent selon le principe de la pile décrite précédemment.

Ces deux zones mémoires ont un emplacement en mémoire fixé grâce à deux registres S et U.

Le registre S réfère à la pile système tandis que le registre U réfère à la pile utilisateur.

La première, comme son nom l'indique, est utilisée par le 68Ø9 pour sauvegarder des données nécessaires à son fonctionnement, en particulier les adresses de retour de sous-programme dont nous avons parlé ci-dessus et le contenu de certains registres internes dans le cas d'interruptions. (Les interruptions serons envisagées dans le chapitre consacré au jeu d'instructions.) La deuxième est utilisée uniquement par l'utilisateur qui a ainsi la possibilité de stocker temporairement certains résultats et d'aller les retrouver rapidement, quand les registres internes du 68Ø9 sont déjà tous utilisés.

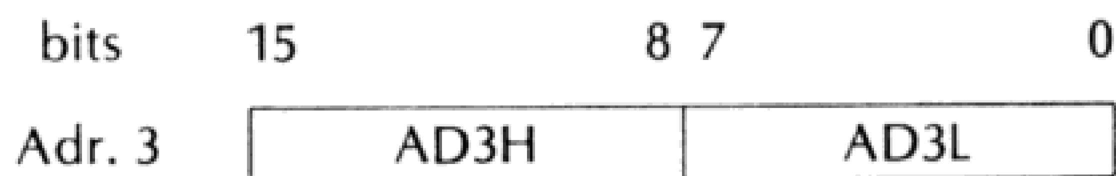
Les deux registres S et U possèdent 16 bits. Leur contenu doit être programmé chaque fois que le 68Ø9 est mis sous tension par l'intermédiaire d'instructions qui apparaîtront au début du programme.

Chacune des deux piles peut donc être située n'importe où dans l'espace adressable du 68Ø9, contrairement au cas de nombreux autres microprocesseurs où la pile (qui est unique) possède 256 octets situés à une place fixe.

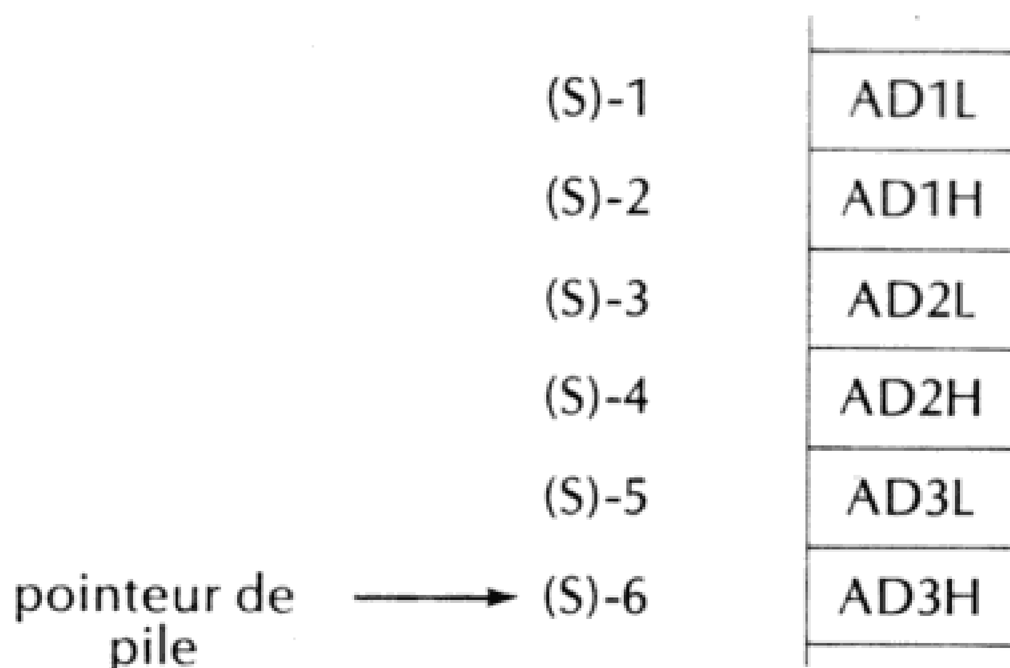
Considérons l'exemple choisi précédemment (appel de sous-programmes). La pile système contiendra après l'appel du sous-programme 3 les octets suivants :

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).



État de la pile système après l'appel du sous-programme 3 :



On remarque que les adresses sont chargées dans la pile à partir du haut. (S) représente le contenu du pointeur de pile système à l'initialisation (si ce contenu était $1FØ7 à l'initialisation, il devient donc $1FØ1 après la sauvegarde de Adr. 3). Lors du stockage d'une adresse de retour de sous-programme les opérations suivantes sont effectuées :

— le pointeur de pile système est décrémenté ;

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

— le pointeur de pile système est à nouveau décrémenté ;

— l'octet de poids fort de l'adresse de retour est chargé à son tour sur la pile.

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 après le chargement (contrairement au cas d'un appel de sous-programme).

Le registre U, relatif à la pile utilisateur fonctionne de la même façon. Lorsque le contenu d'un registre (nous reviendrons sur ce point dans le chapitre consacré au jeu d'instructions du 68Ø9) doit être chargé en haut de la pile, le registre U est d'abord décrémenté d'une unité puis la case-mémoire d'adresse spécifiée par le contenu de U est chargée par la valeur concernée.

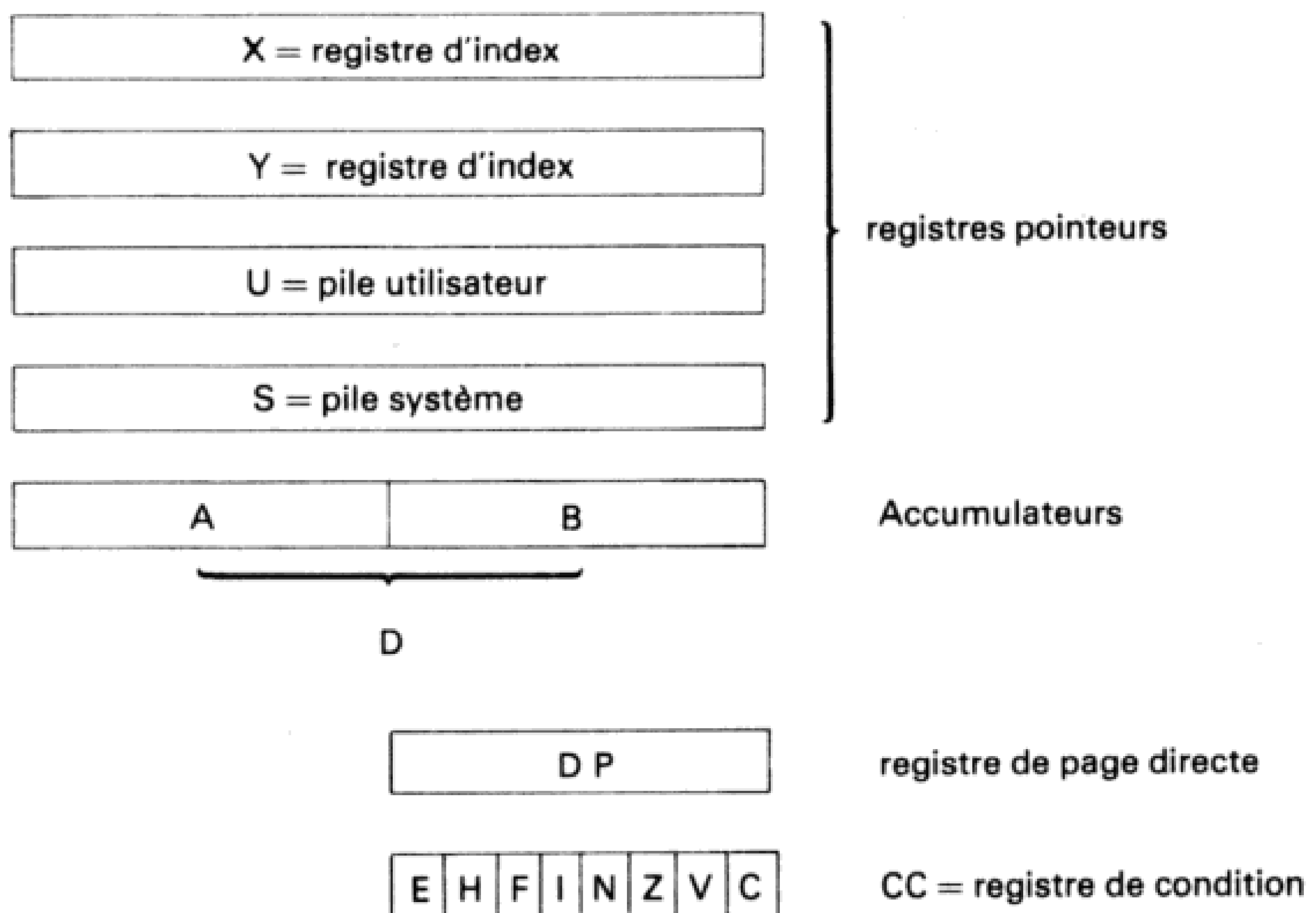
En résumé, les pointeurs de pile U et S pointent sur la dernière donnée stockée dans la pile. Celle-ci fonctionne selon le mode LIFO (en anglais "Last In First Out") qui signifie que la dernière donnée entrée est la première sortie.

4.1.6. Le registre de page directe

Ce registre, comportant 8 bits, est utilisé uniquement avec le mode d'adressage direct qui sera décrit dans le prochain paragraphe. Nous le décrirons donc à cette occasion.

4.1.7. Conclusion

Pour résumer ce paragraphe, les différents registres internes du 68Ø9 sont représentés ci-dessous :



4.2. LES DIFFÉRENTS MODES D'ADRESSAGE DU 68Ø9

Tout d'abord qu'appelle-t-on mode d'adressage ? Nous avons vu au début de ce chapitre (paragraphe 1) un exemple (addition de deux nombres) ou nous disions que pour connaître un nombre x, il fallait spécifier son adresse sur 16 bits (ou deux octets).

Le fait de définir x par la donnée de son adresse constitue un mode d'adressage : nous appellerons ce mode "adressage étendu" (car on peut accéder directement, grâce à lui, à la totalité de l'espace adressable du 68Ø9). Donc, par définition, un mode d'adressage est un moyen d'accéder à une case-mémoire donnée.

4.2.1. 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 68Ø9, sur un ou deux octets. 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 d'échange ou de transfert de registre interne, les opérations sur la pile. Notons que ces deux dernières sortes d'instructions nécessitent deux octets, l'un pour le code-opération proprement dit, l'autre pour sélectionner les registres concernés (registres destination et source pour les instructions de transfert de registre et registres à sauvegarder ou charger dans le cas d'instructions portant sur la pile).

Ce dernier octet se nomme "post-octet". Nous reviendrons sur ce point dans le chapitre traitant du jeu d'instructions du 68Ø9.

Exemples :

```
ASLA
RTI
SWI
TFR A, B
EXG U, S
PSHS A, B, X
```

4.2.2. 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 ou deux octets,

— 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 # $Ø5
```

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 S avec la valeur 16 bits $12F3.

L'instruction sera la suivante :

```
CMPS # $12F3
```

4.2.3. L'adressage étendu

Ce mode d'adressage a déjà été rencontré dans ce chapitre. 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
```

Grâce à cet adressage on peut (nous l'avons déjà dit) accéder à la totalité de l'espace-mémoire du 68Ø9.

Exemple 2 : Soit à charger le registre S avec le contenu des cases-mémoire d'adresses $53F8 et $53F9.

On écrira :

```
LDS $53F8
```

4.2.4. 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. Dans beaucoup de microprocesseurs (et notamment dans le cas du 65Ø2) cet espace est situé en page-zéro qui est la première page de 256 octets adressable par un microprocesseur (adresses comprises entre Ø et 255). Dans le 68Ø9 il existe un registre spécial DP (registre de page directe : "direct-page register") qui permet de spécifier quelle page est utilisée dans ce mode d'adressage. Cela permet donc d'adresser les 64 K octets du 68Ø9 tout en garantissant une rapidité d'exécution du code due au fait que l'opérande n'est plus codée que sur un octet.

Le contenu du registre DP fixe donc l'octet de poids fort de l'adresse 16 bits tandis que l'opérande de l'instruction considérée en fournit l'octet de poids faible.

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

On écrira :

```
LDD < $12        avec :  DP = $ØØ
```

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

Notons qu'après une mise sous tension du 68Ø9 le contenu du registre DP est toujours à zéro. Si donc l'utilisateur ne désire s'intéresser qu'à la page zéro, il lui suffira de ne pas s'occuper de ce registre.

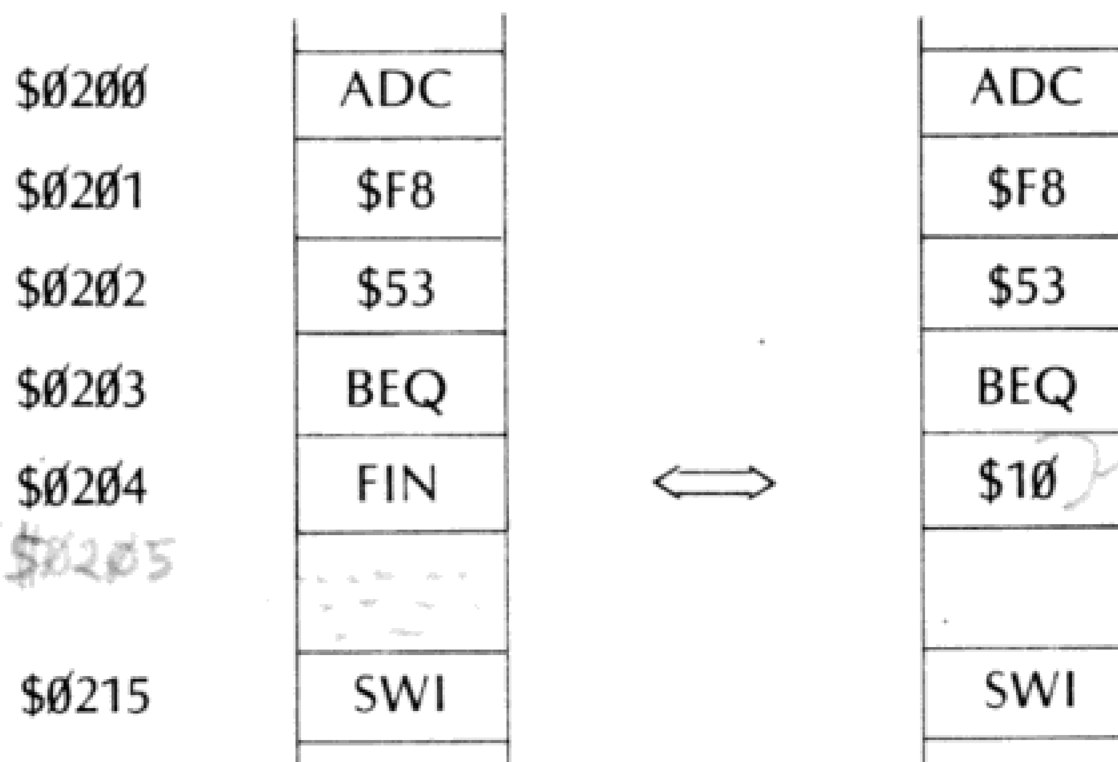
4.2.5. L'adressage relatif

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 codes-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 :

$Ø215 – $Ø2Ø5 = $1Ø

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 :

$Ø2ØØ – $Ø2Ø5 = $–Ø5
= $FB

Nous venons d'envisager des branchements relatifs courts pour lesquels d'une part le code-opération est codé sur un octet et d'autre part le déplacement possible est codé sur un octet également ce qui autorise des déplacements compris entre 127 (branchement relatif vers l'avant) et –128 (branchement relatif vers l'arrière).

Il existe également dans le 68Ø9 des instructions de branchement relatif longs. Dans ce cas, le code-opération est codé sur deux octets et la valeur du déplacement également ce qui permet d'accéder à la totalité de l'espace adressable du 68Ø9.

Naturellement l'utilisation de ce type d'instructions se fait au détriment de la rapidité d'exécution du code assembleur 68Ø9.

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

a) Si on utilise une étiquette :

```
BEQ   FIN     (cas d'un branchement court)
LBEQ  FIN     (cas d'un branchement long)
```

b) Si le déplacement est connu on peut écrire :

```
BEQ  *+1Ø         (cas d'un branchement court)
LBEQ *+$12F6      (cas d'un branchement long)
```

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. Notons que nous verrons un peu plus loin un mode d'adressage très utile pour écrire des programmes relogeables et qui est l'adressage relatif au PC.

4.2.6. Les modes d'adressage indexés

Ces modes d'adressages font du 68Ø9 un microprocesseur extrêmement puissant. Ils peuvent se décomposer en quatre grands groupes qui chacun se divisent en sous-groupes :

— les modes d'adressage indexés à déplacement constant,

— les modes d'adressage indexés par accumulateur,

— les modes d'adressage auto-incrémentés et auto-décrémentés,

— les modes d'adressage relatifs au compteur ordinal (PC).

Nous allons examiner chacun de ces grands groupes à son tour mais nous allons tout d'abord décrire les caractéristiques communes à tous les modes d'adressages indexés.

Le code-opération des instructions possédant ce mode d'adressage est, comme toujours, codé sur un ou deux octets. Tout comme dans le cas des instructions de transfert de registre les modes d'adressage indexés nécessitent un post-octet destiné à renseigner le 68Ø9 sur le type exact d'adressage à utiliser. Nous donnerons à la fin de ce paragraphe un tableau regroupant les différents modes d'adressage indexés ainsi que les post-octets correspondants.

a) Les modes à déplacement constant

Dans ce type de mode d'adressage, l'adresse de la donnée considérée est obtenue en ajoutant le contenu d'un registre interne du 6809 à un déplacement fixe.

Ce déplacement peut être :

* nul et dans ce cas l'adresse de la donnée considérée correspond au contenu du registre spécifié dans l'opérande. L'instruction proprement dite comprend donc le code-opération plus le post-octet qui contient un code de deux bits permettant de connaître le registre par rapport auquel se fait l'indexation (X, Y, U ou S).

Exemple :

```
LDA ,X
```

* codé sur 5 bits et dans ce cas l'adresse de la donnée considérée correspond au contenu du registre spécifié dans l'opérande additionné au déplacement compris entre −16 et +15 spécifié dans l'opérande. L'instruction proprement dite comprend donc le code-opération plus le post-octet qui contient d'une part le code de deux bits dont nous avons parlé ci-dessus, d'autre part le déplacement codé sur 5 bits.

Exemple :

```
LDA 12,X
```

* codé sur 8 bits et dans ce cas l'adresse de la donnée considérée est obtenue en faisant la somme du contenu du registre d'index et du déplacement codé sur 8 bits. Dans ce cas l'instruction proprement dite comporte le code-opération, le post-octet comportant le code de 2 bits dont nous avons parlé ci-dessus et un octet donnant le déplacement sur 8 bits.

Exemple :

```
LDA 102,X
```

En notation en complément à deux le déplacement est compris entre −128 et 127.

* codé sur 16 bits. Le fonctionnement de ce mode d'adressage est similaire au précédent sauf qu'ici le déplacement proprement dit nécessite 16 bits donc 2 octets.

Exemple :

```
LDA $12F5,X
```

En notation avec complément à deux le déplacement est compris entre −32768 et 32767.

b) Les modes indexés par accumulateur

Dans ce cas le contenu des accumulateurs A, B ou D sert comme déplacement. L'adresse de la donnée considérée est donc obtenue en ajoutant le contenu de l'un de ces accumulateurs avec le contenu du registre d'index spécifié.

Comme dans les modes à déplacement constant, le post-octet contient un code de 2 bits permettant de sélectionner le registre d'index considéré.

