MICRO-ORDINATEURS

L'ASSEMBLEUR FACILE DU 6502

ET DU 6510

François MONTEIL

EYROLLES

L'ASSEMBLEUR FACILE DU 6502 ET DU 6510

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 07.
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.

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.

L'ASSEMBLEUR FACILE DU 6502

par

François MONTEIL

Collection animée
par Richard SCHOMBERG

TROISIÈME ÉDITION
Nouveau tirage

EYROLLES

61, Boulevard Saint-Germain — 75005 Paris
1984

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 65Ø2. Par contre, certains doivent déjà avoir l'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 vous 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. Cet ouvrage commence donc avec une introduction à l'ASSEMBLEUR dans laquelle celui-ci est comparé au BASIC. Ensuite, comme rien ne vaut l'expérience personnelle, nous décrivons l'assembleur de l'un des microprocesseurs les plus utilisés actuellement : le 65Ø2. 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 classiques sont largement développés et commentés.

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

Nous n'avons pas abordé ici 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.

De plus, comme le laisse entendre son titre, cet ouvrage s'adresse également aux possesseurs de COMMODORE 64, qui est équipé d'un processeur de type 6510, semblable au 6502 au niveau du jeu d'instructions.

Table des matières

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 :

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

Dans un programme BASIC, l'interpréteur lit une ligne, la traduit en instructions machine et l'exécute. De plus les différentes lignes sont traitées les unes après les autres.

Ici le déroulement est le suivant :

– le microprocesseur attend tout d'abord qu'une valeur soit rentrée au clavier.

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

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

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

– si A<O alors B=A

– si A>O 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
        CMP #$ØØ             ; SI NEGATIF B=A
        BMI SORTIE           ; SI POSITIF B=–A
        LDA #$ØØ             ; CALCULE B=–A
        SEC
        SBC VAR
SORTIE  STA AFFICH           ; RANGE B EN MEMOIRE
        BRK
```

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 : " SORTIE " ; cette étiquette correspond exactement au " 5Ø " du programme BASIC.

2) la zone instruction : exemple " LDA "

Dans cette zone se trouvent des mnémoniques ou ensembles de 3 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 mémoire particulière située à l'intérieur du 65Ø2, la valeur A située initialement à l'adresse VAR.

L'instruction " CMP #$ØØ " la compare avec Ø et si le résultat est négatif il y a branchement, grâce à l'instruction " BMI SORTIE " à l'étiquette SORTIE. La valeur B = A est donc toujours stockée

dans l'Accumulateur et est ensuite rangée à l'adresse AFFICH. Si par contre le résultat avait été positif ou nul le programme aurait continué normalement.

Les trois instructions suivantes

```
LDA #$ØØ
SEC
SBC VAR
```

calculent la valeur B=-A (nous ne rentrerons pas ici dans les détails) 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 microordinateurs 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 a l'air à première vue.

Nous allons maintenant vous proposer une deuxième approche en remontant aux sources et en partant, cette fois-ci, du microprocesseur. Nous en profiterons pour expliquer au passage quelques termes courants qui font partie du vocabulaire de l'assembleur.

Au début était le microprocesseur : un " cafard " à 4Ø pattes qui finalement ne paye pas de mine : on ne soupçonnerait pas, en le voyant, qu'il est capable d'effectuer par exemple 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 " ou des " 1 " (le courant passe ou ne passe pas) : c'est ce qu'on appelle le binaire. Un chiffre binaire (donc Ø ou 1) est appelé bit. La mémoire, dans le cas du 65Ø2 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 Ø et 65535, ce dernier nombre représentant l'espace-mémoire maximal adressable par le microprocesseur.

Un octet est donc de la forme suivante :

a_7	a_6	a_5	a_4	a_3	a_2	a_1	$a_Ø$

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

Le jeu d'instructions du 65Ø2 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 2^8-1 mots différents) chacun réalisant une fonction spécifique : on les nomme code-opérations.

Par exemple le mot Ø11Ø11Ø1 veut dire " additionne le contenu de la case mémoire, dont l'adresse est donnée par les deux octets suivant immédiatement le code-opération, à l'accumulateur et range le résultat dans ce dernier ". Ces deux octets représentent ce qu'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 : 6D correspond à ADC (addition en adressage étendu) avec
6D = Ø11Ø11Ø1

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

A chaque instruction correspond un mnémonique de trois lettres.

Exemple : l'addition s'écrit ADC
la soustraction s'écrit SBC
un appel de sous programme s'écrit JSR

– A chaque mnémonique sera associée une opérande qui doit indiquer en clair à quelle adresse le microprocesseur doit aller chercher la donnée traitée par l'instruction en question.

Exemple :

ADC 64ØØØ veut dire additionner le contenu de la case-mémoire d'adresse 64ØØØ à l'accumulateur.

– 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'instruction 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 65Ø2.

Son rôle est de traduire le programme appelé source écrit en mnémoniques 65Ø2 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 65Ø2 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 65Ø2 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 :

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

ou bien encore

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

Si nous nous plaçons maintenant dans un système de base 2 on aura :

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

ou encore

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

Le nombre 3783 en base 2 s'écrira donc

3783 = 111Ø11ØØØ111

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

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 code-opérations du 65Ø2 étaient données sous forme d'un octet (donc d'un mot de 8 bits). Il peut donc exister au maximum $2^8 = 256$ code-opérations différents ; en fait tous les codes ne sont pas attribués et le jeu d'instruction du 65Ø2 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'HEXADECIMAL

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 10 chiffres (0 à 9), l'hexadécimal comporte 16 caractères qui sont les suivants :

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

On a donc les équivalences suivantes :

hexadécimal	décimal	binaire
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
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.

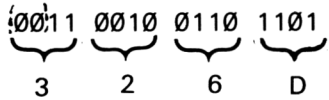
Considérons tout d'abord l'octet suivant :

$$\underbrace{0110}_{6}\ \underbrace{1001}_{9}$$

donc 01101001 = 69 en hexadécimal = 105 en décimal.

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

Soit par exemple le mot suivant :



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 2 caractères (les code-opérations sont donc compris entre ØØ et FF). Une adresse de 16 bits nécessitera de même 4 caractères hexadécimaux.

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 = ØØ11 Ø11Ø

3 6

Cela nous amène à une constatation : ce code perd une grande quantité de place mémoire. En effet un octet ne permet 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 65Ø2 qui effectue 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-dessous 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	^		
2F	/	5F	←		

2.5. REPRESENTATION DES NOMBRES NEGATIFS

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

→ 11111ØØ1 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 complèment 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 65Ø2, ou plutôt son assembleur.

3
La syntaxe assembleur 6502

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 65Ø2 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  ADC ($2Ø),Y      ; 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     ADC ($2Ø),Y      ; EFFECTUE LA SOMME
Délimiteur  champ
            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 également 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 65Ø2.

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 :

%Ø1ØØ11Ø1
ou bien 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 : ADC VALEUR

Supposons que VALEUR = $1Ø, alors il y aura addition entre l'accumulateur et la case-mémoire d'adresse $ØØ1Ø (en page-zéro). La page-zéro sera introduite au prochain chapitre.

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 : ADC 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.

– Ecrivez 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 65Ø2 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
```