Si l'accumulateur est A ou B, le champ adressable par variation du contenu de ces registres sera une page de 256 octets. Par contre si l'accumulateur est D on pourra accéder à la totalité de l'espace adressable du 68Ø9.

Exemple :

```
LDA A,X
LDD D,U
```

c) Les modes auto-incrémentés et auto-décrémentés

Ces modes d'adressage sont extrêmement utiles pour l'écriture de boucles.

Le registre concerné contient l'adresse de l'opérande. En mode auto-incrémenté ce registre pointe sur une donnée et effectue le traitement demandé par l'instruction considérée. Il est ensuite incrémenté automatiquement de une ou deux unités ce qui lui permet de pointer à l'adresse de la donnée suivante.

L'adressage auto-incrémenté d'une unité est utilisé lorsque l'on veut accéder par exemple à des tables d'octets. Au contraire, l'adressage auto-incrémenté de deux unités est utilisé lorsque l'on veut accéder à des tables de mots de 16 bits.

Dans le cas de l'adressage auto-décrémenté, le contenu du registre est tout d'abord décrémenté de une ou deux unités suivant le cas. Il contient ensuite l'adresse de la donnée désirée.

Dans tous ces modes le post-octet contient le code de deux bits caractéristique du registre d'index utilisé.

Exemple :

```
STX  Ø,U+      (adressage auto-incrémenté de une unité)
STX  Ø,U++     (adressage auto-incrémenté de deux unités)
LDD  ,-Y       (adressage auto-décrémenté de une unité)
LDD  ,--Y      (adressage auto-décrémenté de deux unités)
```

d) Les modes d'adressage relatifs au compteur ordinal (PC)

Ces modes d'adressage fonctionnent exactement comme les modes à déplacement constant sur 8 ou 16 bits sauf que le registre d'index est ici tout simplement le compteur ordinal. Ceci a comme principal avantage de permettre l'écriture de programmes fonctionnant à n'importe quelle adresse mémoire et ceci sans modification.

L'adresse de l'opérande est obtenue en faisant la somme du déplacement sur 8 ou 16 bits contenu dans l'opérande et le contenu du registre PC.

Exemple :

```
LDA 26,PCR       (déplacement sur 8 bits)
LDA 23F4,PCR     (déplacement sur 16 bits)
```

Le tableau suivant résume les différents modes d'adressage indexés disponibles sur le 68Ø9.

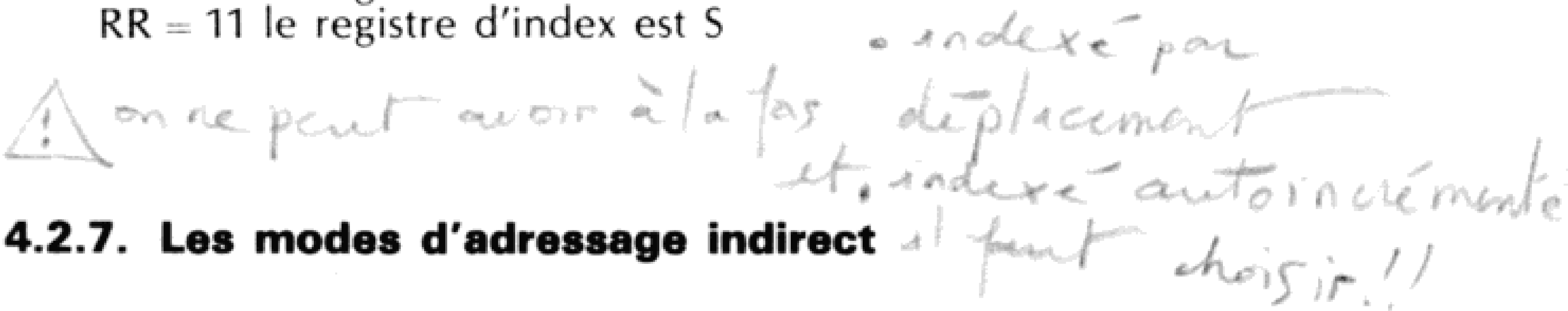
Type	*Forme*	*Syntaxe*	*Post-octet assembleur*
déplacement constant	Ø bits	,R	1RRØØ1ØØ
déplacement constant	5 bits	n,R	ØRRnnnnn
déplacement constant	8 bits	n,R	1RRØ1ØØØ
déplacement constant	16 bits	n,R	1RRØ1ØØ1
déplacement accumulateur	A	A,R	1RRØØ11Ø
déplacement accumulateur	B	B,R	1RRØØ1Ø1
déplacement accumulateur	D	D,R	1RRØ1Ø11
auto-incrémenté	par 1	,R+	1RRØØØØØ
auto-incrémenté	par 2	,R++	1RRØØØØ1
auto-décrémenté	par 1	,–R	1RRØØØ1Ø
auto-décrémenté	par 2	,––R	1RRØØØ11
relatif au PC	8 bits	n,PCR	1xxØ11ØØ
relatif au PC	16 bits	n,PCR	1xxØ11Ø1

Dans ce tableau, "nnnnn" représente une valeur binaire codée sur 5 bits et "xx" signifie "n'importe quelle valeur".

Le code "RR" est le code de deux bits dont nous avons parlé précédemment et chargé de spécifier le registre d'index utilisé.

Ainsi si :

RR = ØØ le registre d'index est X
RR = Ø1 le registre d'index est Y
RR = 1Ø le registre d'index est U
RR = 11 le registre d'index est S



4.2.7. Les modes d'adressage indirect

Le 68Ø9 possède de très nombreux modes d'adressage indirects qui sont dérivés des modes d'adressage indexés décrits précédemment.

Mais tout d'abord voyons ce que l'on appelle par "adressage indirect". Au lieu que l'adresse spécifiée dans l'opérande donne la donnée sur laquelle porte l'instruction, elle en donne l'adresse. L'adresse effective de la donnée est donc contenue dans la case-mémoire spécifiée par l'opérande ainsi que dans la suivante puisqu'il faut deux octets mémoire pour spécifier une adresse.

Nous verrons un exemple un peu plus loin.

Nous avons donc dit que les modes d'adressage indirects sont dérivés des modes d'adressage indexés. Nous allons donc les passer en revue.

a) Les modes indirects indexés à déplacement constant

Tout comme il existait des modes d'adressage indexés à déplacement constant il existe des modes d'adressage indirect indexés à déplacement constant.

Dans ce cas le contenu du registre spécifié ajouté au déplacement (sur 8 bits ou 16 bits uniquement) donne l'adresse de l'adresse de la donnée concernée.

Exemple :

```
LDA (125,X)
```

Les parenthèses indiquent à l'assembleur qu'il est en présence d'un mode d'adressage indirect.

b) Les modes indexés par accumulateur

Ici le contenu du registre spécifié additionné au contenu de l'accumulateur concerné donne l'adresse de l'adresse de la donnée.

Exemple :

```
LDX (A,S)
```

c) Les modes indirects auto-incrémentés et auto-décrémentés

Seuls les modes incrémentés et décrémentés de deux unités sont autorisés. Cela se conçoit aisément puisqu'il faut 2 octets mémoire pour définir une adresse.

A chaque exécution d'une instruction, le contenu du registre spécifié est incrémenté (ou décrémenté) de deux unités de manière à aller pointer sur l'adresse de l'adresse suivante.

Exemple :

```
LDA (,S++)
```

d) Les modes indirects relatif au PC

Le contenu du compteur ordinal, ajouté à un déplacement constant, donne l'adresse de l'adresse de la donnée nécessaire à l'instruction.

Exemple:

```
ROL (128,PCR)
```

e) L'adressage étendu indirect

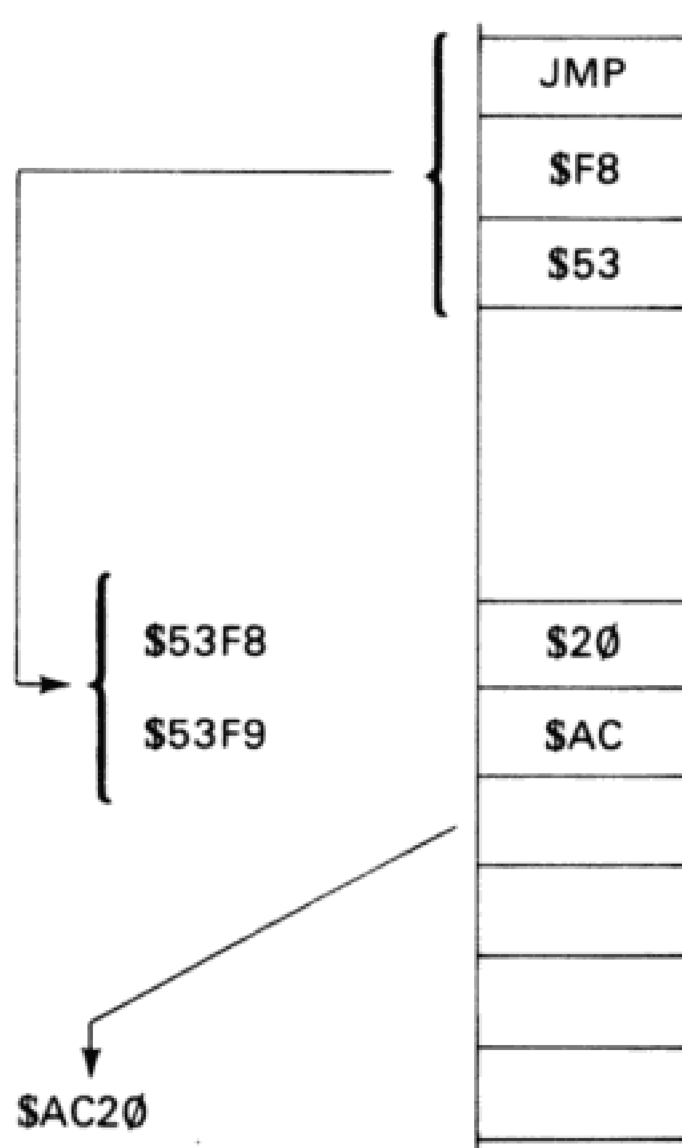
Ce mode est le seul qui ne soit pas dérivé d'un mode indexé.

Tout comme il existe un mode d'adressage étendu, il existe un mode d'adressage étendu indirect. Le nombre spécifié entre crochets donne l'adresse de l'adresse de la donnée concernée.

Exemple: Soit l'instruction:

```
JMP ($53F8)
```

Le microprocesseur, lorsqu'il rencontre l'instruction JMP de saut inconditionnel, se branche à l'adresse $53F8 et charge son compteur ordinal PC avec les contenus des cases-mémoire d'adresses $53F8 et $53F9 (l'octet de poids faible du PC est stocké dans la première adresse: $53F8). Il y a donc branchement à l'adresse $AC2Ø.



Nous allons maintenant donner un tableau regroupant les différents modes d'adressage indirects disponibles sur le 68Ø9.

Type	*Forme*	*Syntaxe*	*Post-octet assembleur*
déplacement constant	Ø bits	(,R)	1RR1Ø1ØØ
déplacement constant	8 bits	(n,R)	1RR11ØØØ
déplacement constant	16 bits	(n,R)	1RR11ØØ1
déplacement accumulateur	A	(A,R)	1RR1Ø11Ø
déplacement accumulateur	B	(B,R)	1RR1Ø1Ø1
déplacement accumulateur	D	(D,R)	1RR11Ø11
auto-incrémenté	par 2	(,R++)	1RR1ØØØ1
auto-incrémenté	par 2	(,−−R)	1RR1ØØ11
relatif au PC	8 bits	(n,PCR)	1xx111ØØ
relatif au PC	16 bits	(n,PCR)	1xx111Ø1
étendu indirect		(n)	1ØØ11111

5
Le jeu d'instructions du 68Ø9

5.1. INTRODUCTION

Contrairement à ce qui se fait couramment, nous avons choisi de ne pas vous présenter les instructions du 68Ø9 dans l'ordre alphabétique mais par "affinités", abondemment commentées et illustrées de nombreux exemples.

Le 68Ø9, nous en avons déjà parlé dans le chapitre précédent, possède 59 instructións différentes qui, combinées avec les différents modes d'adressage, portent ce nombre à 1 4Ø4 ce qui fait de ce microprocesseur le plus puissant des "8 bits" du marché.

De conception récente par rapport à ceux existant aujourd'hui (65Ø2 et Z 8Ø par exemple), il commence à être largement utilisé dans les micro-ordinateurs individuels.

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.

De plus, les exemples que nous vous proposons ont une difficulté croissante. Étant donné l'étendue du jeu d'instructions de ce microprocesseur, il ne sera bien sûr pas possible de donner un exemple pour chacune d'entre elles et pour chaque mode d'adressage. Néanmoins nous essaierons d'utiliser chacun d'entre eux à son tour.

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 registre.

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, LDA, LDB.

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 LDA et LDB 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 CC.

Nous avons vu dans le chapitre précédent la signification de ces 2 bits :

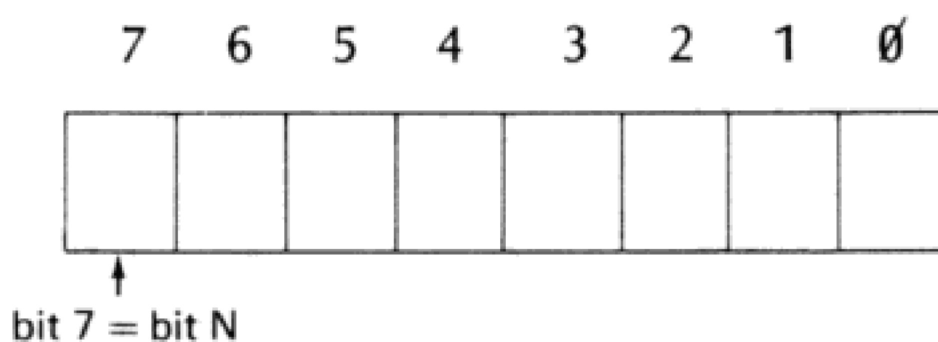
N = indicateur de résultat négatif
Z = indicateur de zéro

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 **$ØØ**, 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 Ø et 255 soit un nombre signé 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 :



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

Exemple:

$17=ØØØ1Ø111

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	direct	ØF	N=V=C=Ø, Z=1
CLR	indexé	6F	N=V=C=Ø, Z=1
CLR	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 standards MOTOROLA. Nous ne donnons pas dans ce tableau la syntaxe des différents modes d'adressage disponibles pour l'instruction CLR puisque ceux-ci ont été décrits en détails précédemment.

Notons de plus que le mode d'adressage indexé regroupe bien sûr à la fois les 13 modes d'adressage indexés et les 11 modes d'adressage indirects.

Exemple :

Soit l'instruction CLRA avec A = $32 initialement.

Après l'exécution de cette instruction, le contenu du registre A sera nul. Les registres internes du 68Ø9 seront donc affectés de la manière suivante :

<table>
<tr><td colspan="2">X=$xxxx</td></tr>
<tr><td colspan="2">Y=$xxxx</td></tr>
<tr><td colspan="2">U=$xxxx</td></tr>
<tr><td colspan="2">S=$xxxx</td></tr>
<tr><td>A=$ØØ</td><td>B=$xx</td></tr>
</table>

DP=$xx

x x x x Ø 1 Ø Ø	CC

Les notations employées sont les suivantes :

— x désigne une valeur indéterminée sur un bit ;
— xx désigne une valeur indéterminée sur un octet (2 chiffres hexadécimaux) dans le cas des registres A, B, DP ;
— xxxx désigne une valeur indéterminée sur deux octets (4 chiffres hexadécimaux) dans le cas des registres X, Y, U et S.

b) Les instructions LDA et LDB

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 LDA)
M→B (instruction LDB)

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
LDA	immédiat	86	N, Z, V=Ø
LDA	direct	96	N, Z, V=Ø
LDA	indexé	A6	N, Z, V=Ø
LDA	étendu	B6	N, Z, V=Ø
LDB	immédiat	C6	N, Z, V=Ø
LDB	direct	D6	N, Z, V=Ø
LDB	indexé	E6	N, Z, V=Ø
LDB	étendu	F6	N, Z, V=Ø

Considérons l'instruction LDA #$94.

Nous sommes donc en présence d'un mode d'adressage immédiat. Le registre A sera donc tout simplement chargé avec la valeur $94.

Celle-ci étant négative et non nulle, le bit N sera positionné à 1 et le bit Z à zéro.

Le contenu des registres internes du 68Ø9 sera donc le suivant:

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$94	B=$xx

DP=$xx

x x x x 1 Ø Ø x	CC

5.2.1.2. Les instructions sur 16 bits

Il s'agit d'une part des instructions de chargement de registres 16 bits qui fonctionnent de manière similaire aux instructions LDA et LDB vues précédemment, d'autre part des instructions SEX, LEAX, LEAY, LEAS et LEAU. Nous reviendrons sur ces instructions un peu plus loin.

a) Les instructions LDD, LDS, LDU, LDX et LDY

Ces instructions permettent de charger respectivement les registres D (constitué de la juxtaposition des registres A et B), S, U, X et Y avec le contenu de la case-mémoire spécifiée ainsi que de la suivante, ou avec une valeur binaire (cas d'un adressage immédiat).

On aura donc : M,M+1 → D (instruction LDD)
M,M+1 → S (instruction LDS)
etc...

L'octet de poids fort (A) 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 (B) est chargé en second avec le contenu de la case-mémoire d'adresse M+1.

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
LDD	immédiat	CC	N, Z, V=Ø
LDD	direct	DC	N, Z, V=Ø
LDD	indexé	EC	N, Z, V=Ø
LDD	étendu	FC	N, Z, V=Ø
LDS	immédiat	1Ø CE	N, Z, V=Ø
LDS	direct	1Ø DE	N, Z, V=Ø
LDS	indexé	1Ø EE	N, Z, V=Ø
LDS	étendu	1Ø FE	N, Z, V=Ø
LDU	immédiat	CE	N, Z, V=Ø
LDU	direct	DE	N, Z, V=Ø
LDU	indexé	EE	N, Z, V=Ø
LDU	étendu	FE	N, Z, V=Ø
LDX	immédiat	8E	N, Z, V=Ø
LDX	direct	9E	N, Z, V=Ø
LDX	indexé	AE	N, Z, V=Ø
LDX	étendu	BE	N, Z, V=Ø
LDY	immédiat	1Ø 8E	N, Z, V=Ø
LDY	direct	1Ø 9E	N, Z, V=Ø
LDY	indexé	1Ø AE	N, Z, V=Ø
LDY	étendu	1Ø BE	N, Z, V=Ø

Soit l'instruction LDX $12F8.

Nous sommes en présence d'un mode d'adressage étendu où les adresses concernées sont $12F8 et $12F9. Supposons qu'à ces adresses se trouvent $ØØ. Après exécution de cette instruction, le contenu des registres internes du 68Ø9 sera affecté de la manière suivante :

X=$ØØØØ	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$xx	B=$xx

DP=$xx

x x x x Ø 1 Ø x	CC

b) Les instructions LEAS, LEAU, LEAX et LEAY

Ces instructions permettent de charger une adresse effective respectivement dans les registres pointeurs de pile et d'index. Cette adresse effective est donnée sur deux octets bien entendu.

Ces instructions fonctionnent exactement comme les instructions de chargement correspondantes mais au lieu de charger la donnée elles chargent l'adresse calculée de celle-ci dans le registre concerné.

On a donc : EA→S (instruction LEAS)
EA→U (instruction LEAU)

etc...

(EA signifiant "adresse effective").

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
LEAS	indexé	32	néant
LEAU	indexé	33	néant
LEAX	indexé	3Ø	Z
LEAY	indexé	31	Z

Notons que ces instructions ne fonctionnent qu'avec un mode d'adressage indexé (ou indirect bien sûr) puisque ce sont les seuls modes qui nécessitent un calcul d'adresse effective.

Considérons l'instruction LEAS $8Ø,X.

Il s'agit ici d'un mode d'adressage indexé à déplacement codé sur 8 bits. Le registre d'index est le registre X et est supposé contenir la valeur $2ØØØ par exemple. L'adresse effective calculée par l'instruction LEAS $8Ø,X sera donc égale à $2Ø8Ø.

Le contenu du registre de condition CC ne sera pas affecté par cette instruction.

Le contenu des registres internes du 68Ø9 sera donc :

<table>
<tr><td colspan="2">X=$2ØØØ</td></tr>
<tr><td colspan="2">Y=$xxxx</td></tr>
<tr><td colspan="2">U=$xxxx</td></tr>
<tr><td colspan="2">S=$2Ø8Ø</td></tr>
<tr><td>A=$xx</td><td>B=$xx</td></tr>
</table>

DP=$xx

x x x x x x x x	CC

c) L'instruction SEX

En anglais elle signifie "extension de signe" et porte uniquement sur le contenu de l'accumulateur A.

Elle fonctionne de la manière suivante :

Si le bit 7 du registre B est égal à 1 (cas d'un nombre négatif) alors le registre A sera chargé avec la valeur $FF. Si par contre le bit 7 du registre B est égal à Ø (cas d'un nombre positif) le registre A sera chargé avec la valeur $ØØ.

En fait le nombre codé en complément à deux stocké dans l'accumulateur B (sous forme d'un nombre de 8 bits) est tout simplement transformé en nombre en complément à deux sur 16 bits et donc stocké dans le registre D.) Rappel : registre D = concaténation des registres A et B chacun sur 8 bits, pour faire des mots de 16 bits.

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
SEX	implicite	1D	N, Z, V=Ø

Soit par exemple l'instruction SEX avec B contenant la valeur $83.

Le contenu des registres internes du 68Ø9 sera donc affecté de la manière suivante :

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$FF	B=$83

DP=$xx

x x x x 1 Ø Ø x	CC

5.2.2. Les instructions de chargement mémoire

5.2.2.1. Les instructions sur 8 bits

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

On aura donc :

A → M (instruction STA)
B → M (instruction STB)

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
STA	direct	97	N, Z, V=Ø
STA	indexé	A7	N, Z, V=Ø
STA	étendu	B7	N, Z, V=Ø
STB	direct	D7	N, Z, V=Ø
STB	indexé	E7	N, Z, V=Ø
STB	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 STA (par exemple) range le contenu de l'accumulateur dans une case-mémoire d'adresse définie.

Soit l'instruction STB A, X.

Nous reconnaissons ici un mode d'adressage indexé à déplacement accumulateur. Le registre d'index est X et l'accumulateur utilisé pour le déplacement est A. Le contenu du registre X est donc ajouté au contenu de l'accumulateur A pour former l'adresse effective où aura lieu le stockage du contenu du registre B.

Supposons que le registre X contienne la valeur $2ØØØ, l'accumulateur A la valeur $8Ø et l'accumulateur B la valeur $32.

Cette dernière valeur sera donc stockée à l'adresse $2Ø8Ø et le contenu des registres internes après exécution de l'instruction sera donc le suivant :

X=$2ØØØ	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$8Ø	B=$32

DP=$xx

x x x x Ø Ø Ø x	CC

5.2.2.2. Les instructions sur 16 bits

De même qu'il existait les opérations de chargement de registre LDD, LDX, LDY, LDS, LDU, il existe les instructions de chargement mémoire correspondantes qui sont donc STD, STX, STY, STS et STY.

On aura donc :

D → M,M+1 (instruction STD)
S → M,M+1 (instruction STS)

etc...

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

Cas d'un STD ?

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
STD	direct	DD	N, Z, V=Ø
STD	indexé	ED	N, Z, V=Ø
STD	étendu	FD	N, Z, V=Ø
STS	direct	1Ø DF	N, Z, V=Ø
STS	indexé	1Ø EF	N, Z, V=Ø
STS	étendu	1Ø FF	N, Z, V=Ø
STU	direct	DF	N, Z, V=Ø
STU	indexé	EF	N, Z, V=Ø
STU	étendu	FF	N, Z, V=Ø
STX	direct	9F	N, Z, V=Ø
STX	indexé	AF	N, Z, V=Ø
STX	étendu	BF	N, Z, V=Ø
STY	direct	1Ø 9F	N, Z, V=Ø
STY	indexé	1Ø AF	N, Z, V=Ø
STY	étendu	1Ø BF	N, Z, V=Ø

Considérons l'instruction STD ,U++.

Nous reconnaissons ici un mode d'adressage auto-incrémenté deux fois. Nous supposons que le registre U contient initialement la valeur $53F2 et que l'accumulateur D contient la valeur $1ØØØ.

Lors de l'exécution de cette instruction les opérations suivantes auront lieu :

— la case-mémoire d'adresse $53F2 sera chargée avec la valeur $1Ø (octet de poids fort) tandis que la case-mémoire d'adresse $53F3 sera chargée avec $ØØ (octet de poids faible) ;

— le registre U est incrémenté deux fois.

Après exécution de cette instruction, les registres internes auront l'état suivant :

X=$xxxx	
Y=$xxxx	
U=$53F4	
S=$xxxx	
A=$1Ø	B=$ØØ

DP=$xx

x x x x Ø Ø Ø x	CC

5.2.3. Les instructions de transfert et d'échange de registres

Ces instructions permettent d'échanger le contenu de deux registres ou bien de transférer le contenu de l'un d'entre-eux dans un autre.

Les instructions de type EXG R1, R2 permettent donc d'échanger le contenu d'une paire quelconque de registres. Ces deux registres doivent bien sûr avoir la même taille (8 bits ou 16 bits).

Les registres 8 bits sont à choisir parmi A, B, CC ou DP tandis que les registres 16 bits sont à choisir parmi X, Y, U, S, D ou PC.

On aura donc pour ces instructions :

R1↔R2

Les instructions de type TFR R1, R2 permettront de même de transférer le contenu de R1 dans R2.

On aura donc :

R1→R2

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
EXG R1, R2 TFR R1, R2	implicite implicite	1E 1F	néant néant

Comme nous l'avons précisé dans le chapitre concernant les différents modes d'adressage, le contenu de l'octet suivant le code-opération (appelé post-octet) précise la paire de registres sur laquelle s'appliquent ces instructions.

Les 4 bits de poids faible donnent le registre destination (R2 en l'occurrence) tandis que les 4 bits de poids fort donnent le registre source (R1 en l'occurrence).

Notons que cette distinction ne concerne bien sûr que l'instruction TFR.

Le tableau ci-dessous donne le registre concerné en fonction du code de 4 bits utilisé :

ØØØØ	D (A,B)
ØØØ1	X
ØØ1Ø	Y
ØØ11	U
Ø1ØØ	S
Ø1Ø1	PC
1ØØØ	A
1ØØ1	B
1Ø1Ø	CC
1Ø11	DP

Considérons l'instruction EXG A, B. Les registres A et B contiennent respectivement avant exécution les valeurs **\$1Ø** et **\$3Ø**.

Après exécution de cette instruction, le contenu des registres internes sera le suivant :

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$3Ø	B=$1Ø

DP=$xx

x x x x x x x x	CC

Nous venons de décrire le fonctionnement des instructions les plus employées dans tout programme "tournant sur 68Ø9". 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 appellerons instructions arithmétiques les instructions suivantes :

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

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

— l'instruction MUL (multiplication),

— les instructions d'incrémentation : INC, INCA, INCB,

— les instructions de décrémentation : DEC, DECA, DECB,

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

— 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 = ØØØØØ1Ø1
      └──┘└──┘
       Ø   5

$17 = ØØØ1Ø111
      └──┘└──┘
       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          Ø5       ØØØØØ1Ø1
+ $17        + 23     + ØØØ1Ø111
-----        ----     ----------
= $1C        = 28     = ØØØ111ØØ
                        └──┘└──┘
                         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 CC.

5.3.1.2. Retour sur le registre CC

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

Nous avons vu dans le chapitre précédent 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 a effectuer la somme de deux nombres $8A et $D5.

En représentation binaire on a :

$8A = 1ØØØ1Ø1Ø
$D5 = 11Ø1Ø1Ø1

La somme de ces deux nombres donne :

```
  $8A            1ØØØ1Ø1Ø
+ $D5          + 11Ø1Ø1Ø1
-------        ------------
= $15F         = (1)Ø1Ø11111
```

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 $Ø8 et $19 ($Ø8 + $19 = $21), 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 codes 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 de type "carry" vue précédemment, dépassement de capacité (> 127) lors de l'addition de 2 nbres positifs.

— 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 = Ø1ØØ1Ø11
$71 = Ø111ØØØ1

$4B	Ø1ØØ1Ø11
+ $71	+ Ø111ØØØ1
= $BC	= 1Ø1111ØØ

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 signés, ils sont tous deux positifs (75 et 113) mais leur somme, qui est positive, dépasse la capacité du microprocesseur (188 est supérieur à 127) et 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 $–43.

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

```
   $–Ø1          11111111
 + $–Ø5        + 11111Ø11
 --------      ------------
 = $–Ø6        = (1)11111Ø1Ø
```

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 signés 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 et ADDB.

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 :

$A + \text{bit } C + M \rightarrow A$

ou bien :

$B + \text{bit } C + M \rightarrow B$

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
ADCA	immédiat	89	H, N, Z, V, C
ADCA	direct	99	H, N, Z, V, C
ADCA	indexé	A9	H, N, Z, V, C
ADCA	étendu	B9	H, N, Z, V, C
ADCB	immédiat	C9	H, N, Z, V, C
ADCB	direct	D9	H, N, Z, V, C
ADCB	indexé	E9	H, N, Z, V, C
ADCB	étendu	F9	H, N, Z, V, C

Exemple : Soit l'instruction ADCA #$71 avec C=1 et A=$4B initialement.

Le résultat de cette addition est $BD avec C=Ø.

Les registres internes du 68Ø9 sont donc affectés de la manière suivante :

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$BD	B=$xx

DP=$xx

x x Ø x 1 Ø 1 Ø	CC

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	immédiat	8B	H, N, Z, V, C
ADDA	direct	9B	H, N, Z, V, C
ADDA	indexé	AB	H, N, Z, V, C
ADDA	étendu	BB	H, N, Z, V, C
ADDB	immédiat	CB	H, N, Z, V, C
ADDB	direct	DB	H, N, Z, V, C
ADDB	indexé	EB	H, N, Z, V, C
ADDB	étendu	FB	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 codés BCD. Elle permet, comme son nom l'indique, d'effectuer un ajustement décimal sur l'accumulateur A.

Un ajustement décimal consiste à ajouter la valeur Ø6 au contenu de l'accumulateur. Nous allons voir pourquoi avec un exemple.

Considérons les deux nombres BCD 8 et 7.

On a :

```
  8 = ØØØØ1ØØØ
  7 = ØØØØØ111
------------------
8+7 = ØØØØ1111 = 15
```

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.

L'instruction DAA permet donc de travailler aisément avec des nombres considérés comme décimaux et non binaires.

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

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	immédiat	C3	N, Z, V, C
ADDD	direct	D3	N, Z, V, C
ADDD	indexé	E3	N, Z, V, C
ADDD	étendu	F3	N, Z, V, C

Exemple: Soit l'instruction ADDD $12, X.

Nous sommes en présence d'un adressage indexé à déplacement constant codé sur 5 bits.

Nous supposons que X=$1ØØØ initialement et que les adresses $1Ø12 et $1Ø13 contiennent respectivement les valeurs $12 et $F4.

Nous supposons de plus que D=$2ØØØ initialement.

Les registres internes du 68Ø9 sont donc affectés de la manière suivante :

X=$1ØØØ	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$32	B=$F4

DP=$xx

x x x x Ø Ø Ø Ø	CC

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

On a donc :

$B + X \rightarrow X$

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
ABX	implicite	3A	aucun

Nous allons maintenant envisager un programme d'addition BCD sur 16 bits.

Nous voulons additionner les deux nombres A et B (stockés initialement aux adresses ADR1 et ADR2 constituées de parties hautes et basses H et L). Le résultat est alors stocké à l'adresse ADR3.

```
LDA     ADR1L      ; charge octet de poids faible de A
ADDA    ADR2L      ; additionne octet de poids faible de B
DAA                ; ajustement décimal
STA     ADR3L      ; sauvegarde octet de poids faible
LDA     ADR1H      ; charge octet de poids fort de A
ADCA    ADR2H      ; additionne octet de poids fort de B
DAA                ; ajustement décimal
STA     ADR3H      ; sauvegarde octet de poids fort
SWI
```

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'un microprocesseur ne sait pas faire de soustractions : il ne fait que des additions.

Ceci dit il est très facile, à partir d'une soustraction, de se ramener à une addition. En effet, on additionne au premier nombre l'inverse du second. C'est ainsi qu'en arithmétique décimale classique on a :

13 – 1Ø = 13 + (–1Ø)

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

$20 – $15 = $20 + ($–15)

$20 = 00100000
$15 = 00010101
$–15 = 11101011
donc : $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 et SUBB.

Les instructions SBCA et SBCB sont les analogues 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 adressage immédiat), ainsi que le bit C du registre CC 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	immédiat	82	N, Z, V, C
SBCA	direct	92	N, Z, V, C
SBCA	indexé	A2	N, Z, V, C
SBCA	étendu	B2	N, Z, V, C
SBCB	immédiat	C2	N, Z, V, C
SBCB	direct	D2	N, Z, V, C
SBCB	indexé	E2	N, Z, V, C
SBCB	étendu	F2	N, Z, V, C

Notons que le bit H après une instruction de type SBC prend une valeur quelconque.

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	immédiat	8Ø	N, Z, V, C
SUBA	direct	9Ø	N, Z, V, C
SUBA	indexé	AØ	N, Z, V, C
SUBA	étendu	BØ	N, Z, V, C
SUBB	immédiat	CØ	N, Z, V, C
SUBB	direct	DØ	N, Z, V, C
SUBB	indexé	EØ	N, Z, V, C
SUBB	étendu	FØ	N, Z, V, C

Comme dans le cas des instructions de type SBC, le bit H prend une valeur quelconque après une instruction de type SUB.

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	immédiat	83	N, Z, V, C
SUBD	direct	93	N, Z, V, C
SUBD	indexé	A3	N, Z, V, C
SUBD	étendu	B3	N, Z, V, C

Nous n'avons pas ici inclus de l'exemples puisque le principe de fonctionnement des instructions de soustraction est le même que celui des instructions d'addition vues précédemment.

5.3.3. Les instructions d'incrémentation

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

5.3.3.1. Notions 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 et INCB

On a donc :

A + 1 → A (instruction INCA)
B + 1 → B (instruction INCB)
M + 1 → M (instruction INC)

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
INC	direct	ØC	N, Z, V
INC	indexé	6C	N, Z, V
INC	étendu	7C	N, Z, V
INCA	implicite	4C	N, Z, V
INCB	implicite	5C	N, Z, V

Exemple :

Soit l'instruction INCA avec A=$FF initialement. Le résultat d'une telle instruction sera A=$ØØ et le contenu des registres internes du 68Ø9 sera affecté de la manière suivante :

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$ØØ	B=$xx

DP=$xx

x x x x Ø 1 Ø x	CC

5.3.4. Les instructions de décrémentation

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

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 et DECB

On a donc:

A − 1→A (instruction DECA)
B − 1→B (instruction DECB)
M − 1→M (instruction DEC)

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
DEC	direct	ØA	N, Z, V
DEC	indexé	6A	N, Z, V
DEC	étendu	7A	N, Z, V
DECA	implicite	4A	N, Z, V
DECB	implicite	5A	N, Z, V

5.3.5. L'instruction MUL

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

Les contenus des registres A et B sont considérés comme des nombres non signés.

La présence de cette instruction est une particularité du 68Ø9 rarement trouvée sur les microprocesseurs 8 bits du marché.

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

On a donc :

$A \times B \rightarrow D$

Nous allons brièvement décrire ce que nous appelons par multiplication binaire.

En décimal on multiplie un multiplicande par un multiplicateur de la manière suivante :

```
   32      multiplicande
 × 25      multiplicateur
 ----
  16Ø
+ 64
 ----
  8ØØ
```

Le chiffre le plus à droite du multiplicateur est multiplié par le multiplicande (5 × 32). Puis le deuxième chiffre du multiplicateur est à son tour multiplié par le multiplicande et décalé d'un rang vers la gauche avant d'être ajouté au résultat partiel obtenu précédemment. Soit maintenant à effectuer une multiplication en binaire :

```
     11Ø       (6)
 ×   Ø1Ø       (2)
 -------
     ØØØ
 +  11Ø
 + ØØØ
 -------
   Ø11ØØ       (12)
```

Elle s'effectue de la même manière qu'en décimal.

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
MUL	implicite	3D	Z, C=1 si bit B7=1

Exemple: Soit à effectuer la multiplication $12 × $5E.

Le résultat, égal à $Ø69C sera stocké dans D.

Le contenu des registres internes du 68Ø9 est donc affecté de la manière suivante:

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$Ø6	B=$9C

DP=$xx

x x x x x Ø x x	CC

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:

$\overline{A} + 1 \rightarrow A$

ou bien:

$\overline{B} + 1 \rightarrow B$

ou bien :

$\overline{M} + 1 \rightarrow M$

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
NEG	direct	ØØ	N, Z, V, C
NEG	indexé	6Ø	N, Z, V, C
NEG	étendu	7Ø	N, Z, V, C
NEGA	implicite	4Ø	N, Z, V, C
NEGB	implicite	5Ø	N, Z, V, C

Notons que le bit H prend une valeur quelconque après l'exécution d'une instruction de négation.

5.3.7. Les instructions de décalage arithmétique

Il s'agit des instructions ASL, ASLA, ASLB, 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'ou le nom de décalage "arithmétique".

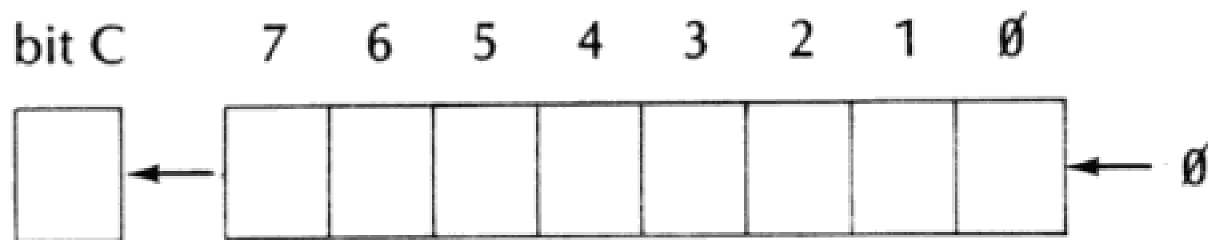
Exemple : Soit l'instruction ASLA (adressage implicite).

On suppose que A = $9D initialement :

$9D = 1ØØ111Ø1

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.

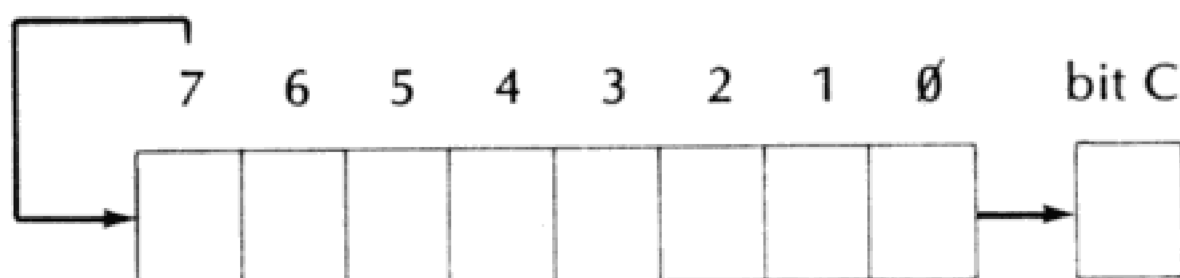
Exemple : Soit l'instruction ASRA.