ou bien `*=$Ø2ØØ` selon l'assembleur utilisé

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
      ADC    #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 $Ø5          ou bien COMPT  =$Ø5
FIN     EQU DEBUT+$6Ø            FIN    =DEBUT+$6Ø
```

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

En général, on placera les directives " EQU " ou " = " 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 •BYTE.* – Elle permet de réserver un octet en mémoire.

Exemple : VAL .BYTE $3Ø

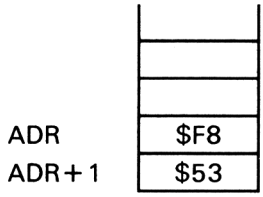
Lorsque l'assembleur rencontre cette ligne, il place la valeur $3Ø dans la première case-mémoire et lui assigne la variable VAL.

Le pointeur d'adresse est alors incrementé de 1.

– *la directive •WORD.* – Elle permet de réserver un double octet en mémoire. Cette directive permet de stocker une adresse en mémoire. En effet l'octet de poids faible est stocké le premier. Ici le pointeur d'adresse est incrementé de 2.

Exemple : ADR .WORD $53F8

L'état de la mémoire sera le suivant :

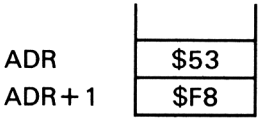


La variable ADR est assignée à $53F8

– *la directive •DBYTE (double byte).* – Cette pseudo-instruction permet également de réserver un double octet en mémoire mais, contrairement à la précédente, elle stocke l'octet de poids fort en premier.

Exemple : ADR .DBYTE $53F8

L'état de la mémoire sera le suivant :



La variable ADR est assignée à $53F8

– *la directive •TEXT.* – Cette directive permet de réserver un bloc mémoire pour stocker une chaîne de caractères ASCII.

Exemple : DONNEE .TEXT /NUMERO?/

Les caractères " N ", " U ", " M ", " E ", " R ", " O ", " ? " sont stockés en mémoire.

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

– *la directive réserve.* – Elle permet de réserver un espace-mémoire afin d'y stocker par exemple un tableau de données.

Exemple : •TAB RESERVE 30
ou bien *=*+30

Ici la variable TAB est assignée à un tableau de 30 octets

e) **Les directives agissant sur le listing du programme**

Il s'agit par exemple de la directive •TITLE qui permet d'imprimer un titre en haut de chaque page de listing ou de la directive •PAGE qui provoque un saut de page lors du listing du programme assemblé.

Le listing est le suivant

```
PRODB       EQU    $00        ;PARTIE BASSE DU RESULTAT
PRODH       EQU    $01        ;PARTIE HAUTE DU RESULTAT
VAR         EQU    $02        ;INTERMEDIAIRE DE CALCUL
MCANDE      EQU    $05        ;MULTIPLICANDE
MTEUR       .BYTE  $12        ;MULTIPLICATEUR
            .ORG   $0200
DEBUT       LDX    #$08       ;INITIALISE COMPTEUR DE BOUCLE
            LDA    #$00       ;ANNULE LE RESULTAT
            STA    PRODB      ;PARTIE BASSE
            STA    PRODH      ;PARTIE HAUTE
            STA    VAR        ;INTERMEDIAIRE DE CALCUL
DECAL       LSR    MTEUR      ;BIT 0=0?
            BCC    SUITE      ;OUI,DECALAGE MULTIPLICANDE
            LDA    PRODB      ;NON,ADDITIONNE MULTIPLICANDE
            CLC
            ADC    MCANDE     ;PARTIE BASSE
            STA    PRODB      ;SAUVEGARDE PARTIE BASSE DU RESULTAT
            LDA    PRODH      ;PARTIE HAUTE
            ADC    VAR        ;
            STA    PRODH      ;SAUVEGARDE PARTIE HAUTE
SUITE       ASL    MCANDE     ;DECALAGE MULTIPLICANDE A GAUCHE
            ROL    VAR        ;SAUVEGARDE BIT 7 DANS VAR
            DEX               ;DERNIERE ITERATION?
            BNE    DECAL      ;NON,RECOMMENCE
            BRK               ;OUI,TERMINE
            .END
```

Nous allons nous arrêter là dans notre énumération. Nous vous donnons ci-dessus 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 65Ø2. 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 possèdez.

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 code-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.

Nous vous donnons ci-dessous le listing du programme précédent mais assemblé cette fois-ci.

```
                0010          •ORG   $0200
                0020 ;
                0030 ;
                0040 ;
                0050 PRODB    EQU    $00      ; PARTIE BASSE DU RESULTAT
                0060 PRODH    EQU    $01      ; PARTIE HAUTE DU RESULTAT
                0070 VAR      EQU    $02      ; INTERMEDIAIRE DE CALCUL
                0080 MCANDE   EQU    $05      ; MULTIPLICANDE
0200 12         0090 MTEUR    •BYTE  $12      ; MULTIPLICATEUR
                0100
                0110
                0120
0201 A2 08      0130 DEBUT    LDX    #$08     ; INITIALISE COMPTEUR DE BOU-
                                                CLE
0203 A9 00      0140          LDA    #$00     ; ANNULE LE RESULTAT
0205 85 00      0150          STA    PRODB    ; PARTIE BASSE
0207 85 01      0160          STA    PRODH    ; PARTIE HAUTE
0209 85 02      0170          STA    VAR      ; INTERMEDIAIRE DE CALCUL
020B 4E 00 02   0180 DECAL    LSR    MTEUR    ; BIT 0=0?
020E 90 0D      0190          BCC    SUITE    ; OUI, DECALAGE MULTIPLICANDE
0210 A5 00      0200          LDA    PRODB    ; NON, ADDITIONNE MULTIPLI-
                                                CANDE
0212 18         0210          CLC
0213 65 05      0220          ADC    MCANDE   ; PARTIE BASSE
0215 85 00      0230          STA    PRODB    ; SAUVEGARDE PARTIE BASSE DU
                                                RESULTAT
0217 A5 01      0240          LDA    PRODH    ; PARTIE HAUTE
0219 65 02      0250          ADC    VAR
021B 85 01      0260          STA    PRODH    ; SAUVEGARDE PARTIE HAUTE
021D 06 05      0270 SUITE    ASL    MCANDE   ; DECALAGE MULTIPLICANDE A
                                                GAUCHE
021F 26 02      0280          ROL    VAR      ; SAUVEGARDE BIT 7 DANS VAR
0221 CA         0290          DEX             ; DERNIERE ITERATION?
0222 D0 E7      0300          BNE    DECAL    ; NON,RECOMMENCE
0224 00         0310          BRK             ; OUI,TERMINE
                0320          •END