On suppose que A = \$9D initialement.

Après décalage cela nous donne :

\$CE = 11ØØ111Ø et C=1

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



5.3.7.2. Les instructions ASL, ASLA et ASLB

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
ASL	direct	Ø8	N, Z, V, C
ASL	indexé	68	N, Z, V, C
ASL	étendu	78	N, Z, V, C
ASLA	implicite	48	N, Z, V, C
ASLB	implicite	58	N, Z, V, C

Exemple: Soit l'instruction ASL ,U qui représente un mode d'adressage indexé par U avec déplacement nul. Nous supposons que U contient la valeur $63C8 initialement et qu'à cette adresse se trouve la valeur $9D.

Après exécution de cette instruction, le contenu des registres internes du 68Ø9 est affecté de la manière suivante:

X=$xxxx	
Y=$xxxx	
U=$63C8	
S=$xxxx	
A=$3A	B=$xx

DP=$xx

x x x x Ø Ø Ø 1	CC

5.3.7.3. Les instructions ASR, ASRA et ASRBRL

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
ASR	direct	Ø7	N, Z, V, C
ASR	indexé	67	N, Z, V, C
ASR	étendu	77	N, Z, V, C
ASRA	implicite	47	N, Z, V, C
ASRB	implicite	57	N, Z, V, C

5.4. LES INSTRUCTIONS LOGIQUES

Nous regrouperons sous cette appellation les instructions suivantes :

— le "ET" logique : ANDA, ANDB,

— le "OU" logique : ORA, ORB,

— le "OU" exclusif : EORA, EORB,

— les instructions de complémentation : COM, COMA, COMB,

— les instructions de décalage logique : LSL, LSLA, LSLB, LSR, LSRA, LSRB,

— les instructions de rotation : ROL, ROLA, ROLB, ROR, RORA, RORB,

— les instructions de test de bits : BITA, BITB, TST, TST, TSTA, TSTB.

5.4.1. 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 68Ø9 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 $A \wedge B = 1$
sinon $A \wedge B = \emptyset$

On peut définir une table de vérité pour cette opération.

A \ B	Ø	1
Ø	Ø	Ø
1	Ø	1

Les chiffres inscrits dans les 4 cases centrales donnent la valeur de $A \wedge B$ 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	immédiat	84	N, Z, V=Ø
ANDA	direct	94	N, Z, V=Ø
ANDA	indexé	A4	N, Z, V=Ø
ANDA	étendu	B4	N, Z, V=Ø
ANDB	immédiat	C4	N, Z, V=Ø
ANDB	direct	D4	N, Z, V=Ø
ANDB	indexé	E4	N, Z, V=Ø
ANDB	étendu	F4	N, Z, V=Ø

Exemple: Soit à effectuer l'opération suivante:

ANDA $12F5

L'adressage utilisé est ici l'adressage étendu.

On suppose qu'avant l'exécution de cette instruction, la case-mémoire d'adresse $12F5 contient la valeur $35 et l'accumulateur A contient $12.

Après l'exécution de cette instruction, le contenu de la case-mémoire d'adresse $12F5 reste inchangé et le contenu des registres internes est affecté de la manière suivante:

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$1Ø	B=$xx

DP=$xx

x x x x Ø Ø Ø x	CC

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 :

$A = B = \emptyset$ alors $A \vee B = \emptyset$

sinon :

$A \vee B = 1$

La table de vérité de cette opération est la suivante :

A \ B	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

$12 ∨ $35 = ØØ11Ø111 = $37

5.4.2.2. Les instructions ORA et ORB

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 :

$A \vee M \rightarrow A$ (cas de l'instruction ORA)
$B \vee M \rightarrow B$ (cas de l'instruction ORB)

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
ORA	immédiat	8A	N, Z, V=Ø
ORA	direct	9A	N, Z, V=Ø
ORA	indexé	AA	N, Z, V=Ø
ORA	étendu	BA	N, Z, V=Ø
ORB	immédiat	CA	N, Z, V=Ø
ORB	direct	DA	N, Z, V=Ø
ORB	indexé	EA	N, Z, V=Ø
ORB	étendu	FA	N, Z, V=Ø

Exemple :Soit à effectuer l'opération suivante :

ORB $12F5

L'adressage utilisé est ici aussi l'adressage étendu.

On suppose qu'avant l'exécution de cette instruction, la case-mémoire d'adresse $12F5 contient la valeur $35 et l'accumulateur B contient $12.

Après l'exécution de cette instruction, le contenu de la case-mémoire d'adresse $12F5 reste inchangé et le contenu des registres internes est affecté de la manière suivante :

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$xx	B=$37

DP=$xx

x x x x ø ø ø x	CC

5.4.3. Le "OU exclusif"

5.4.3.1. Notion de "OU exclusif"

L'opération "OU exclusif" (notée souvent V) est définie de la manière suivante :

si A = B alors A ⩡ B = ø
si A ≠ B alors A ⩡ 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 :

A \ B	ø	1
ø	ø	1
1	1	0

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 V \$35:

$$
\begin{array}{r}
\$12 = 00010010 \\
\$35 = 00110101 \\
\hline
\$12 \veebar \$35 = 00100111 = \$27
\end{array}
$$

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:

$A \veebar M \rightarrow A$ (cas de l'instruction EORA)

ou bien:

$B \veebar M \rightarrow B$ (cas de l'instruction EORB)

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
EORA	immédiat	88	N, Z, V=0
EORA	direct	98	N, Z, V=0
EORA	indexé	A8	N, Z, V=0
EORA	étendu	B8	N, Z, V=0
EORB	immédiat	C8	N, Z, V=0
EORB	direct	D8	N, Z, V=0
EORB	indexé	E8	N, Z, V=0
EORB	étendu	F8	N, Z, V=0

Exemple: Soit à effectuer l'opération suivante:

EORA # \$35

L'adressage utilisé ici est l'adressage immédiat.

On suppose qu'avant l'exécution de cette instruction l'accumulateur A contient \$12.

Après l'exécution de cette instruction le contenu des registres internes est affecté de la manière suivante :

X=\$xxxx	
Y=\$xxxx	
U=\$xxxx	
S=\$xxxx	
A=\$27	B=\$xx

DP=\$xx

x x x x Ø Ø Ø x	CC

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 $\overline{A} = Ø$ (A est le complément de A)

si A = Ø alors $\overline{A} = 1$

Exemple : Soit à effectuer : $\overline{\$12}$

\$12 = ØØØ1ØØ1Ø alors $\overline{\$12}$ = 11Ø111Ø1 = \$ED

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	direct	Ø3	N, Z, C=1, V=Ø
COM	indexé	63	N, Z, C=1, V=Ø
COM	étendu	73	N, Z, C=1, V=Ø

Exemple : Soit l'instruction COMB avec B contenant $12 initialement.

Après l'exécution de cette instruction, le contenu des registres internes est affecté de la manière suivante :

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$xx	B=$ED

DP=$xx

x x x x 1 Ø Ø 1 CC

5.4.5. Les instructions de décalage logique

Il s'agit des instructions suivantes :

— les instructions de décalage à gauche : LSL, LSLA, LSLB,

— les instructions de décalage à droite : LSR, LSRA, LSRB.

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 CC 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).

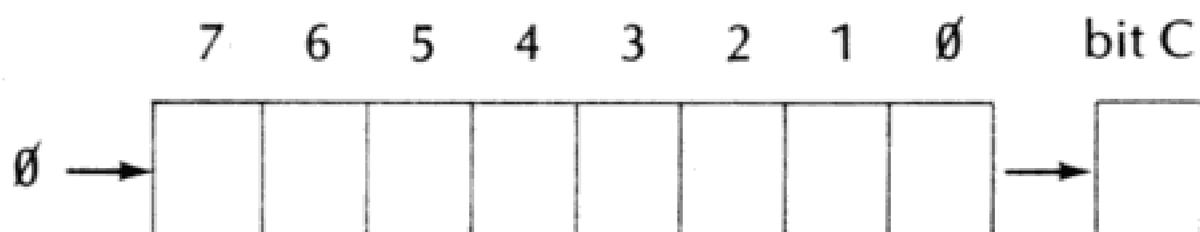
On suppose que A = $9D initialement :

$9D = 1ØØ111Ø1

Après décalage cela nous donne :

$4E = Ø1ØØ111Ø et C=1

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



5.4.5.2. Les instructions LSL, LSLA, LSLB

Le décalage 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*
LSLA	implicite	48	N, Z, V, C
LSLB	implicite	58	N, Z, V, C
LSL	direct	Ø8	N, Z, V, C
LSL	indexé	68	N, Z, V, C
LSL	étendu	78	N, Z, V, C

Nous ne donnerons pas ici d'exemple de fonctionnement de ces instructions puisqu'elles sont similaires aux instructions ASL.

5.4.5.3. Les instructions LSR, LSRA, LSRB

Le décalage 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*
LSRA	implicite	44	Z, C, N=Ø
LSRB	implicite	54	Z, C, N=Ø
LSR	direct	Ø4	Z, C, N=Ø
LSR	indexé	64	Z, C, N=Ø
LSR	étendu	74	Z, C, N=Ø

Considérons l'instruction LSR<$2Ø avec DP contenant la valeur $Ø1.

Il s'agit donc d'un mode d'adressage direct qui porte sur l'adresse $Ø12Ø (puisque le registre de page directe contient la valeur $Ø1).

Nous supposons de plus que la case-mémoire d'adresse $Ø12Ø contient la valeur $9D initialement.

Après exécution de l'instruction, cette case-mémoire contiendra la valeur $4E et le contenu des registres internes sera modifié de la façon suivante :

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$xx	B=$xx

DP=$Ø1

x x x x Ø Ø x 1	CC

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

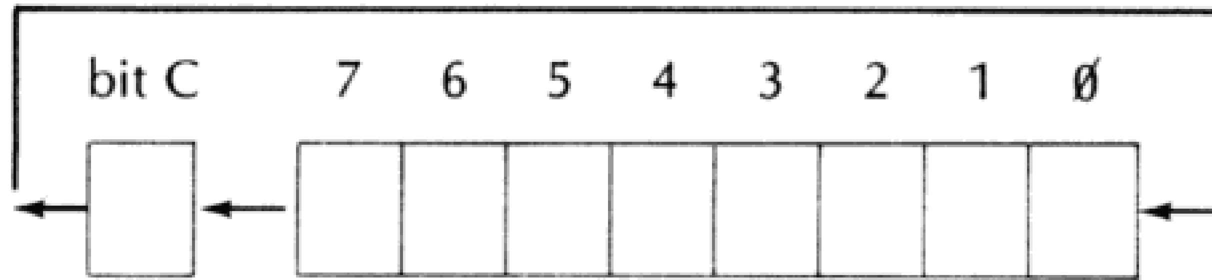
Chaque bit est décalé d'un rang vers la gauche. Le bit C devient le bit Ø tandis que le bit 7 se trouve propagé dans la retenue. Il y a donc la rotation :

Ø→1→2→3→4→5→6→7→C→Ø

Exemple: Soit l'instruction ROLA avec A = $9D et C=Ø initialement:

$9D = 1ØØ111Ø1 qui donne après rotation
$3A = ØØ111Ø1Ø et C=1

Plus généralement nous avons donc le schéma suivant:



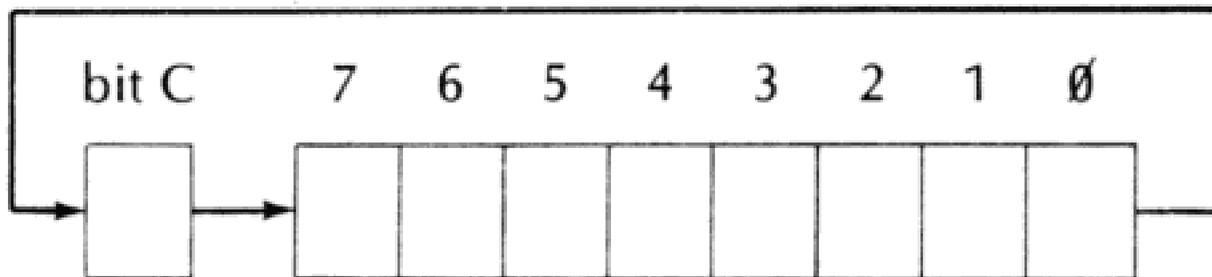
b) ROR

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=11ØØ111Ø et C=1

Plus généralement nous avons le schéma suivant:



5.4.6.2. Les instructions 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	direct	Ø9	N, Z, V, C
ROL	indexé	69	N, Z, V, C
ROL	étendu	79	N, Z, V, C

Considérons l'instruction ROLA avec A contenant la valeur $9D initialement.

Le contenu des registres internes du 68Ø9 est alors modifié de la façon suivante :

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$3A	B=$xx

DP=$xx

x x x x Ø Ø 1 1	CC

Remarque : Le bit V du registre CC correspond au "OU exclusif" des bits 6 et 7 (2 bits de poids fort) de l'opérande d'origine dans le cas des instructions de rotation vers la gauche.

Dans le cas d'une rotation vers la droite, ce bit V n'est pas affecté.

5.4.6.3. Les instructions ROR, RORA et RORB

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*
RORA	implicite	46	N, Z, C
RORB	implicite	56	N, Z, C
ROR	direct	Ø6	N, Z, C
ROR	indexé	66	N, Z, C
ROR	étendu	76	N, Z, 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 68Ø9 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 $A \wedge M \neq \emptyset$

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	immédiat	85	N, Z, V=Ø
BITA	direct	95	N, Z, V=Ø
BITA	indexé	A5	N, Z, V=Ø
BITA	étendu	B5	N, Z, V=Ø
BITB	immédiat	C5	N, Z, V=Ø
BITB	direct	D5	N, Z, V=Ø
BITB	indexé	E5	N, Z, V=Ø
BITB	étendu	F5	N, Z, V=Ø

Exemple: Soit l'instruction BITA ,X.

Le mode d'adressage utilisé est de type indexé à déplacement constant (nul en l'occurrence).

On suppose que le registre X contient la valeur $Ø15Ø et qu'à l'adresse $Ø15Ø se trouve la valeur $51. On suppose de plus que l'accumulateur contient la valeur $2A.

On a:

$51 = Ø1Ø1ØØØ1
$2A = ØØ1Ø1Ø1Ø

$51 ∧ $2A = ØØØØØØØØ = $ØØ

L'état des registres internes du 68Ø9 est donc affecté de la manière suivante:

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$ØØ	B=$xx

DP=$xx

x x x x Ø 1 Ø x	CC

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. Le bit V est mis à zéro.

Exemple: Soit l'instruction TSTA avec A = $B8 = 1Ø111ØØØ.

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=Ø
TSTB	implicite	5D	N, Z, V=Ø
TST	direct	ØD	N, Z, V=Ø
TST	indexé	6D	N, Z, V=Ø
TST	étendu	7D	N, Z, V=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.

On prend comme exemple :

23 = ØØ1ØØØ11 (en BCD)

avec :

23 = 2*1Ø + 3

Le problème est donc d'obtenir le "2", le "3" et de multiplier 2 par 1Ø. Or on sait que 1Ø=8+2 donc on pourra obtenir facilement le résultat escompté par des décalages adéquats.

Le listing du programme est le suivant :

```
LDA     # ØØ          ; registre DP=ØØ
TFR     A,DP
LDA     < $ØØ         ; charge chiffre BCD de poids faible (LSB)
ANDA    # %ØØØØ1111
STA     < $Ø1         ; sauvegarde en mémoire
LDA     < $ØØ         ; charge chiffre BCD de poids fort
AND     # %1111ØØØØ
LSRA                  ; A contient 8 fois MSB
STA     < $Ø2         ; sauvegarde de 1
LSRA                  ; A contient 4 fois MSB
LSRA                  ; A contient 2 fois MSB
ADDA    < $Ø2         ; A contient 1Ø fois MSB
ADDA    < $Ø1         ; A contient 1Ø fois MSB
STA     < $Ø2         ; range en mémoire le résultat
SWI
```

Nous avons utilisé ici des modes d'adressage directs en page zéro en chargeant initialement le registre DP avec la valeur $ØØ.

Le chiffre BCD à convertir est situé à l'adresse $ØØØØ initialement.

Le reste du programme n'amène aucun commentaire car son principe a été décrit ci-dessus.

5.5. LES INSTRUCTIONS SUR LE REGISTRE D'ÉTAT

Nous regrouperons sous cette appellation les instructions suivantes :

— l'instruction ANDCC,
— l'instruction ORCC,
— l'instruction CWAI.

5.5.1. L'instruction ANDCC

Cette instruction effectue un "ET" logique entre le registre de condition et une valeur binaire. Le seule mode d'adressage disponible est donc l'adresse immédiat. Cette instruction fonctionne exactement comme l'instruction AND.

Elle peut être utile pour positionner à "Ø" certains bits du registre de condition.

On a donc :

$CC \wedge N \rightarrow CC$

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
ANDCC	implicite	1C	tous (selon valeur de N)

Exemple: Supposons que nous voulions positionner à zéro les bits Ø à 3 du registre de condition CC.

Soit l'instruction ANDCC #$FØ avec CC=$12 initialement.

On a:

```
           $FØ = 1111ØØØØ
           $12 = ØØØ1ØØ1Ø
-----------------------------
$FØ ∧ $12 = ØØØ1ØØØØ = $1Ø
```

Les registres internes du 68Ø9 sont donc affectés de la manière suivante après l'exécution de cette instruction:

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$xx	B=$xx

DP=$xx

Ø Ø Ø 1 Ø Ø Ø Ø	CC

5.5.2. L'instruction ORCC

Cette instruction effectue un "OU" logique entre le registre de condition et une valeur binaire. Comme dans le cas de l'instruction ANDCC le seul mode d'adressage disponible est l'adressage immédiat. Cette instruction fonctionne exactement comme l'instruction ORA.

Elle peut être utile pour positionner à " 1 " certains bits du registre de condition.