SYMBOL TABLE:
PRODB=0000    PRODH=0001    VAR=0002
MCANDE =0005  MTEUR=0200    DEBUT=0201
DECAL=020B    SUITE=021D
```

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, 2 ou 3 octets), 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 incrementé de 1, 2 ou 3 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 65Ø2 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.

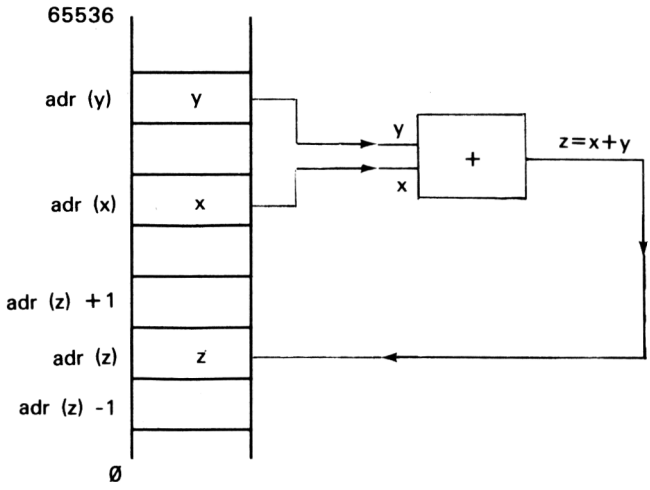
4
Description du 6502

4.1. LES REGISTRES INTERNES DU 6502

4.1.1. L'accumulateur

Avant de présenter les différents registres du 65Ø2 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 :

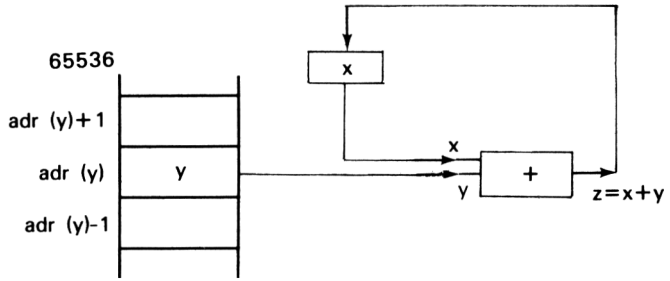


Nous avons déjà dit qu'un microprocesseur classique (et en particulier le 65Ø2) pouvait adresser 64 K octets de mémoire (ou plus exactement 65536 = 2^{16} octets). Afin de pouvoir sélectionner une case-mémoire et une seule, on définit une adresse sur 16 bits (ce qui fait donc 2 mots de 8 bits). Nous avons vu d'autre part qu'une instruction (ici addition) pouvait être codée par un mot de 8 bits 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 65Ø2 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 x+y=z) que nous nommerons registre. Plus précisément le rôle joué par celui-ci dans le cas qui nous intéresse est généralement confié à l'accumulateur (noté A) qui est un registre fondamental de la plupart des microprocesseurs existant actuellement sur le marché et notamment du 65Ø2.

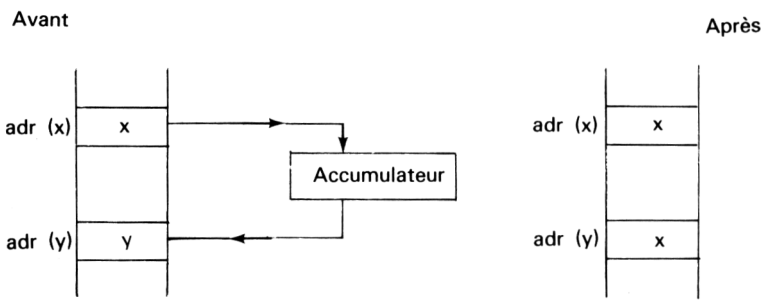
On l'utilise pour les instructions suivantes :

– pratiquement toutes les instructions arithmétiques et logiques

– certaines instructions de comparaison.

Mais il est également utilisé comme intermédiaire lors d'un transfert d'une case-mémoire à une autre case-mémoire.

Le cheminement de la donnée est alors le suivant :



Le nombre binaire x contenu dans adr (x) est d'abord transféré dans l'accumulateur A puis le contenu de ce dernier (donc x) est transféré à son tour dans adr (y).

4.1.2. Les registres d'index X et Y

A côté de l'accumulateur il existe deux registres (X et Y) appelés registres d'index. Ceux-ci sont accessibles à l'utilisateur au même titre que l'accumulateur et ils peuvent être mis en jeu directement dans la plupart des instructions arithmétiques et logiques. Ces deux registres, comme l'accumulateur A, possèdent chacun 8 bits. Ils ont des domaines d'application assez particuliers.

– comme registres d'index (nous reviendrons sur ce point un peu plus loin lorsque nous parlerons des modes d'adressage du 65Ø2).

– comme registres de stockage de résultats intermédiaires.

– comme compteurs de boucle.

Prenons par exemple le programme BASIC suivant :

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

Chaque fois que l'interpréteur BASIC passe en 1Ø la valeur de I est incrémentée et le programme affiche :

1
2
3
.
.
.
.
.
.
1Ø

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 (1Ø par exemple).

Nous reviendrons sur les méthodes employées pour écrire des boucles en assembleur dans le chapitre consacré au jeu d'instruction du 65Ø2.

4.1.3. Le registre d'état P

Ce registre est un peu particulier : il comporte 8 bits dont 7 seulement sont utilisés. Ces 7 bits sont les suivants :

N	V		B	D	I	Z	C

N = indicateur de résultat négatif
V = indicateur de débordement
B = indicateur de Break (BRK)
D = indicateur de mode décimal
I = masque d'interruptions
Z = indicateur de zéro
C = indicateur de retenue (CARRY)

Ces indicateurs (du moins pour la plupart d'entre eux) peuvent jouer deux rôles :

– Ils peuvent influencer le déroulement futur du programme si on les prépositionne à une certaine valeur (Ø ou 1)

– Leur état peut dépendre du résultat d'un calcul antérieur et peut donc constituer une information précieuse pour la suite.

Le microprocesseur possède des instructions particulières qui permettent de positionner un indicateur de ce registre P à une valeur déterminée. D'autre part, certaines instructions du 65Ø2 ont un déroulement qui dépend de la valeur d'un de ces indicateurs, par exemple les 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 65Ø2.

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).

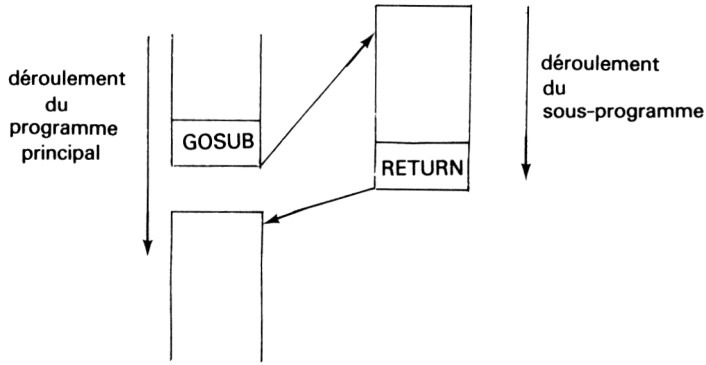
Afin que le programme se déroule correctement, il apparaît comme évident que le microprocesseur, lorsqu'il est en train d'exécuter une instruction, doit connaître l'adresse de la suivante.

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 : en effet nous avons vu ci-dessus que l'adresse d'un mot mémoire pouvait être 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 incrementé de 1.

4.1.5. Le pointeur de pile : registre SP

a) **Notion de pile**

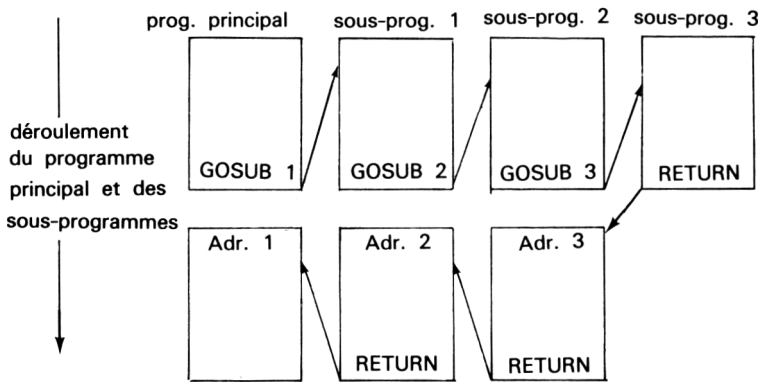
Revenons sur le rôle du compteur ordinal (PC) : nous avons vu qu'il était nécessaire que le microprocesseur sache à tout instant l'adresse de la prochaine instruction à exécuter. Examinons ce qui se passe lorsque le programme principal fait appel à un sous-programme.



Le déroulement dù 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.

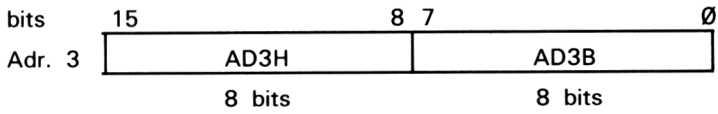
Il existe une zone mémoire située dans l'espace adressable du microprocesseur et qui fonctionne suivant le principe de la pile d'assiettes décrite ci-dessus. Cette zone-mémoire est située dans le cas du 65Ø2 entre les adresses $Ø1ØØ et $Ø1FF (elle comporte 256 octets et pourra stocker par exemple 128 adresses de retour de sous-programme ce qui finalement n'est pas si mal !).

Si nous considérons l'exemple choisi précédemment la pile contiendra après l'appel du sous-programme 3, les octets suivants :

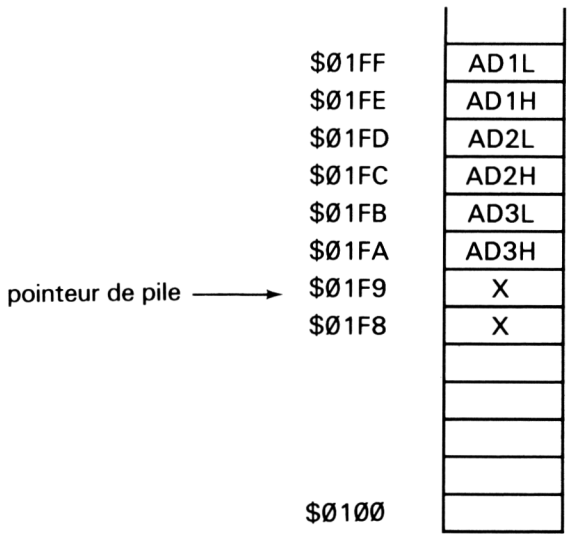
On notera

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

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



Etat de la pile après l'appel du sous-programme 3.



On remarque que les adresses sont chargées dans la pile à partir du haut (donc de $Ø1FF) et vers le bas, l'octet de poids faible de l'adresse étant stocké en premier.

Il faut noter que la pile peut servir à sauvegarder la valeur de certains registres (par exemple l'accumulateur A ou le registre d'état P) lors de l'appel de sous-programmes par exemple. Ces données pourront être restituées ensuite lors du retour au programme principal.

b) **Le pointeur de pile : registre SP** (de l'Anglais Stack Pointer = pointeur de pile).

Mais, nous direz-vous, comment le microprocesseur fait-il pour savoir quelle est la dernière adresse de retour stockée ? Cela est bien simple : il existe un registre spécial (registre SP) qui accomplit cette tâche. A chaque instant celui-ci pointe vers le dernier octet vide de la pile (voir figure ci-dessus). Dans notre cas le contenu du registre SP est $F9.

Chaque fois que le processeur doit mémoriser une adresse il la transfère dans la pile (dans les deux premiers octets libres) et le pointeur de pile se déplace vers le bas (c'est-à-dire est décrémenté de 2).

Lors d'une instruction de retour de sous-programme, 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 (donc est incrémenté de 2).

Nous en avons maintenant terminé avec la description des registres internes du 65Ø2. Que le lecteur se rassure si quelques points lui semblent encore un peu obscurs : nous retrouverons tous ces registres bientôt avec le jeu d'instruction du 65Ø2.

Nous allons maintenant parler des différents modes d'adressage de ce microprocesseur.

4.2. LES DIFFERENTS MODES D'ADRESSAGE DU 65Ø2

Tout d'abord qu'appelle-t-on exactement mode d'adressage ? Nous avons vu au début de ce chapitre (paragraphe I) un exemple (addition de deux nombres) où 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 appelerons ce mode " adressage direct ou étendu " (car on peut accéder directement, grâce à lui, à la totalité de l'espace mémoire adressable du 65Ø2).

Donc 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 sur un octet et un seul. On peut mentionner parmi elles les opérations sur la pile, les opérations de transfert de registre interne à registre interne, les opérations sur le registre d'état P etc. Nous n'en dirons pas plus concernant cet adressage car nous étudierons en détail les instructions dans le prochain chapitre.

4.2.2. L'adressage accumulateur

Dans ce mode d'adressage, l'instruction agit directement sur le contenu de l'accumulateur A. Le nombre d'instructions concernées est très réduit dans le cas du 65Ø2 (au nombre de 4). Il s'agit des instructions de décalage arithmétique et de rotation. Notons que ces instructions sont également codées sur un octet.

4.2.3. L'adressage immédiat

Dans ce mode d'adressage les instructions sont codées sur deux

octets. Le premier contient le code-opération de l'instruction et le deuxième une constante de 8 bits (deux chiffres hexadécimaux).

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

L'instruction sera la suivante :

```
ADC #$Ø5
```

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

4.2.4. L'adressage direct ou étendu

Ce mode d'adressage a déjà été vu dans ce chapitre. L'opérande est ici une adresse de 16 bits (quatre chiffres hexadécimaux).

Exemple : Soit à additionner le contenu de la case-mémoire $53F8 à l'accumulateur.

L'instruction s'écrit : `ADC $53F8`

Grâce à cet adressage on peut (nous l'avons déjà dit) accéder à la totalité de l'espace-mémoire du 65Ø2.

4.2.5. L'adressage en page-zéro

Ce mode d'adressage est une sorte de restriction de l'adressage direct. Supposons que nous voulions coder l'instruction de l'exemple précédent sur deux octets au lieu de trois ; l'adresse ne sera alors codée que sur un octet mais cela sera suffisant si nous nous limitons aux 256 (=$FF) premiers octets de l'espace-mémoire, ce qui correspond à la page-zéro (adresses $ØØØØ à $ØØFF). Ce mode d'adressage présente de grands avantages car il permet :

– de diminuer la place mémoire occupée par le programme (réduction de 3 à 2 du nombre d'octets nécessaires pour coder l'instruction).

– de diminuer de cette façon le temps d'exécution de ce programme. Cette page-zéro est souvent utilisée par les programmeurs

comme une extension du nombre de registres possédés par le 65Ø2. Donc cette zone mémoire devra être réservée aux données souvent utilisées et auxquelles il est nécessaire d'accéder très rapidement.

Exemple : On veut additionner le contenu de l'adresse $ØØ58 à l'accumulateur.

L'instruction s'écrit : ADC $58

Certains assembleurs reconnaissent automatiquement qu'il s'agit d'un adressage en page-zéro par la présence d'un octet et un seul après le mnémonique ADC. D'autres demandent un signe particulier (un astérisque par exemple).

4.2.6. L'adressage direct indexé (ou étendu indexé)

Nous avons vu que le 65Ø2 possédait deux registres d'index X et Y. Comme leur nom l'indique, ils sont utilisés dans ce nouveau mode d'adressage.
Dans l'adressage étendu l'adresse de la case-mémoire cherchée était donnée par deux octets et l'addition du contenu de la mémoire $53F8 avec l'accumulateur s'écrivait.