On a donc :

$CC \vee N \rightarrow CC$

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
ORCC	implicite	1A	tous (selon valeur de N)

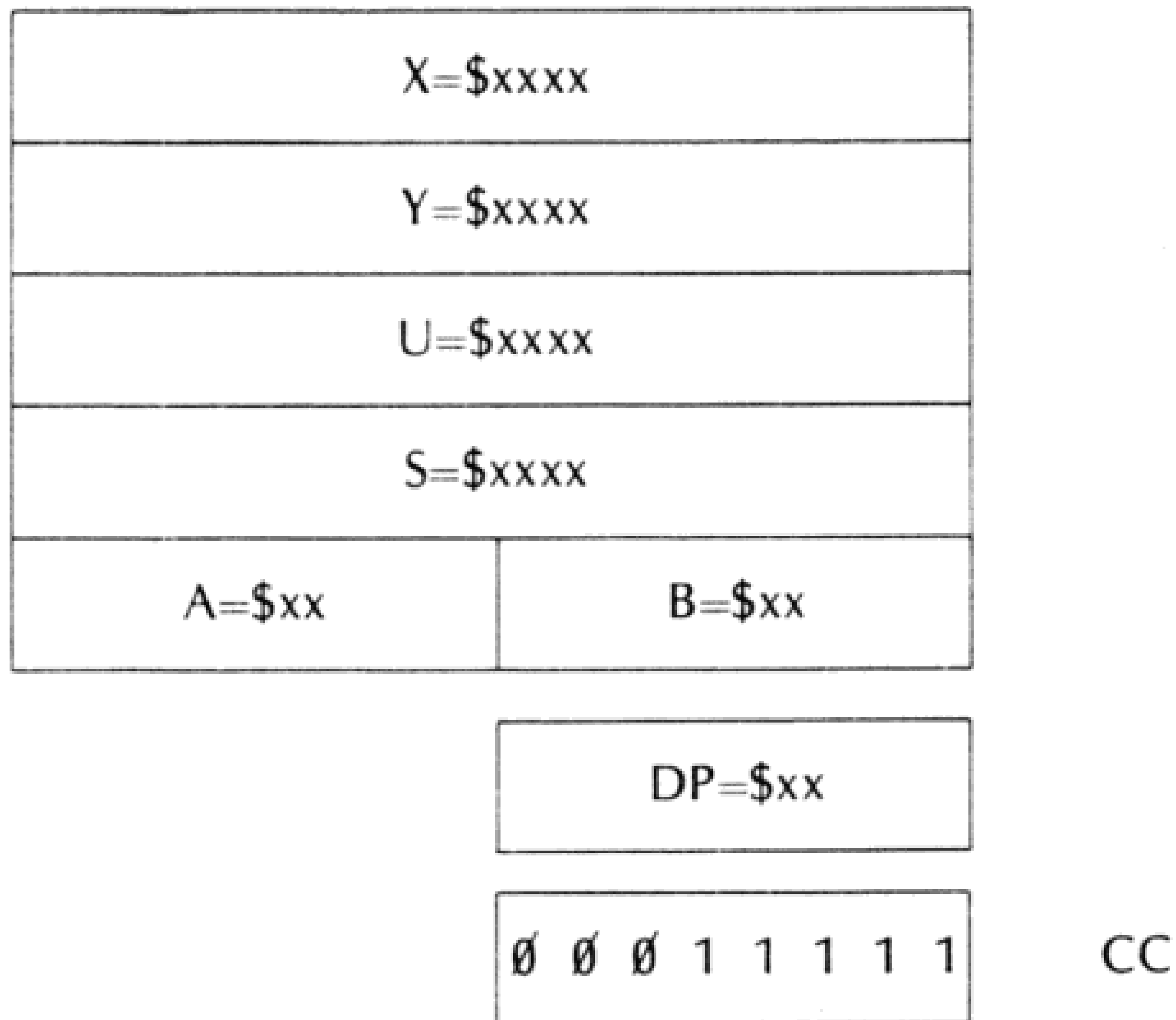
Exemple : Supposons que nous voulions positionner à 1 les bits Ø à 3 du registre de condition CC.

Nous écrirons par exemple ORCC #$ØF avec CC=$12 initialement.

On a :

```
        $ØF = ØØØØ1111
        $12 = ØØØ1ØØ1Ø
--------------------------
  $ØF ∨ $12 = ØØØ11111
```

Les registres internes du 68Ø9 sont affectés de la manière suivante :



5.5.3. L'instruction CWAI

Cette instruction permet de synchroniser le 68Ø9 sur un événement externe par le biais d'une interruption. Nous reviendrons sur la notion d'interruption un peu plus loin avec le paragraphe qui leur est consacré.

L'instruction CWAI fonctionne dans sa première phase comme l'instruction ANDCC décrite précédemment. Le 68Ø9 effectue donc un ET logique entre le contenu du registre de condition et une valeur binaire. Ceci a pour but de positionner à "Ø" certains bits de ce registre et en particulier les masques d'interruption. Ensuite le bit E de ce registre est positionné à "1" ce qui a pour but de provoquer une sauvegarde totale du contexte (état de tous les registres internes) dans la pile matérielle. Cet état est donc sauvegardé et le 68Ø9 se met en position d'attente d'interruption (il n'exécute donc plus d'instructions). Lorsqu'une interruption intervient, le 68Ø9 se branche à la routine de gestion d'interruptions et l'exécute. Lorsqu'il rencontre une instruction RTI (retour d'interruption), le contexte est restauré entièrement puisque le bit E du registre CC sauvegardé précédemment a été positionné à 1.

Si F=1 avant l'instruction CWAI, seule l'interruption IRQ pourra être activée. Par contre, si F=Ø, IRQ et FIRQ (interruption rapide) pourront être activées.

Ces différents types d'interruptions seront décrits un peu plus loin.

On a donc le fonctionnement suivant :

CC $\wedge$ N $\rightarrow$ CC
E = 1
pousse PC bas sur la pile
pousse PC haut sur la pile
pousse U bas sur la pile
pousse U haut sur la pile
pousse Y bas sur la pile
pousse Y haut sur la pile
pousse X bas sur la pile
pousse X haut sur la pile
pousse DP sur la pile
pousse B sur la pile
pousse A sur la pile
pousse CC sur la pile

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
CWAI	implicite	3C	tous (selon valeur de N)

Nous ne donnerons pas d'exemple pour cette instruction puisque son fonctionnement dépend énormément de l'architecture interne du micro-ordinateur autour duquel est bâti le 68Ø9.

5.6. LES INSTRUCTIONS DE COMPARAISON

Nous regrouperons sous cette appellation les instructions CMPA, CMPB, CMPD, CMPS, CMPU, CMPX, CMPY.

Les deux premières opèrent sur les accumulateurs A et B et portent donc sur des mots de 8 bits. Au contraire, les cinq suivantes portent sur les registres S, U, X et Y qui possèdent 16 bits chacun.

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 sous-

traction 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

D – M,M+1 pour l'instruction CMPD

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
CMPA	immédiat	81	N, Z, V, C
CMPA	direct	91	N, Z, V, C
CMPA	indexé	A1	N, Z, V, C
CMPA	étendu	B1	N, Z, V, C
CMPB	immédiat	C1	N, Z, V, C
CMPB	direct	D1	N, Z, V, C
CMPB	indexé	E1	N, Z, V, C
CMPB	étendu	F1	N, Z, V, C
CMPD	immédiat	1Ø 83	N, Z, V, C
CMPD	direct	1Ø 93	N, Z, V, C
CMPD	indexé	1Ø A3	N, Z, V, C
CMPD	étendu	1Ø B3	N, Z, V, C
CMPS	immédiat	11 8C	N, Z, V, C
CMPS	direct	11 9C	N, Z, V, C
CMPS	indexé	11 AC	N, Z, V, C
CMPS	étendu	11 BC	N, Z, V, C
CMPU	immédiat	11 83	N, Z, V, C
CMPU	direct	11 93	N, Z, V, C
CMPU	indexé	11 A3	N, Z, V, C
CMPU	étendu	11 B3	N, Z, V, C
CMPX	immédiat	8C	N, Z, V, C
CMPX	direct	9C	N, Z, V, C
CMPX	indexé	AC	N, Z, V, C
CMPX	étendu	BC	N, Z, V, C
CMPY	immédiat	1Ø 8C	N, Z, V, C
CMPS	direct	1Ø 9C	N, Z, V, C
CMPS	indexé	1Ø AC	N, Z, V, C
CMPS	étendu	1Ø BC	N, Z, V, C

Notons que dans le cas des instructions CMPA et CMPB le bit H du registre de condition CC prend une valeur indéterminée. Il conviendra donc de faire bien attention dans le cas de l'utilisation de nombres codés BCD.

Exemple d'utilisation: Soit l'instruction CMPA #$12.

On suppose que l'accumulateur A contient la valeur $3A.

Le 68Ø9 effectue alors la différence $3A – $12 = $3A + ($–12):

```
    $12 = ØØØ1ØØ1Ø
   $–12 = 111Ø111Ø
    $3A = ØØ111Ø1Ø

$3A–$12 = (1)ØØ1Ø1ØØØ
```

Le bit C est positionné à 1 tandis que les bits V, Z et N sont positionnés à Ø.

Les registres internes après l'exécution de l'instruction sont donc les suivants:

X=$xxxx	
Y=$xxxx	
U=$xxxx	
S=$xxxx	
A=$3A	B=$xx

DP=$xx

x x x x Ø Ø Ø 1	CC

Programme d'application: Il s'agit d'un programme qui compare deux chaînes de caractères ASCII. La longueur de ces deux chaînes de caractères est stockée à l'adresse page-zéro $5Ø. Les caractères sont

stockés en page-zéro à partir de $ØØ pour la première chaîne et à partir de $2Ø pour la seconde.

Le programme est le suivant :

```
        LDB    $FF      ; indicateur de chaînes différentes
        LDX    $ØØØØ    ; pointe sur la première chaîne
        LDY    $ØØ2Ø    ; pointe sur la deuxième chaîne
COMPT   LDA    ,X+      ; premier caractère de la première
                        ; chaîne
        CMPA   ,Y+      ; premier caractère de la deuxième
                        ; chaîne
        BNE    FIN      ; non égalité, terminé
        CMPX   $5Ø      ; dernier caractère?
        BNE    COMPT    ; non, recommence
        LDB    #ØØ      ; indicateur de chaînes égales
FIN     STB    $51
        SWI
```

Ce programme fonctionne de la manière suivante :

Le registre d'index X est initialement chargé avec la valeur $ØØØØ ce qui lui permet de pointer sur le 1er caractère de la première chaîne.

Le registre Y pointe de même sur le 1er caractère de la deuxième chaîne.

On utilise un mode d'adressage auto-incrémenté qui facilite l'écriture de boucles.

Le registre B est chargé avec une valeur qui indique si les deux chaînes de caractères sont identiques ou non.

B = $FF si les deux chaînes sont différentes
B = $ØØ si les deux chaînes sont égales

5.7. LES INSTRUCTIONS DE BRANCHEMENT

Il existe deux grandes catégories d'instructions de branchement :

— les instructions de branchement inconditionnel : BRA, LBRA, BRN, LBRN, BSR, LBSR 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 directs et étendus), sauf qu'en assembleur l'étiquette peut avoir 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

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
JMP	direct	ØE	aucun
JMP	indexé	6E	aucun
JMP	étendu	7E	aucun

Nous avons inclus sous cette rubrique les instructions BRA (branch always), LBRA (long branch always), BRN (branch never), LBRN (long branch never), BSR (branch to subroutine), LBSR (long branch to subroutine).

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 peut être codé sur 8 bits (pour accéder à un espace adressable de 256 octets) ou sur 16 bits (pour accéder à la totalité de l'espace adressable du 68Ø9).

Bien sûr, si le temps d'exécution est critique, il y a tout intérêt à utiliser le maximum d'instructions avec déplacement relatif de 8 bits.

Les instructions BRA, BRN, BSR appartiennent à la première catégorie tandis que les instructions LBRA, LBRN et LBSR, comme leur nom l'indique, appartiennent à la seconde. Bien sûr, le fonctionnement des instructions de branchement relatif est le même que le déplacement soit sur 8 ou 16 bits.

L'instruction BRA signifie "branchement dans tous les cas". Elle correspond à l'instruction JMP avec un adressage relatif.

On a donc:

PC + Deplacement → 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:

S – 1 → S; pousse PC bas sur la pile
S – 1 → S; pousse PC haut sur la pile
PC + déplacement → PC

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
BRA	relatif	2Ø	aucun
LBRA	relatif	16	aucun
BRN	relatif	21	aucun
LBRN	relatif	1Ø 21	aucun
BSR	relatif	8D	aucun
LBSR	relatif	17	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). Comme dans le cas des instructions de branchement inconditionnel, le branchement peut être codé sur 8 bits ou bien 16 bits.

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
BCC	relatif	24	aucun, condition C=Ø
LBCC	relatif	1Ø 24	aucun, condition C=Ø
BCS	relatif	25	aucun, condition C=1
LBCS	relatif	1Ø 25	aucun, condition C=1
BEQ	relatif	27	aucun, condition Z=1
LBEQ	relatif	1Ø 27	aucun, condition Z=1
BGE	relatif	2C	aucun, condition (N XOR V))=Ø
LBGE	relatif	1Ø 2C	aucun, condition (N XOR V))=Ø
BGT	relatif	2E	aucun, condition (Z Λ (N XOR V))=Ø
LBGT	relatif	1Ø 2E	aucun, condition (Z Λ (N XOR V))=Ø
BHI	relatif	22	aucun, condition C V Z=Ø
LBHI	relatif	1Ø 22	aucun, condition C V Z=Ø
BHS	relatif	24	aucun, condition C=Ø
LBHS	relatif	1Ø 24	aucun, condition C=Ø
BLE	relatif	2F	aucun, condition (Z V (N XOR V))=1
LBLE	relatif	1Ø 2F	aucun, condition (Z V (N XOR V))=1
BLO	relatif	25	aucun, condition C=1
LBLO	relatif	1Ø 25	aucun, condition C=1
BLS	relatif	23	aucun, condition C V Z=1
LBLS	relatif	1Ø 23	aucun, condition C V Z=1
BLT	relatif	2D	aucun, condition (N XOR V)=1
LBLT	relatif	1Ø 2D	aucun, condition (N XOR V)=1
BMI	relatif	2B	aucun, condition N=1
LBMI	relatif	1Ø 2B	aucun, condition N=1
BNE	relatif	26	aucun, condition Z=Ø
LBNE	relatif	1Ø 26	aucun, condition Z=Ø
BPL	relatif	2A	aucun, condition N=Ø
LBPL	relatif	1Ø 2A	aucun, condition N=Ø
BVC	relatif	28	aucun, condition V=Ø
LBVC	relatif	1Ø 28	aucun, condition V=Ø
BVS	relatif	29	aucun, condition V=1
LBVS	relatif	1Ø 29	aucun, condition V=1

Comme vous avez pu le constater le nombre de ces instructions est extrêmement important. De plus certaines d'entre elles semblent avoir des modes de fonctionnement similaires. C'est pourquoi nous allons essayer de clarifier un peu tout cela dans les lignes qui suivent.

Pour cela nous allons regrouper ces instructions de branchement en trois catégories :

a) celles qui traduisent une condition simple (par exemple dépendant 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 groupes simultanément. En effet l'égalité par exemple concerne les nombres signés ou non signés de manière similaire.

Nous allons examiner en détails les instructions appartenant à chacun de ces trois groupes.

a) Il s'agit des instructions BEQ, BNE, BMI, BPL, BCS, BCC, BVS et BVC ainsi que les instructions à branchement long associées.

— les instructions BEQ (branchement si égalité) et BNE (branchement si non égalité) testent la valeur du bit Z et donc l'égalité ou non à zéro. 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 du nombre considéré ;

— les instructions BCC (branchement si C=Ø) et BCS (branchement 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 68Ø9 un certain nombre d'instructions appartenant aux deux dernières catégories et qui utilisent certaines combinaisons de bits du registre de condition.

Bien entendu ces combinaisons sont traitées de façon interne par le 68Ø9 et l'utilisateur n'a à se soucier que du résultat.

b) Les instructions portant sur des nombres signés sont les suivantes : BGT, BLE, BGE, BLT, BEQ, BNE, ainsi que les instructions à branchement long associées.

— l'instruction BGT (branchement si supérieur à zéro) teste une inégalité stricte. En effet la condition détectée est :

Z ∧ (N XOR V)=Ø

Le test sur Z donne l'inégalité stricte. En effet, si Z=1 le résultat 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=$37 et B=$A8. Nous voulons tester si A est strictement inférieur à B.

On a :

```
     $37 = ØØ11Ø111
     $A8 = 1Ø1Ø1ØØØ
    $-A8 = Ø1Ø11ØØØ

$37-$A8 = 1ØØØ1111
```

Nous avons alors Z=Ø, N=1 et V=1 ce qui donne Z Λ (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 Λ (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 au sens 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 soit 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 BEQ et BNE), une inégalité stricte (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.

L'instruction BLO est le contraire de BHS (A<B) et teste donc la condition simple C=1.

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 :

S–1 → S ; PC bas → (S)
S–1 → S ; PC haut → (S)
EA → PC

Lors d'une instruction JSR le pointeur de pile système 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, l'incrémente et la charge ensuite dans le compteur ordinal.

On a donc les opérations suivantes :

(S) → PC haut ; S–1 → S
(S) → PC bas ; S–1 → S

Lors d'une instruction RTS, le pointeur de pile système est donc incrémenté de deux unités.

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
JSR	direct	9D	aucun
JSR	indexé	AD	aucun
JSR	étendu	BD	aucun
RTS	implicite	39	aucun

Exemple: Soit l'instruction JSR (,X).

Nous sommes en présence d'un mode d'adressage indirect indexé par rapport à X et à déplacement nul. Nous supposons que X contient la valeur $53F8 et qu'aux adresses $53F8 et $53F9 résident les valeurs $2Ø et $67. Ceci permet donc de pointer sur l'adresse $672Ø qui sera chargée dans le registre PC du 68Ø9.

Nous allons vous proposer un exemple de programme faisant appel à un sous-programme: il s'agit de trouver le plus grand élément d'une table de valeurs. On place dans le registre B la longueur du bloc à scruter et dans X l'adresse de début du bloc.

Le résultat sera sauvegardé à l'adresse $ØØØØ.

```
        LDB   LONG      ; charge longueur de la table
        LDX   ADRES     ; charge adresse de début
        JSR   MAX       ; recherche du max
        STA   $ØØØØ     ; sauvegarde du résultat
        SWI

MAX     LDA   #$ØØ      ; valeur maximale=Ø
COMP    CMP   B,X       ; nouvelle valeur=maximum?
        BGE   SUIT      ; non, recommence
        LDA   B,X       ; oui, charge ce nouveau maximum
SUIT    DECB            ; bloc terminé?
        BNE   COMP      ; non, continue
        RTS
```

5.9. LES INSTRUCTIONS SUR LA PILE

Elles sont au nombre de quatre et comprennent les instructions d'empilement et de dépilement.

— empilement PSHS et PSHU,

— dépilement PULS et PULU.

5.9.1. Fonctionnement des instructions d'empilement

Ces instructions permettent de stocker le contenu d'un ou plusieurs registres internes du 68Ø9 au sommet de la pile. Cette pile peut être soit la pile matérielle S soit la pile utilisateur U.

Grâce à ces instructions les registres PC, U, S, Y, X, DP, B, A, CC peuvent être sauvegardés sur la pile. La syntaxe de l'instruction d'empilement indique au microprocesseur les registres concernés. C'est ainsi que pour sauvegarder les registres A, B, CC, on écrira :

```
PSHU A,B,CC
```

ou bien :

```
PSHS A,B,CC
```

Notons que l'instruction portant sur le registre S permet d'empiler le registre U tandis que l'instruction portant sur le registre U permet d'empiler le registre S. Il n'est donc pas possible d'empiler le registre S à l'aide de l'instruction PSHS par exemple.

Le microprocesseur connaît les registres qu'il doit transférer sur la pile grâce à un octet supplémentaire suivant le code-opération. Selon le positionnement des bits Ø à 7 de cet octet, les registres correspondant seront affectés.

Ainsi si :

bit Ø=1 :	S–1 → S	et	CC	→	(S)
bit 1=1 :	S–1 → S	et	A	→	(S)
bit 2=1 :	S–1 → S	et	B	→	(S)
bit 3=1 :	S–1 → S	et	DP	→	(S)
bit 4=1 :	S–1 → S	et	X bas	→	(S)
	S–1 → S	et	X haut	→	(S)
bit 5=1 :	S–1 → S	et	Y bas	→	(S)
	S–1 → S	et	Y haut	→	(S)
bit 6=1 :	S–1 → S	et	U bas	→	(S)
	S–1 → S	et	U haut	→	(S)
bit 7=1 :	S–1 → S	et	PC bas	→	(S)
	S–1 → S	et	PC haut	→	(S)

Les lignes ci-dessous sont valables pour l'instruction PSHS. Le même fonctionnement peut bien sûr être observé pour l'instruction

PSHU, sachant que cette fois-ci c'est la pile utilisateur qui est concernée et que le registre S sera sauvegardé quand le bit 6 sera positionné à 1.

5.9.2. Fonctionnement des instructions de dépilement

Ces instructions permettent de charger un ou plusieurs registres internes du 68Ø9 à l'aide d'octets stockés dans la pile (pile système S ou pile utilisateur U).