```
ADC $53F8
```

Ici l'adresse de la case mémoire cherchée résulte de l'addition de l'adresse spécifiée par l'opérande (ici $53F8) et du contenu du registre X (ou Y) qui est un nombre signé (donc compris entre -128 et +127). La syntaxe assembleur est la suivante :

```
ADC $53F8,X adressage direct indexé par X
ADC $53F8,Y adressage direct indexé par Y
```

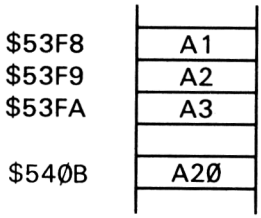
Exemple : Cas d'un adressage direct indexé par X.

Supposons que X contienne la valeur $2Ø
l'adresse résultante sera alors $53F8+$2Ø=$5418

Maintenant que le principe d'utilisation de ce mode d'adressage a été donné, voyons à quoi il peut servir.

Supposons qu'il existe dans la mémoire du 65Ø2 un tableau de valeurs situées à des adresses consécutives et que l'on veuille accéder successivement à chacun des éléments de cette table. Comme vous pouvez vous en douter, nous allons utiliser l'adressage direct indexé.

Prenons par exemple une table de 2Ø éléments situés entre les adresses $53F8 et $54ØB



Chargeons au départ le registre X par $ØØ. Grâce à l'instruction

```
ADC $53F8,X
```

nous pouvons accéder à A1. Si nous incrémentons X de 1 nous accédons alors à A2, et ainsi de suite.

A2Ø sera connu lorsque le registre X contiendra la valeur $13. Du fait de la taille des registres X et Y, les tableaux de valeurs pourront contenir au maximum 256 valeurs ce qui est souvent largement suffisant.

4.2.7 L'adressage direct indexé en page-zéro

De la même façon que l'adressage direct en page-zéro est une restriction de l'adressage direct, l'adressage direct indexé en page-zéro est une restriction de l'adressage direct indexé.

Nous ne nous étendrons pas sur ce mode d'adressage qui est en tous points similaire au précédent sauf en ce qui concerne l'adresse

de base spécifiée dans l'instruction. Elle n'est codée que sur un octet et ne concerne donc que la page-zéro. La syntaxe assembleur 65Ø2 est la suivante :

```
ADC $58,X
ADC $58,Y
```

Notons que l'adressage direct page-zéro indexé par Y n'existe que pour un nombre très réduit d'instructions.

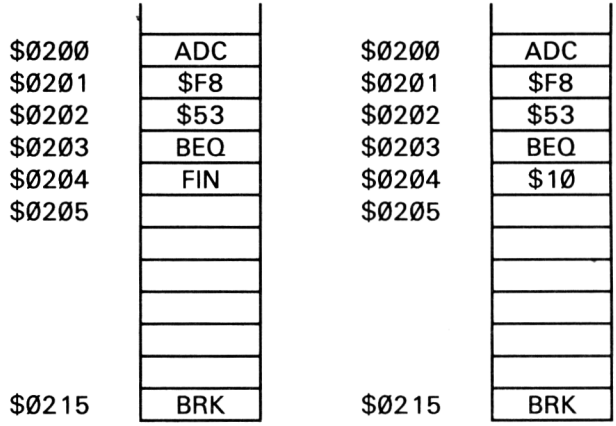
4.2.8. L'adressage relatif

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 :

```
      ADC $53F8
      BEQ FIN
        -
        -
        -
FIN   BRK
```

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 d'état est alors égal à 1. Sinon, le programme continue et exécute les instructions présentes après le BEQ. Supposons que le début du programme (instruction ADC) soit placé à l'adresse $Ø2ØØ. Nous donnons ci-dessous le contenu de la mémoire à partir de cette adresse :



Nous n'avons pas introduit les code-opérations des instructions ADC, BEQ et BRK 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 hexadécimal) 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
```

La syntaxe assembleur est la suivante :

a) si on utilise une étiquette

```
BEQ FIN
```

L'utilisation d'une étiquette est bien utile car l'assembleur se charge lui-même du calcul du déplacement relatif ce qui, croyez-nous-en sur parole, est bien sympathique de sa part.

b) si le déplacement est connu on peut écrire

```
BEQ *+10
```

Notons que la valeur du déplacement peut varier de -128 (branchement relatif vers l'arrière) à +127 (branchement relatif vers l'avant).

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

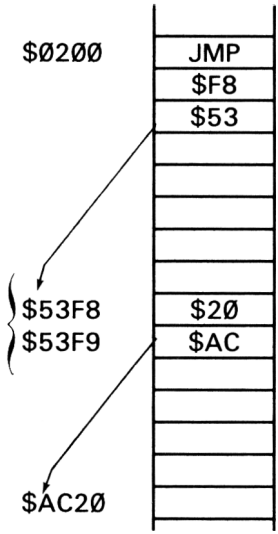
4.2.9. L'adressage indirect

Dans le 6502, ce mode d'adressage n'est utilisé que pour les sauts inconditionnels. La syntaxe assembleur est la suivante :

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

les parenthèses indiquant l'indirection.