Grâce à ces instructions les registres PC, U, S, Y, X, DP, B, A, CC peuvent être chargés à partir de la pile. La syntaxe de l'instruction de dépilement indique au microprocesseur les registres concernés. C'est ainsi que pour restaurer les registres A, B, CC, on écrira :

```
PULU A,B,CC
```

ou bien :

```
PULS A,B,CC
```

Comme dans le cas des instructions d'empilement le microprocesseur connaît les registres qu'il doit transférer sur la pile grâce à un octet supplémentaire suivant le code-opération. Selon le positionnement des bits Ø à 7 de cet octet, les registres correspondant seront affectés. Son fonctionnement est exactement le même que pour les instructions d'empilement.

5.9.3. Les instructions PSHU, PSHS, PULS

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
PSHS	immédiat	34	aucun
PSHU	immédiat	36	aucun
PULS	immédiat	35	aucun
PULU	immédiat	37	aucun

5.10. LES INSTRUCTIONS SPÉCIALES

Il s'agit des instructions suivantes :

— l'instruction NOP,

— l'instruction RTI,

— les instructions SWI, SWI2, SWI3,

— l'instruction SYNC.

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'entres 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 ;

— 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 débarassé de ces instructions inutiles afin d'en diminuer le temps d'exécution.

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
NOP	implicite	12	aucun

5.10.2. L'instruction RTI

En pratique, vous n'aurez probablement jamais à l'utiliser au même titre que d'autres instructions telles que CWAI, SYNC, etc...

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 68Ø9 trois 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 68Ø9, 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.

De plus le bit E du registre de condition CC est mis à un avant son chargement sur la pile système pour indiquer que l'état complet du processeur a été sauvegardé (dans l'ordre registres PC bas, PC haut, U bas, U haut, Y bas, Y haut, X bas, X haut, DP, B, A, CC).

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 donné par le contenu des adresses $FFF8 et $FFF9.

Comme dans le cas de l'interruption non masquable NMI, le bit E est positionné à 1 avant que le contenu de tous les registres (sauf S) soit sauvegardé dans la pile système.

— *L'interruption rapide FIRQ*

De même que la précédente, cette interruption peut être masquée mais cette fois-ci par l'intermédiaire du bit F (bit 6) du registre de condition CC.

L'adresse de départ de la routine de traitement des interruptions rapides est donnée par le contenu des cases-mémoire d'adresses $FFF6 et $FFF7.

Cette interruption est appelée "rapide" car ici le bit E est positionné à Ø de manière à indiquer que seuls les registres PC bas, PC haut et CC sont sauvegardés dans la pile système.

Les routines de traitement des interruptions se terminent toutes par une instruction RTI. Lorsque le 68Ø9 rencontre cette instruction, il teste tout d'abord la valeur du bit E du registre de condition qui est restauré en premier. S'il est à un alors les registres internes sont chargés les uns après les autres dans l'ordre CC, A, B, DP, X haut, X bas, Y haut, Y bas, U haut, U bas, PC haut, PC bas.

Par contre s'il est à zéro seuls les registres CC, PC haut et PC bas sont restaurés.

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
RTI	implicite	38	tous (registre CC restauré)

5.10.3. Les instructions SWI, SWI2, SWI3

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 :

S–1 → S PC bas → (S)
S–1 → S PC haut → (S)
S–1 → S U bas → (S)
S–1 → S U haut → (S)
S–1 → S Y bas → (S)
S–1 → S Y haut → (S)
S–1 → S X bas → (S)
S–1 → S X haut → (S)
S–1 → S DP → (S)
S–1 → S B → (S)
S–1 → S A → (S)
S–1 → S CC → (S)

Puis selon l'instruction considérée les actions suivantes ont lieu :

a) Instruction SWI

Avant la sauvegarde des registres le bit E du registre CC est positionné à 1 pour indiquer que l'état total du 68Ø9 est sauvegardé dans la pile.

Après la sauvegarde de ces registres, les bits I et F du registre de condition sont positionnés à 1 (ce qui signifie que les interruptions matérielles sont interdites) et 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 en contient l'octet de poids fort.

b) Instruction SWI2

Le fonctionnement de cette instruction est similaire à la précédente sauf que celle fois-ci les masques d'interruption I et F ne sont pas affectés et que le branchement est donné par le contenu des cases-mémoire d'adresses $FFF4 et $FFF5.

Les interruptions matérielles sont donc autorisées lors de l'exécution de la routine de traitement de l'interruption logicielle SWI2. On dira que SWI2 a une priorité inférieure à SWI.

c) Instruction SWI3

Son fonctionnement est similaire à SWI2 sauf que l'adresse de branchement est donnée par le contenu des cases-mémoire d'adresses $FFF2 et $FFF3.

Selon la valeur des vecteurs de branchement, il y aura appel du programme moniteur du système, du Basic, ou de tout autre programme. Ces instructions sont très utilisées pour la mise au point des programmes car elles permettent d'en arrêter le déroulement la ou on le désire.

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
SWI SWI2 SWI3	implicite implicite implicite	3F 1Ø 3F 11 3F	aucun aucun aucun

5.10.4. L'instruction SYNC

Cette instruction permet de synchroniser le 68Ø9 sur un événement extérieur. Ceci peut être utile par exemple dans le cas d'une application biprocesseur ou 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 68Ø9 rencontre une instruction SYNC, il s'arrête et attend qu'une interruption se produise. Cette interruption peut avoir été interdite par le biais de l'un des indicateurs I ou F. Dans ce cas, lorsqu'elle apparaît le 68Ø9 reprend son programme et exécute les instructions suivant immédiatement le SYNC. Dans le cas contraire (interruption autorisée) il y a exécution de la routine d'interruption classique se terminant bien sûr par un RTI.

Instruction	*Mode d'adressage*	*Code Opération*	*Indicateurs affectés*
SYNC	implicite	13	aucun

6
Les entrées-sorties

6.1. GÉNÉRALITÉS

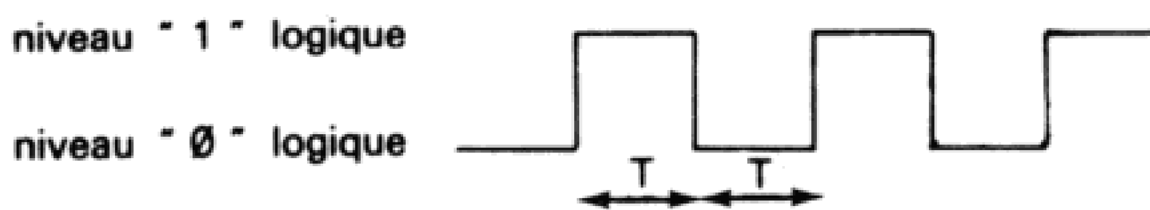
Nous avons vu jusqu'à présent des instructions qui permettaient d'effectuer des transferts entre Mémoire et Registres et des opérations sur ceux-ci. Le problème est qu'il est nécessaire également de communiquer avec l'extérieur : c'est ce que nous appellerons les Entrées-Sorties. Les entrées permettent au microprocesseur de recevoir des données de l'extérieur (clavier, disque, carte d'acquisition de données, etc.). Les sorties permettent au microprocesseur d'envoyer des données vers l'extérieur (écran, disque, imprimante).

Le 68Ø9, contrairement à d'autres microprocesseurs, ne possède pas d'instructions spécialisées pour les Entrées-Sorties ce qui ne veut pas dire qu'il n'en a pas, bien au contraire. Elles sont traitées comme de "vulgaires" cases-mémoire placées dans l'espace adressable du microprocesseur. On peut donc utiliser avec elles toutes les instructions que nous avons rencontrées.

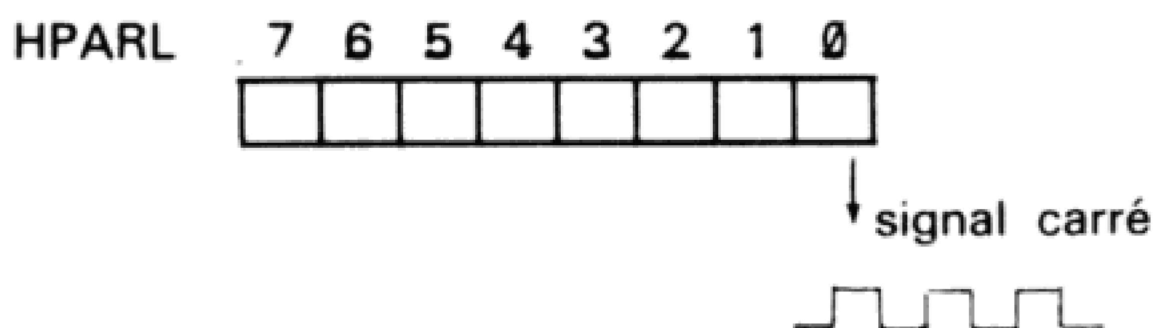
Plutôt que de faire des discours abstraits nous allons examiner quelques exemples.

1) Supposons que nous voulions générer un son dans un haut-parleur. Avant toute chose nous devons connaître son adresse en mémoire (soit HPARL cette adresse).

Pour sortir ce son il nous suffit d'envoyer à cette adresse un signal carré qui a la forme suivante :

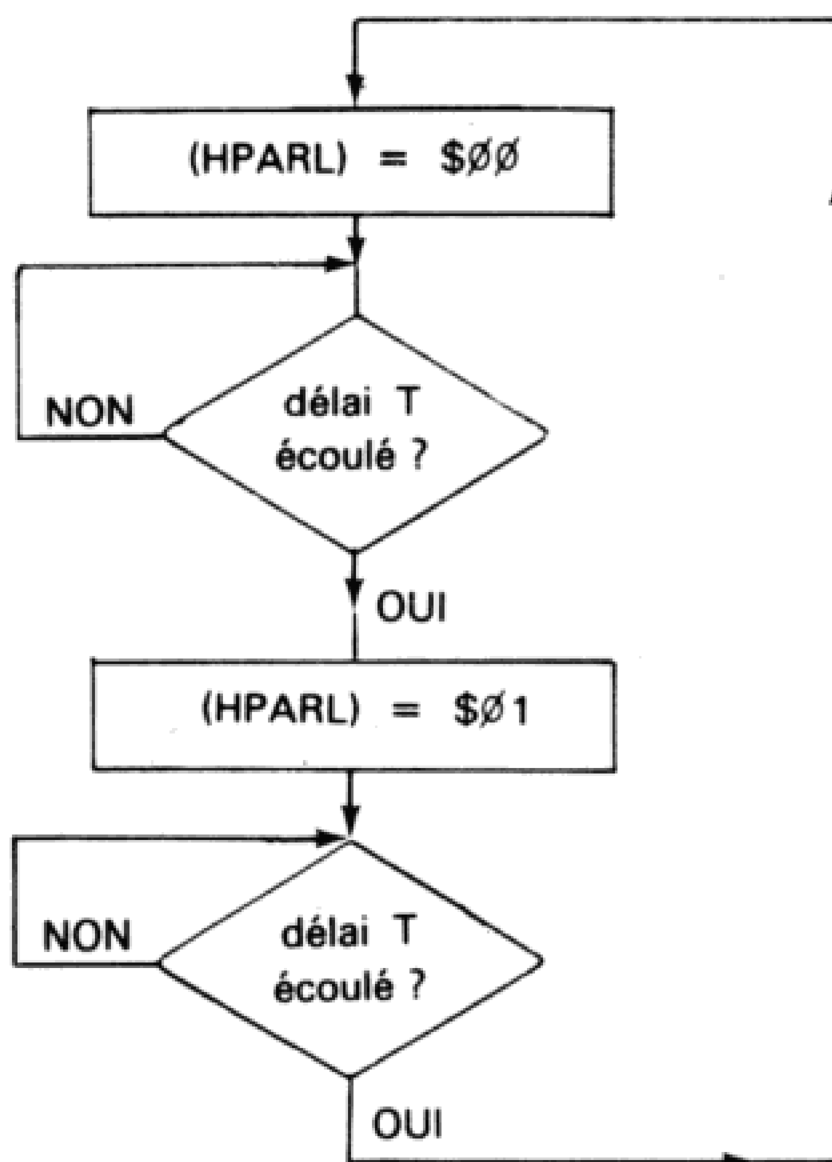


C signal pourra être présent par exemple sur le bit Ø de l'octet présent à l'adresse HPARL.



Ce bit prendra donc successivement les valeurs "Ø" et "1" logiques chaque fois pendant la durée T.

L'organigramme d'un tel dispositif est le suivant :



Nous allons vous présenter deux méthodes pour écrire ce programme.

1re méthode : On utilise un sous-programme pour générer le délai de durée T :

```
DEBUT   LDA   #$ØØ         ; sortie haut-parleur=Ø
        STA   HPARL
        JSR   DELAI        ; reste à cet état pendant T
        LDA   #$Ø1         ; sortie haut-parleur=1
        STA   HPARL
        JSR   DELAI        ; reste à cet état pendant T
        JMP   DEBUT        ; recommence
DELAI   LDB   #$AØ         ; initialisation de T
COMPT   DECB               ; fini ?
        BNE   COMPT        ; non, continue
        RTS                ; oui, change l'état du haut-parleur
```

Ce programme est très simple et n'amène aucun commentaire. La valeur **$AØ** chargée dans B permet d'obtenir une fréquence audible qui bien sûr dépend de la fréquence d'horloge du 68Ø9.

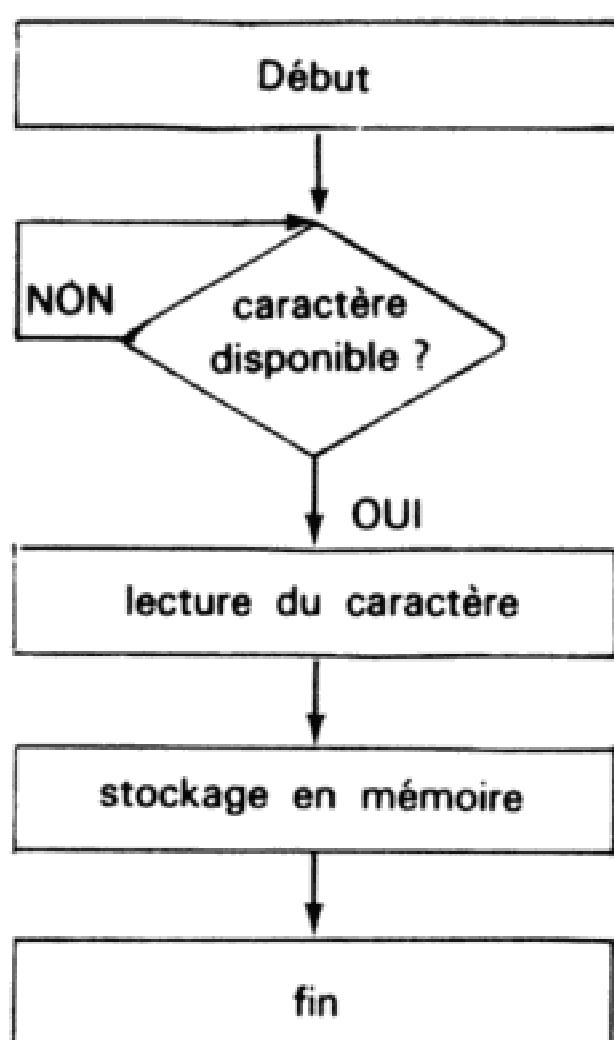
2e méthode : Ici nous n'utilisons pas de sous-programme. On se base sur la propriété suivante $X \veebar 1 = \overline{X}$ ("OU exclusif"). Le programme est le suivant :

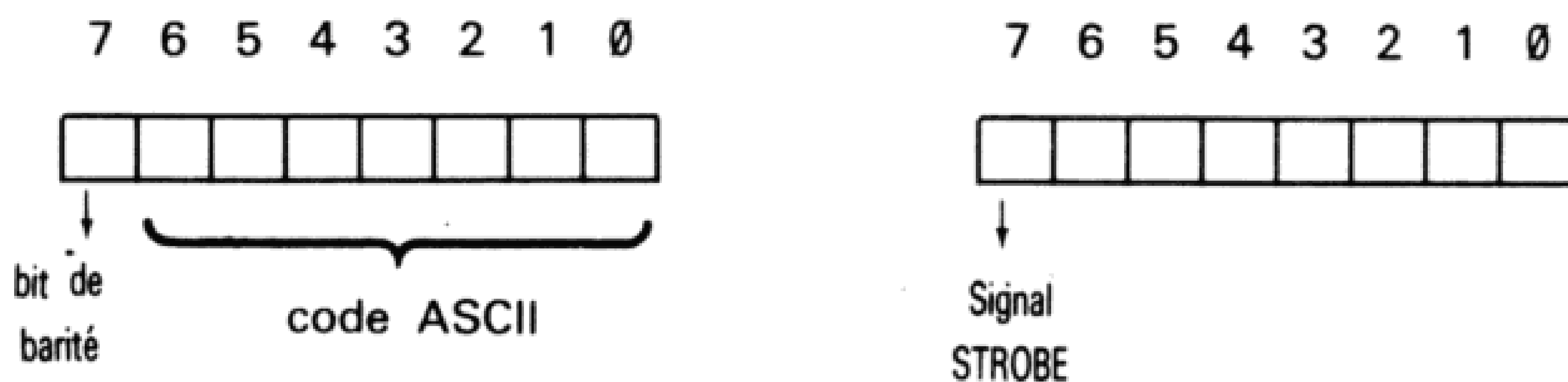
```
        LDA   #$ØØ         ; met le haut-parleur à zéro
CHANGE  EORA  #%ØØØØØØØ1   ; change cet état
        STA   HPARL        ; transfert vers le haut-parleur
        LDB   #$AØ         ; initialisation de T
COMPT   DECB               ; fini ?
        BNE   COMPT        ; non, continue
        JMP   CHANGE       ; oui, change à nouveau l'état du
                           ; haut-parleur
```

2) Un des organes d'Entrées-Sorties les plus communs est le clavier. Nous allons examiner ici une routine de scrutation d'un tel périphérique.

Beaucoup de claviers utilisés dans les micro-ordinateurs sont encodés et donc délivrent après chaque pression de touche le code ASCII correspondant (cela nous donne donc 7 bits pour le code ASCII et 1 bit de parité pour la détection d'erreurs). En plus de cela un neuvième bit délivre un signal appelé STROBE indiquant qu'un caractère est disponible.

Le programme qui suit va lire un caractère et le transférer en mémoire. L'organigramme est le suivant :





On suppose que l'adresse CARACT contient le code ASCII du caractère ainsi que son bit de parité. De plus le contenu de l'adresse STROBE nous donne le signal du même nom. Le code ASCII du caractère débarassé de son bit de parité, est stocké à l'adresse RANGE.

Le programme est le suivant :

```
SCRUT   TST    STROBE      ; caractère disponible ?
        BLT    SCRUT       ; non recommence
        LDA    CARACT      ; oui, chargement dans A
        ANDA   #%Ø1111111  ; masque de bit de parité
        STA    RANGE       ; rangement en mémoire
        SWI
```

Vous pouvez remarquer que nous avons utilisé ici l'instruction TST. Elle permet de détecter très facilement la présence ou non du signal STROBE : (STROBE)=1XXXXXXX, pas de caractère disponible, (STROBE)=ØXXXXXXX, caractère disponible.

L'instruction TST positionne le bit N du registre de condition CC en fonction du bit 7 de la case-mémoire STROBE. L'instruction BLT permet un branchement à l'adresse SCRUT si N=1 donc si STROBE=1. Sinon le caractère est rangé en mémoire après avoir été débarassé de son bit de parité par un "ET" logique.

Nous avons exposé ici quelques idées générales concernant les Entrées-Sorties avec de petits exemples à l'appui.