L'instruction JMP est l'équivalent du GOTO en BASIC.
Dans le cas de l'adressage indirect, l'adresse effective de saut est donnée par le contenu de l'adresse spécifiée dans l'opérande.



Le microprocesseur, lorsqu'il rencontre l'instruction 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.

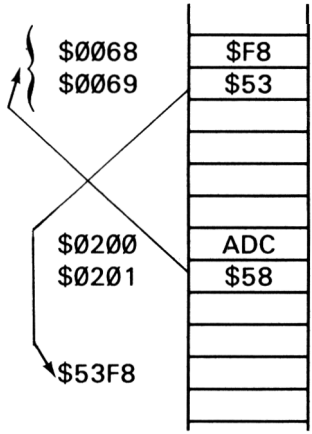
4.2.10 L'adressage pré-indexé indirect

Ce mode d'adressage est une combinaison de l'adressage indirect et de l'adressage indexé. La syntaxe assembleur est la suivante :

```
ADC ($58,X)
```

Comme nous le laisse prévoir cette écriture, le contenu du registre d'index X est additionné à l'adresse page-zéro (ici $58) qui sert de base. Soit (X) le contenu du registre X (ici on prendra par exemple (X) = $1Ø . Aux adresses $58+(X) et $58+(X)+1 (donc

$68 et $69) nous trouverons respectivement l'octet de poids faible et l'octet de poids fort de l'adresse effective sur laquelle s'effectue l'instruction ADC. L'état de la mémoire est le suivant :



Lorsque le microprocesseur rencontre l'instruction ADC, il se branche en $68 et $69 (page-zéro) où il trouve respectivement les valeurs $F8 et $53 qui représentent l'adresse effective $53F8.

Il effectue alors l'addition du contenu de cette case-mémoire avec l'accumulateur.

Notons que ce mode d'adressage n'est indexé que par le registre X. Il n'opère que sur la page-zéro donc ne permet d'accéder qu'à 256 octets (donc 128 adresses effectives). Ce mode d'adressage, bien que plus lent (au point de vue temps d'exécution) qu'un mode d'adressage direct classique, n'en reste pas moins très commode pour transférer des blocs de données de taille importante.

4.2.11. L'adressage post-indexé (par Y indirect)

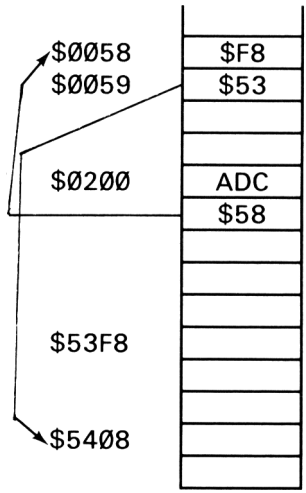
Contrairement au mode précédent, celui-ci est indexé par Y.

De plus il est d'usage moins fréquent. La syntaxe assembleur 65Ø2 est la suivante.

```
ADC ($58),Y
```

Le second octet de l'instruction ($58) contient une adresse en page-zéro. Celle-ci ainsi que la suivante contiennent deux octets qui forment une adresse et qui, additionnés au contenu du registre Y, donnent l'adresse effective sur laquelle s'effectue l'addition. Ici l'indexation est faite après l'indirection contrairement au cas de l'adressage pré-indexé indirect où l'indexation est faite avant l'indirection.

Grâce à ce mode d'adressage on peut accéder facilement à une table de valeurs située à n'importe quel endroit de l'espace adressable du 65Ø2, à condition d'avoir stocké l'adresse de base (adresse d'indirection) préalablement en page-zéro. L'état de la mémoire est le suivant (en supposant que le registre Y contient $1Ø par exemple).



Lorsque le microprocesseur rencontre l'instruction ADC, il se branche à l'adresse page-zéro $58 puis à l'adresse $59. Son PC se charge alors avec l'adresse

```
$53F8 + $1Ø = $54Ø8
```

Nous venons de passer en revue tous les modes d'adressage du 65Ø2. Comme vous pouvez le constater ils sont très nombreux (au nombre de 13) et certains d'entre eux sont extrêmement puissants. Leur bonne maîtrise vous permettra de développer efficacement vos programmes.

Il faut toujours garder bien présent à l'esprit qu'une bonne utilisation des possibilités du jeu d'instructions de son microprocesseur permet une réduction sensible de l'espace-mémoire occupé par le programme ainsi que de son temps d'exécution.

REMARQUE : La syntaxe assembleur donnée dans les exemples est celle d'un assembleur particulier du 65Ø2 (MOS-TECHNOLOGY). Elle peut donc varier sensiblement d'une machine à une autre sans qu'en aucune façon, les principes de fonctionnement n'en soient affectés.

5
Le jeu d'instruction du 6502

5.1. INTRODUCTION

Contrairement à ce qui se fait couramment, nous avons choisi de ne pas vous présenter les instructions du 65Ø2 dans l'ordre alphabétique mais par " affinités ", abondamment commentées et illustrées de nombreux exemples.

Le 65Ø2 possède 56 instructions différentes qui, combinées avec les différents modes d'adressage, portent ce nombre à 152 ce qui fait de ce microprocesseur un des plus puissants " 8 bits " du marché.

Ce n'est pas sans raison que tant de constructeurs de microordinateurs l'ont choisi. Nous les avons regroupées en dix 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
- l'instruction RTI
- les instructions sur la pile
- les instructions spéciales.