Bien entendu nous ne prétendons pas vous avoir tout dit sur ce sujet. En effet pour aller plus loin nous devrions rentrer dans la structure d'un micro-ordinateur et décortiquer le matériel, ce qui sortirait du cadre de cet ouvrage.

Notons tout de même qu'il existe des composants périphériques spécialisés dans les Entrées-Sorties (PIA : Peripheral Interface Adaptor) qui se connectent facilement au 68Ø9 et qui permettent de remplir des fonctions complexes telles que ports d'Entrées-Sorties programmables ou même Timers, générateurs d'impulsions, etc... Certains micro-ordinateurs de conception récente en sont équipés. Ceci dit leur utilisation peut paraître un petit peu difficile à comprendre à un novice.

7

La mise au point d'un programme en assembleur

Jusqu'à présent nous avons aligné des instructions les unes après les autres, le résultat étant des programmes qui "tournent" et que vous pourrez tester vous-même sur votre micro-ordinateur. Mais le problème est qui arrivera bien un jour (nous espérons d'ailleurs dès maintenant) où il faudra vous-même vous mettre au travail et "pondre" un programme de votre cru. Au début cela ira probablement bien, vous alignerez les instructions du mieux que vous pourrez et puis viendra la phase de l'assemblage. C'est alors que, oh douleur atroce, vous verrez très certainement s'afficher un assez grand nombre d'erreurs (à condition tout de même que votre programme comporte un peu plus que trois instructions).

A supposer que vous arriviez à les supprimer (ce dont nous ne doutons pas puisque vous avez lu attentivement les chapitres précédents), vous essaierez de lancer votre programme et vous vous apercevrez peut-être qu'il ne "tourne" pas, ou qu'il boucle indéfiniment. Vous en serez arrivé à la phase la plus délicate de l'écriture d'un programme : sa mise au point.

Car il faudra bien vous rendre à l'évidence qu'un programme (de taille suffisante bien entendu) qui tourne du premier coup n'existe pas. Et si cela vous arrive un jour, dites-vous que c'est l'exception qui confirme la règle.

Le but de ce chapitre est de vous aider à réduire au maximum cette phase de mise au point en vous indiquant la manière de travailler, depuis le moment où l'idée d'un programme germe dans votre esprit jusqu'au moment où il marche parfaitement dans toutes les configurations imaginables.

Nous pouvons distinguer plusieurs étapes dans la rédaction d'un programme :

1) position du problème,

2) l'organigramme,

3) l'écriture proprement dite,

4) le debugging.

Nous allons étudier chacune de ces étapes séparément.

1) Position du problème

Cela vous paraîtra peut-être une "La palissade" mais il faut tout d'abord savoir exactement ce que l'on veut, c'est-à-dire quelle (ou quelles) fonctions le programme devra réaliser, quelles Entrées-Sorties il utilisera (clavier, écran, Bip-Bip sonore, etc.).

Cette phase est peut-être un peu astreignante mais elle oblige à avoir les idées claires et permet d'aborder plus sereinement la suite.

2) L'organigramme

Un organigramme est tout à fait similaire dans le cas d'un programme assembleur que dans le cas d'un programme BASIC.

Il sert à poser clairement les problèmes sur le papier et se trouve à mi-chemin entre les phases "Position du Problème" et "Écriture proprement dite".

Un organigramme clair et correctement conçu doit pouvoir être traduit en une suite de mnémoniques pratiquement sans papier. Nous allons maintenant vous donner quelques règles utiles qu'il vous faudra essayer de respecter au maximum si vous voulez accroître votre efficacité.

* Si votre programme est d'une longueur suffisante (quelques dizaines à quelques centaines d'octets, voire plus) il vous faudra tout d'abord le diviser en plusieurs parties, chacune d'elles réalisant une ou plusieurs fonctions élémentaires. Ces différentes parties pourront être des sous-programmes indépendants, ceci afin de pouvoir les tester séparément lors de la phase de mise au point.

* En dessinant votre organigramme procédez de manière systématique et logique. Évitez au maximum les "astuces géniales" qui peuvent ne pas être comprises par d'autres et que vous-mêmes ne comprendrez peut-être pas dans quelques mois lorsque vous voudrez apporter une amélioration à votre programme.

Dans la plupart des cas le microprocesseur (68Ø9) qui équipe votre machine va très vite et vous ne vous apercevrez pas du ralentissement occasionné par le rajout de quelques instructions en plus. Et puis on perd souvent un temps énorme à vouloir diminuer la longueur des programmes.

* Évitez absolument durant cette phase d'aligner des instructions sur le papier même si la tentation est forte. Ce n'est que lorsque l'organigramme sera terminé et testé mentalement pas à pas par tous les chemins imaginables que vous pourrez passer à la phase suivante qui est (enfin !) l'écriture proprement dite du programme. Et puis rappelez-vous ce que vous disait probablement votre "prof de maths" dans votre jeunesse : une figure est plus parlante que des calculs. L'organigramme est là pour représenter graphiquement ce que le programme réalisera par la suite.

* Pour rentrer un peu plus dans les détails, nous allons énoncer une règle d'or : il est nécessaire de concevoir son programme d'une façon structurée, séquence après séquence, ceci afin d'éviter que plusieurs sources d'erreurs puissent interférer les unes entre les autres. Chaque séquence pourra être une seule ou bien un groupe d'instructions qui devra si possible avoir une entrée et une sortie.

Exemple : Deux nombres sont rangés en mémoire (entrée), une séquence d'instructions en calcule le produit (sortie).

Dans chaque séquence essayez au maximum d'adopter une démarche logique et structurée. Le dernier point nous semble extrêmement important c'est pourquoi nous allons nous y attarder quelque peu.

De plus en plus de langages de haut niveau, et même certains assembleurs offrent aujourd'hui la possibilité d'une programmation structurée. Dans ce cas un programme n'est plus (comme c'est le cas en BASIC et généralement en Assembleur) une suite d'instructions, mais plutôt un ensemble de "blocs" interagissant les uns avec les autres. Un bon exemple d'un tel langage est PASCAL qui devient d'année en année plus populaire parmi les utilisateurs de micro-ordinateurs.

Nous allons énoncer ci-dessous quelques idées générales permettant de programmer en Assembleur d'une manière un peu similaire, ceci afin de faciliter l'écriture du code, sa mise au point, sa lecture, en même temps que sa rapidité d'exécution.

En programmation structurée les instructions de branchement inconditionnel (JMP dans le cas du 68Ø9) sont à éviter dans toute la mesure du possible. Ces instructions étant les analogues du GOTO en BASIC, ce sont elles qui sont pourtant les plus aisément assimilables, à tort cependant.

En effet les micro-ordinateurs sont des machines d'usage général capables de communiquer avec l'extérieur. Pour cette raison ils sont amenés à exécuter des séquences d'instructions différentes suivant certaines informations venant de l'extérieur. C'est pourquoi les instructions de branchement conditionnel sont très adaptées pour ce type de tâches (sélection de deux différentes séquences d'instructions à l'intérieur d'un programme). De plus ces instructions permettent la création aisée de boucles (par simple branchement au début d'une même séquence un nombre répété de fois).

L'implémentation de ces différents concepts peut bien sûr *toujours* être effectuée à l'aide d'instructions de branchement inconditionnel. Cela se traduit en général par des programmes plus longs, donc moins efficaces au point de vue temps exécution.

Par contre l'utilisation d'instructions de branchement conditionnel judicieusement choisies (et vous avez pu constater qu'à ce point de vue le 68Ø9 est extrêmement riche) permet d'obtenir des programmes possédant une structure lisible dès le premier coup d'œil.

Il existe en programmation structurée trois formes extrêmement employées qui sont les suivantes :

1) IF...THEN...ELSE
2) DO...WHILE
3) REPEAT...UNTIL

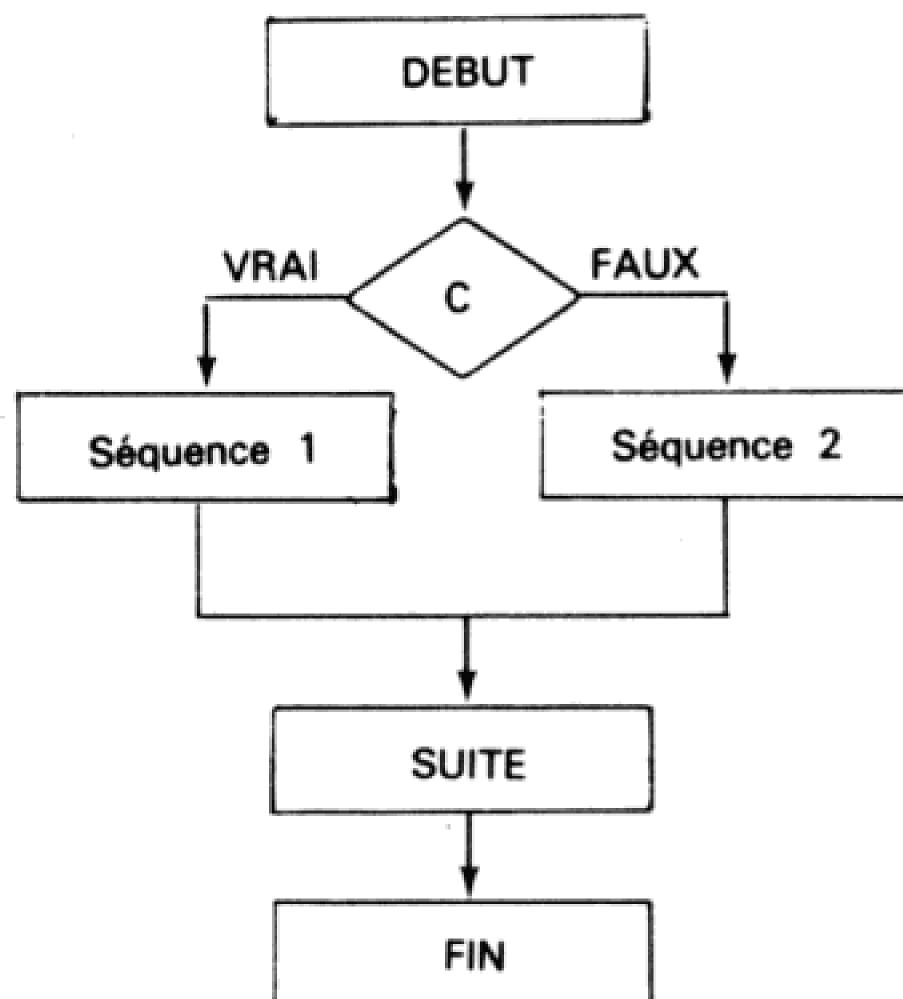
La première permet d'obtenir une exécution conditionnelle à l'intérieur d'un programme tandis que les deux dernières permettent la création de boucles.

Nous allons examiner successivement ces trois formes en donnant pour chacune d'elle l'organigramme correspondant ainsi qu'un exemple d'exécution.

a) IF...THEN...ELSE

Cette instruction est bien connue de tous les possesseurs de micro-ordinateurs programmant en Basic (bien que souvent le "ELSE" ne soit pas implémenté).

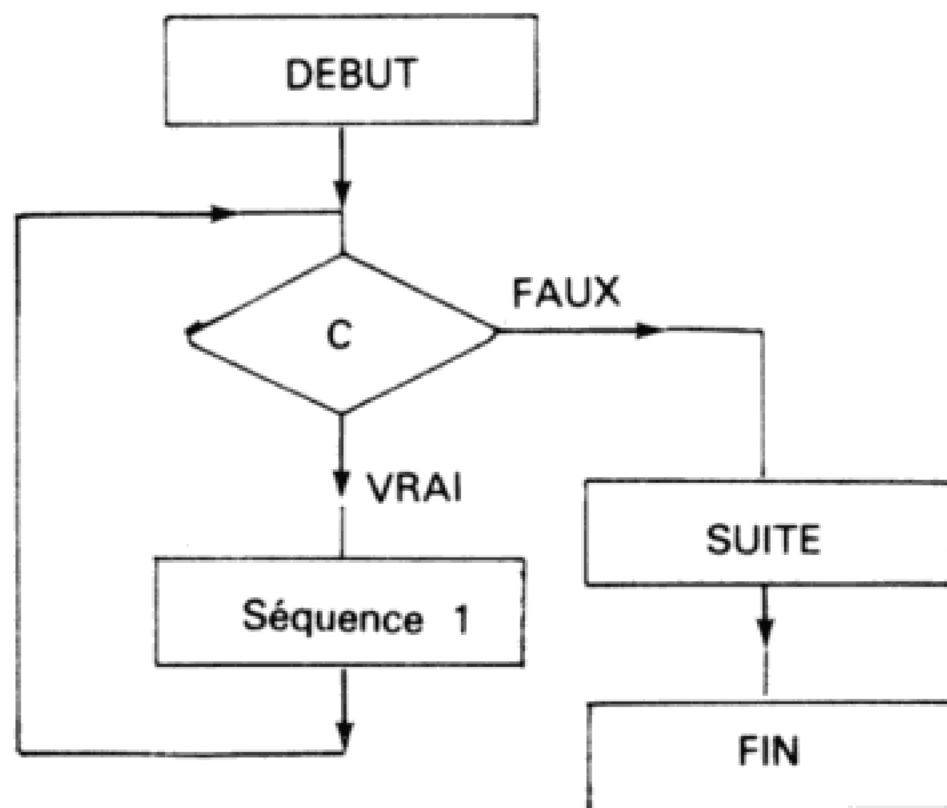
Traduisons ceci en bon français : si la condition C est réalisée alors on exécute la séquence 1, sinon on exécute la séquence 2.



Un exemple de programme implémentant cette structure est le suivant :

```
        LDA   VAR        ; charge VAR
        CMPA  #$00
        BLT   TEST1      ; si négatif, exécute PROG1
        JSR   PROG2
        BRA   TEST2      ; si positif, exécute PROG2
TEST1   JSR   PROG1
TEST2   SWI
```

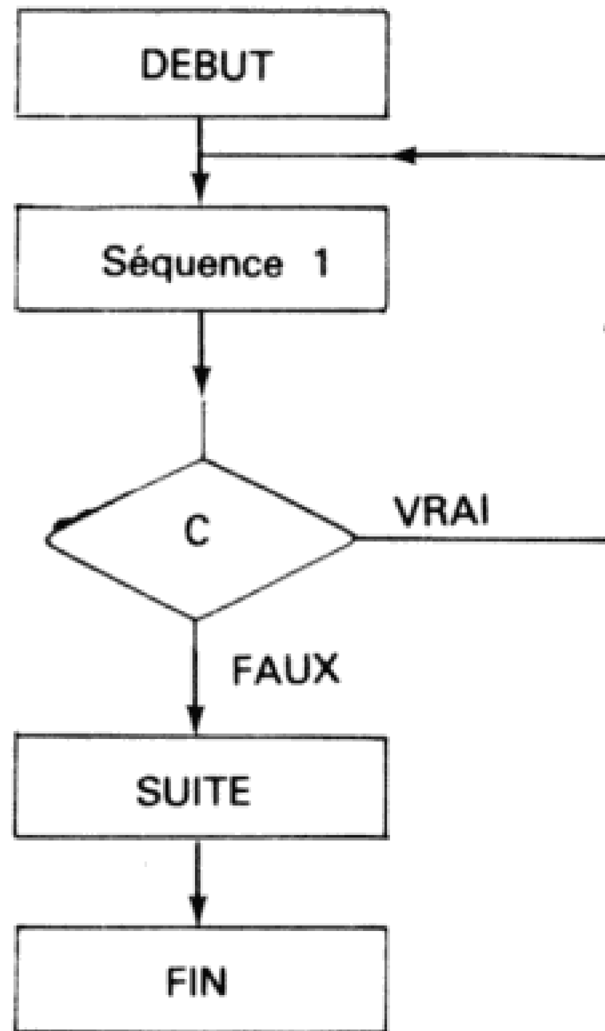
b) DO...WHILE, ce qui veut dire en bon français: exécuter la séquence 1 pendant que la condition C est réalisée.



Un exemple de programme implémentant cette structure est le suivant:

```
                     LDA    VAR
TEST1                CMPA   #$00
                     BLE    TEST2
                   { .
séquence 1         { .
d'instructions     { .
                   { .
                     BRA    TEST1
TEST2                SWI
```

c) DO...UNTIL, ce qui veut dire: exécuter la séquence 1 tant que la condition C est réalisée.



Un exemple de programme utilisant cette structure est le suivant :

```
                              LDA      VAR
          TEST1
                              .
        séquence 1
      d'instructions          .
                              .
                              CMPA     # $ØØ
                              BGT      TEST1
                              .
                              .
                              .
```

3) L'écriture du programme proprement dite

Ceci est pratiquement la partie la plus rapide dans l'élaboration d'un programme en assembleur si bien sûr on connaît suffisamment le jeu d'instructions.

N'oubliez surtout pas d'écrire les commentaires qui vous seront bien utiles lors de la phase mise au point ou plus tard quand vous voudrez modifier votre programme.

4) Le Debugging (en français déverminage)

Sous ces deux mots un peu barbares se cache une chose bien simple : il s'agit tout simplement de transformer un programme truffé d'erreurs (dans la syntaxe mais aussi dans la conception) en un programme qui "tourne".

Il existe de nombreux outils mis à la disposition des programmeurs pour mettre au point des programmes. Nous n'aborderons pas ici les machines spécialisées qui sont souvent de véritables ordinateurs et donc destinés aux professionnels.

Dans les machines que le particulier peut posséder il y a en général deux outils qui facilitent le debugging.

a) Le pas-à-pas

Le pas-à-pas permet, comme son nom l'indique, d'exécuter une seule instruction à la fois dans un programme en assembleur.

Il vous permettra de corriger des erreurs grossières telles que : confusions de mnémoniques, erreurs de branchement, erreurs dans les opérandes.

b) Les points d'arrêts

Nous avons vu que les instructions SWI étaient des interruptions logicielles et permettaient donc d'arrêter le déroulement d'un programme là où on le désirait. Les points d'arrêt sont très utiles lorsque l'on désire tester certaines parties du programme séparément. On peut ensuite visualiser le contenu des registres internes du 68Ø9 et de la mémoire grâce à des commandes spéciales.

Nous allons vous donner maintenant quelques erreurs classiques :

— Attention aux sauts conditionnels : ne vous trompez pas sur la condition.

Exemple : Z est positionné à 1 si Accumulateur = Mémoire, par exemple, après une instruction de comparaison.

L'instruction BEQ teste si le résultat est égal à zéro donc si Z=1.

— Attention à l'ordre des opérandes : l'instruction TFR transfère le contenu du premier registre précisé dans le second et non pas l'inverse (par exemple TFR R1, R2).

— Attention aux modes d'adressage : en particulier, ne confondez pas données et adresses.

Exemple: LDA #$ØØ est différent de LDA $ØØ.

— N'oubliez pas d'initialiser les compteurs (de boucle par exemple) et les pointeurs (pointeur de pile par exemple).

— Attention aux modifications que peuvent apporter des sous-programmes dans l'état des registres internes du 68Ø9.

— Attention aux changements éventuels des bits du registre de condition CC lors de leur utilisation.

Lorsque le programme semble marcher, il est nécessaire de passer à une phase très importante de la mise au point qui est celle du test. En effet un programme qui marche dans un cas, ne marche pas forcément dans tous les cas. Inversement pour qu'un programme marche de façon sûre il faudrait le tester dans tous les cas ce qui, malheureusement, n'est généralement pas possible.

— Il faudra donc sélectionner un certain nombre de configurations qui permettront d'être quasiment sûr du bon fonctionnement du programme. On utilisera pour cela des données test convenablement classées :

— cas simples tout d'abord,
— cas particuliers,
— nombres positifs et négatifs,
— nombres nuls.

Voilà qui termine ce qui concerne l'écriture d'un programme en assembleur proprement dite.

Nous allons envisager deux cas particuliers.

Économie de place mémoire

Nous avons donné jusqu'à présent des méthodes générales pour écrire de façon sûre des programmes qui marchent. Ceci dit il peut être nécessaire dans le cas de certaines machines qui ne possèdent que peu de mémoire vive (RAM) ou dans le cas de programmes très longs, d'avoir à réduire la place occupée par un programme.

Certaines méthodes seront utiles et nous vous en donnons quelques exemples.

— Utilisez au maximum les sous-programmes pour effectuer des tâches répétitives.

— Utilisez des instructions utilisant peu d'octets et notamment l'adressage direct (surtout pour les données qui sont fréquemment utilisées).

— Utilisez la pile utilisateur pour le passage de paramètres entre différentes parties du programme.

— Utilisez des instructions qui opèrent directement sur des registres ou des cases-mémoire.

Cette liste n'est pas limitative mais est destinée à donner quelques idées générales.

Économie de temps machine

Il peut être nécessaire dans certains cas (très rares en général, du moins pour des applications individuelles de la micro-informatique) de réduire le temps d'exécution d'un programme. Certaines règles énoncées dans le paragraphe précédent iront dans ce sens, d'autres seront plus spécifiques au cas qui nous occupe ici.

— Utilisez des instructions utilisant peu d'octets.

— Essayez de sortir certaines instructions des boucles lorsqu'elles ne sont pas réellement utiles (car sinon elles seront exécutées autant de fois que la boucle elle-même).

— Utilisez peu d'instructions de saut du type JMP qui ont un temps d'exécution assez long (préférez-leurs les instructions de branchement relatif court).

— Utilisez des tables de valeurs pour les données même si elles sont nombreuses.

— N'utilisez pas ou peu d'adressages compliqués et notamment les adressages indirects, indexés, à incrémentation et décrémentation automatiques.

— Utilisez des instructions qui opèrent directement sur les registres et en mode d'adressage direct.

Nous espérons vous avoir donné quelques "trucs" qui vous permettront de mener à bien vos projets, même les plus ambitieux.

Nous donnons en annexe la liste des instructions présentes dans le 68Ø9 comme référence.

Si nous avons un conseil à vous donner, c'est celui d'acquérir un maximum de pratique en imaginant de vous-même des programmes. Et même, si vous en avez le courage, essayez de désassembler le logiciel présent dans votre micro-ordinateur (moniteur BASIC, DOS, etc...). Vous pourrez ainsi accéder à des routines très intéressantes telles que opérations en virgule flottante, bibliothèque scientifique, etc... et ainsi augmenter la puissance de vos propres programmes en assembleur.

ANNEXE

TABLEAU RÉCAPITULATIF DES INSTRUCTIONS DU 68Ø9

Le tableau ci-après regroupe les différentes instructions disponibles sur le 68Ø9. Il s'agit en fait de la feuille de caractéristiques publiée par le fabricant du 68Ø9 : Motorola.

Elle regroupe les informations suivantes :

— les mnémoniques des différentes instructions,

— les modes d'adressage disponibles,

— les code-opérations,

— les nombres de cycles d'horloge nécessaires pour l'exécution de chaque instruction,

— le nombre d'octets nécessaires pour la définition d'une instruction,

— les indicateurs du registre de condition CC affectés,

— le fonctionnement de base de l'instruction.

Instruction	Forms	Immediate Op	Immediate ~	Immediate #	Direct Op	Direct ~	Direct #	Indexed[1] Op	Indexed[1] ~	Indexed[1] #	Extended Op	Extended ~	Extended #	Inherent Op	Inherent ~	Inherent #	Description	5 H	3 N	2 Z	1 V	0 C
LSL	LSLA													48	2	1	A, B, M: C ← b7…b0 ← 0	•	↕	↕	↕	↕
	LSLB													58	2	1		•	↕	↕	↕	↕
	LSL				08	6	2	68	6+	2+	78	7	3					•	↕	↕	↕	↕
LSR	LSRA													44	2	1	A, B, M: 0 → b7…b0 → C	•	0	↕	•	↕
	LSRB													54	2	1		•	0	↕	•	↕
	LSR				04	6	2	64	6+	2+	74	7	3					•	0	↕	•	↕
MUL														3D	11	1	A × B → D (Unsigned)	•	•	↕	•	9
NEG	NEGA													40	2	1	$\overline{A}$ + 1 → A	8	↕	↕	↕	↕
	NEGB													50	2	1	$\overline{B}$ + 1 → B	8	↕	↕	↕	↕
	NEG				00	6	2	60	6+	2+	70	7	3				$\overline{M}$ + 1 → M	8	↕	↕	↕	↕
NOP														12	2	1	No Operation	•	•	•	•	•
OR	ORA	8A	2	2	9A	4	2	AA	4+	2+	BA	5	3				A V M → A	•	↕	↕	0	•
	ORB	CA	2	2	DA	4	2	EA	4+	2+	FA	5	3				B V M → B	•	↕	↕	0	•
	ORCC	1A	3	2													CC V IMM → CC				7	
PSH	PSHS	34	5+[4]	2													Push Registers on S Stack	•	•	•	•	•
	PSHU	36	5+[4]	2													Push Registers on U Stack	•	•	•	•	•
PUL	PULS	35	5+[4]	2													Pull Registers from S Stack	•	•	•	•	•
	PULU	37	5+[4]	2													Pull Registers from U Stack	•	•	•	•	•
ROL	ROLA													49	2	1	A, B, M: C ← b7…b0 ← C	•	↕	↕	↕	↕
	ROLB													59	2	1		•	↕	↕	↕	↕
	ROL				09	6	2	69	6+	2+	79	7	3					•	↕	↕	↕	↕
ROR	RORA													46	2	1	A, B, M: C → b7…b0 → C	•	↕	↕	•	↕
	RORB													56	2	1		•	↕	↕	•	↕
	ROR				06	6	2	66	6+	2+	76	7	3					•	↕	↕	•	↕
RTI														3B	6/15	1	Return From Interrupt					7
RTS														39	5	1	Return from Subroutine	•	•	•	•	•
SBC	SBCA	82	2	2	92	4	2	A2	4+	2+	B2	5	3				A − M − C → A	8	↕	↕	↕	↕
	SBCB	C2	2	2	D2	4	2	E2	4+	2+	F2	5	3				B − M − C → B	8	↕	↕	↕	↕
SEX														1D	2	1	Sign Extend B into A	•	↕	↕	0	•
ST	STA				97	4	2	A7	4+	2+	B7	5	3				A → M	•	↕	↕	0	•
	STB				D7	4	2	E7	4+	2+	F7	5	3				B → M	•	↕	↕	0	•
	STD				DD	5	2	ED	5+	2+	FD	6	3				D → M:M + 1	•	↕	↕	0	•
	STS				10 DF	6	3	10 EF	6+	3+	10 FF	7	4				S → M:M + 1	•	↕	↕	0	•
	STU				DF	5	2	EF	5+	2+	FF	6	3				U → M:M + 1	•	↕	↕	0	•
	STX				9F	5	2	AF	5+	2+	BF	6	3				X → M:M + 1	•	↕	↕	0	•
	STY				10 9F	6	3	10 AF	6+	3+	10 BF	7	4				Y → M:M + 1	•	↕	↕	0	•
SUB	SUBA	80	2	2	90	4	2	A0	4+	2+	B0	5	3				A − M → A	8	↕	↕	↕	↕
	SUBB	C0	2	2	D0	4	2	E0	4+	2+	F0	5	3				B − M → B	8	↕	↕	↕	↕
	SUBD	83	4	3	93	6	2	A3	6+	2+	B3	7	3				D − M:M + 1 → D	•	↕	↕	↕	↕
SWI	SWI[6]													3F	19	1	Software Interrupt 1	•	•	•	•	•
	SWI[6]													10 3F	20	2	Software Interrupt 2	•	•	•	•	•
	SWI[6]													11 3F	20	1	Software Interrupt 3	•	•	•	•	•
SYNC														13	≥4	1	Synchronize to Interrupt	•	•	•	•	•
TFR	R1, R2													1F	6	2	R1 → R2[2]	•	•	•	•	•
TST	TSTA													4D	2	1	Test A	•	↕	↕	0	•
	TSTB													5D	2	1	Test B	•	↕	↕	0	•
	TST				0D	6	2	6D	6+	2+	7D	7	3				Test M	•	↕	↕	0	•

Instruction	Forms	Addressing Modes															Description	5	3	2	1	0
		Immediate			Direct			Indexed			Extended			Inherent								
		Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#		H	N	Z	V	C
ABX														3A	3	1	B + X → X (Unsigned)	•	•	•	•	•
ADC	ADCA	89	2	2	99	4	2	A9	4+	2+	B9	5	3				A + M + C → A	↕	↕	↕	↕	↕
	ADCB	C9	2	2	D9	4	2	E9	4+	2+	F9	5	3				B + M + C → B	↕	↕	↕	↕	↕
ADD	ADDA	8B	2	2	9B	4	2	AB	4+	2+	BB	5	3				A + M → A	↕	↕	↕	↕	↕
	ADDB	CB	2	2	DB	4	2	EB	4+	2+	FB	5	3				B + M → B	↕	↕	↕	↕	↕
	ADDD	C3	4	3	D3	6	2	E3	6+	2+	F3	7	3				D + M:M + 1 → D	•	↕	↕	↕	↕
AND	ANDA	84	2	2	94	4	2	A4	4+	2+	B4	5	3				A ∧ M → A	•	↕	↕	0	•
	ANDB	C4	2	2	D4	4	2	E4	4+	2+	F4	5	3				B ∧ M → B	•	↕	↕	0	•
	ANDCC	1C	3	2													CC ∧ IMM → CC					7
ASL	ASLA													48	2	1	A	8	↕	↕	↕	↕
	ASLB													58	2	1	B	8	↕	↕	↕	↕
	ASL				08	6	2	68	6+	2+	78	7	3				M (c ← b7 … b0 ← 0)	8	↕	↕	↕	↕
ASR	ASRA													47	2	1	A	8	↕	↕	•	↕
	ASRB													57	2	1	B	8	↕	↕	•	↕
	ASR				07	6	2	67	6+	2+	77	7	3				M (b7 … b0 → c)	8	↕	↕	•	↕
BIT	BITA	85	2	2	95	4	2	A5	4+	2+	B5	5	3				Bit Test A (M ∧ A)	•	↕	↕	0	•
	BITB	C5	2	2	D5	4	2	E5	4+	2+	F5	5	3				Bit Test B (M ∧ B)	•	↕	↕	0	•
CLR	CLRA													4F	2	1	0 → A	•	0	1	0	0
	CLRB													5F	2	1	0 → B	•	0	1	0	0
	CLR				0F	6	2	6F	6+	2+	7F	7	3				0 → M	•	0	1	0	0
CMP	CMPA	81	2	2	91	4	2	A1	4+	2+	B1	5	3				Compare M from A	8	↕	↕	↕	↕
	CMPB	C1	2	2	D1	4	2	E1	4+	2+	F1	5	3				Compare M from B	8	↕	↕	↕	↕
	CMPD	10	5	4	10	7	3	10	7+	3+	10	8	4				Compare M:M + 1 from D	•	↕	↕	↕	↕
		83			93			A3			B3											
	CMPS	11	5	4	11	7	3	11	7+	3+	11	8	4				Compare M:M + 1 from S	•	↕	↕	↕	↕
		8C			9C			AC			BC											
	CMPU	11	5	4	11	7	3	11	7+	3+	11	8	4				Compare M:M + 1 from U	•	↕	↕	↕	↕
		83			93			A3			B3											
	CMPX	8C	4	3	9C	6	2	AC	6+	2+	BC	7	3				Compare M:M + 1 from X	•	↕	↕	↕	↕
	CMPY	10	5	4	10	7	3	10	7+	3+	10	8	4				Compare M:M + 1 from Y	•	↕	↕	↕	↕
		8C			9C			AC			BC											
COM	COMA													43	2	1	$\overline{A}$ → A	•	↕	↕	0	1
	COMB													53	2	1	$\overline{B}$ → B	•	↕	↕	0	1
	COM				03	6	2	63	6+	2+	73	7	3				$\overline{M}$ → M	•	↕	↕	0	1
CWAI		3C	≥20	2													CC ∧ IMM → CC Wait for Interrupt					7
DAA														19	2	1	Decimal Adjust A	•	↕	↕	0	↕
DEC	DECA													4A	2	1	A − 1 → A	•	↕	↕	↕	•
	DECB													5A	2	1	B − 1 → B	•	↕	↕	↕	•
	DEC				0A	6	2	6A	6+	2+	7A	7	3				M − 1 → M	•	↕	↕	↕	•
EOR	EORA	88	2	2	98	4	2	A8	4+	2+	B8	5	3				A ⊻ M → A	•	↕	↕	0	•
	EORB	C8	2	2	D8	4	2	E8	4+	2+	F8	5	3				B ⊻ M → B	•	↕	↕	0	•
EXG	R1, R2													1E	8	2	R1 ↔ R2[2]	•	•	•	•	•
INC	INCA													4C	2	1	A + 1 → A	•	↕	↕	↕	•
	INCB													5C	2	1	B + 1 → B	•	↕	↕	↕	•
	INC				0C	6	2	6C	6+	2+	7C	7	3				M + 1 → M	•	↕	↕	↕	•
JMP					0E	3	2	6E	3+	2+	7E	4	3				EA[3] → PC	•	•	•	•	•
JSR					9D	7	2	AD	7+	2+	BD	8	3				Jump to Subroutine	•	•	•	•	•
LD	LDA	86	2	2	96	4	2	A6	4+	2+	B6	5	3				M → A	•	↕	↕	0	•
	LDB	C6	2	2	D6	4	2	E6	4+	2+	F6	5	3				M → B	•	↕	↕	0	•
	LDD	CC	3	3	DC	5	2	EC	5+	2+	FC	6	3				M:M + 1 → D	•	↕	↕	0	•
	LDS	10	4	4	10	6	3	10	6+	3+	10	7	4				M:M + 1 → S	•	↕	↕	0	•
		CE			DE			EE			FE											
	LDU	CE	3	3	DE	5	2	EE	5+	2+	FE	6	3				M:M + 1 → U	•	↕	↕	0	•
	LDX	8E	3	3	9E	5	2	AE	5+	2+	BE	6	3				M:M + 1 → X	•	↕	↕	0	•
	LDY	10	4	4	10	6	3	10	6+	3+	10	7	4				M:M + 1 → Y	•	↕	↕	0	•
		8E			9E			AE			BE											
LEA	LEAS							32	4+	2+							EA[3] → S	•	•	•	•	•
	LEAU							33	4+	2+							EA[3] → U	•	•	•	•	•
	LEAX							30	4+	2+							EA[3] → X	•	•	↕	•	•
	LEAY							31	4+	2+							EA[3] → Y	•	•	↕	•	•

Instruction	Forms	Addressing Mode Relative OP	—	#	Description	5 H	3 N	2 Z	1 V	0 C
BCC	BCC	24	3	2	Branch C = 0	•	•	•	•	•
	LBCC	10 24	5(6)	4	Long Branch C = 0	•	•	•	•	•
BCS	BCS	25	3	2	Branch C = 1	•	•	•	•	•
	LBCS	10 25	5(6)	4	Long Branch C = 1	•	•	•	•	•
BEQ	BEQ	27	3	2	Branch Z = 0	•	•	•	•	•
	LBEQ	10 27	5(6)	4	Long Branch Z = 0	•	•	•	•	•
BGE	BGE	2C	3	2	Branch ≥ Zero	•	•	•	•	•
	LBGE	10 2C	5(6)	4	Long Branch ≥ Zero	•	•	•	•	•
BGT	BGT	2E	3	2	Branch > Zero	•	•	•	•	•
	LBGT	10 2E	5(6)	4	Long Branch > Zero	•	•	•	•	•
BHI	BHI	22	3	2	Branch Higher	•	•	•	•	•
	LBHI	10 22	5(6)	4	Long Branch Higher	•	•	•	•	•
BHS	BHS	24	3	2	Branch Higher or Same	•	•	•	•	•
	LBHS	10 24	5(6)	4	Long Branch Higher or Same	•	•	•	•	•
BLE	BLE	2F	3	2	Branch ≤ Zero	•	•	•	•	•
	LBLE	10 2F	5(6)	4	Long Branch ≤ Zero	•	•	•	•	•
BLO	BLO	25	3	2	Branch lower	•	•	•	•	•
	LBLO	10 25	5(6)	4	Long Branch Lower	•	•	•	•	•

Instruction	Forms	Addressing Mode Relative OP	—	#	Description	5 H	3 N	2 Z	1 V	0 C
BLS	BLS	23	3	2	Branch Lower or Same	•	•	•	•	•
	LBLS	10 23	5(6)	4	Long Branch Lower or Same	•	•	•	•	•
BLT	BLT	2D	3	2	Branch < Zero	•	•	•	•	•
	LBLT	10 2D	5(6)	4	Long Branch < Zero	•	•	•	•	•
BMI	BMI	2B	3	2	Branch Minus	•	•	•	•	•
	LBMI	10 2B	5(6)	4	Long Branch Minus	•	•	•	•	•
BNE	BNE	26	3	2	Branch Z ≠ 0	•	•	•	•	•
	LBNE	10 26	5(6)	4	Long Branch Z ≠ 0	•	•	•	•	•
BPL	BPL	2A	3	2	Branch Plus	•	•	•	•	•
	LBPL	10 2A	5(6)	4	Long Branch Plus	•	•	•	•	•
BRA	BRA	20	3	2	Branch Always	•	•	•	•	•
	LBRA	16	5	3	Long Branch Always	•	•	•	•	•
BRN	BRN	21	3	2	Branch Never	•	•	•	•	•
	LBRN	10 21	5	4	Long Branch Never	•	•	•	•	•
BSR	BSR	8D	7	2	Branch to Subroutine	•	•	•	•	•
	LBSR	17	9	3	Long Branch to Subroutine	•	•	•	•	•
BVC	BVC	28	3	2	Branch V = 0	•	•	•	•	•
	LBVC	10 28	5(6)	4	Long Branch V = 0	•	•	•	•	•
BVS	BVS	29	3	2	Branch V = 1	•	•	•	•	•
	LBVS	10 29	5(6)	4	Long Branch V = 1	•	•	•	•	•

Adressing mode: mode d'adressage

Immediate: immédiat

Direct: direct

Extended: étendu

Inhérent: implicite

Relative: relatif

OP: code-opération

—: nombre de cycles horloge nécessaires

#: nombre d'octets nécessaires

●: non affecté

I: bit positionné à 1 ou Ø

EA: adresse effective

8: valeur du bit H indéfinie

9: C=1 si b7=1

Imprimerie de la Manutention à Mayenne
Dépôt légal : novembre 1984
N° d'Éditeur : 4180

Ce livre est destiné à tous ceux, à toutes celles qui ont décidé d'aller un peu plus loin en informatique individuelle, grâce aux étonnantes possibilités de la programmation en ASSEMBLEUR.

Plutôt que de vous plonger à corps perdu dans la mer des instructions au risque de vous y noyer, vous aborderez tranquillement le problème en utilisant au maximum ce que vous connaissez déjà : le BASIC.

Ensuite, nous décrivons de manière progressive et à l'aide de nombreux exemples l'assembleur du 68Ø9, le microprocesseur 8 bits le plus puissant existant actuellement sur le marché, et utilisé déjà sur de nombreuses machines (THOMSON TO 7, TO7-7Ø, MO5, DRAGON, TRS8Ø couleur, VEGAS 2 ØØØ, etc.).

Afin de prendre un bon départ, des exemples de programmes classiques sont largement développés et commentés, et vous trouverez des conseils sur la façon de bien programmer et de faire tourner vos propres programmes.

EYROLLES

8709

Téléchargé sur
Le Vieux Manuel

Http://www.abandonware-manuels.org