De plus, les exemples que nous vous proposons ont une difficulté croissante. Cela est à notre avis une meilleure solution que celle qui consiste à expliquer théoriquement le fonctionnement des instructions et ensuite à consacrer un chapitre entier à des listings de programmes plus ou moins commentés. Nous ne pourrons bien sûr pas donner un exemple pour chaque instruction et chaque mode d'adressage mais nous essaierons d'utiliser chacun d'entre eux chacun 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

5.2.1. Les instructions de chargement de registre

Il s'agit des trois instructions suivantes :

LDA, LDX, LDY

En bon anglais " LD " veut dire " Load" (charger).

Donc ces instructions permettent de charger respectivement l'accumulateur A, le registre d'index X, le registre d'index Y. 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 d'état qui sont affectés par cette instruction.

Mais, auparavant, effectuons un retour sur les bits N et Z du registre d'état P. Nous avons vu dans le chapitre précédent la signification de ces 2 bits :

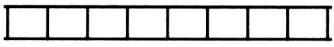
N = indicateur de résultat négatif.
Z = indicateur de zéro.

Dans les trois instructions LDA, LDX, LDY, 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 Ø. 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 fort poids (bit 7) représente alors le signe du nombre binaire.

bits 7 6 5 4 3 2 1 Ø



bit 7 = bit N

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

Exemple :

bit 7
↓
$17 = ØØØ1Ø111
d'où N = Ø

bit 7
↓
$FE = 11111110
d'où N = 1

a) **L'instruction LDA**

Grâce à cette instruction on peut charger l'accumulateur avec un nombre binaire (adressage immédiat) ou par le contenu d'une case-mémoire.

Instruction	Mode d'adressage	Code opération	Indicateurs affectés
LDA #xx	immédiat	A9	N,Z
LDA HHLL	étendu (direct)	AD	N,Z
LDA LL	direct page-zéro	A5	N,Z
LDA HHLL,X	étendu indexé (X)	BD	N,Z
LDA HHLL,Y	étendu indexé (Y)	B9	N,Z
LDA LL,X	indexé page zéro (X)	B5	N,Z
LDA (LL,X)	pré-indexé indirect	A1	N,Z
LDA (LL),Y	post-indexé indirect	B1	N,Z

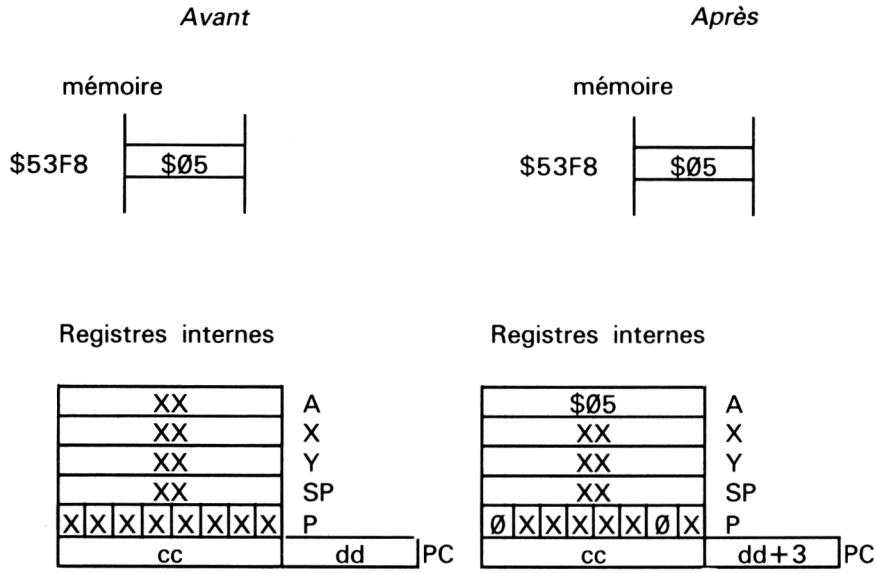
Explicitons tout d'abord un petit peu les notations employées. HHLL désigne une adresse étendue sur 16 bits ; HH désigne l'octet de poids fort et LL l'octet de poids faible de cette adresse ; xx désigne un nombre qui sera chargé directement dans l'accumulateur (adressage immédiat).

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

Soit l'instruction suivante :

```
LDA $53F8
```

Nous allons examiner le contenu des registres du 65Ø2 avant et après l'exécution de cette instruction en supposant que la case-mémoire $53F8 contient la valeur $Ø5.



Une croix " X " veut dire que le contenu du registre ou du bit est indéterminé. Avant l'exécution de l'instruction, l'accumulateur A et le registre d'état P contiennent des valeurs inconnues.

Après l'éxécution, A contient la valeur hexadécimale $Ø5 et le contenu du registre d'état P est modifié.

En effet

$Ø5 est différent de zéro ⇒ Z = Ø
$Ø5 est positif ⇒ N = Ø

Si l'instruction avait été :

LDA $53F8

avec

($53F8) = $ØØ, on aurait eu Z = 1 et N = Ø

REMARQUE : Une adresse hexadécimale entre parenthèses désigne le contenu de celle-ci.

Après exécution de l'instruction, le contenu du compteur ordinal est incrémenté de 3 puisque l'instruction `LDA $53F8` nécessite 3 octets.

En effet le contenu de la mémoire programme est le suivant :

ccdd	$AD
ccdd+1	$F8
ccdd+2	$53

On remarque que l'octet de poids faible est stocké le premier en mémoire.

b) **Les instructions LDX, LDY**

Nous avons choisi de regrouper ces deux instructions car elles sont totalement identiques.

Instruction	Mode d'adressage	Code opération	Indicateurs affectés
LDX #xx	immédiat	A2	N,Z
LDX HHLL	étendu	AE	N,Z
LDX LL	page zéro direct	A6	N,Z
LDX LL,Y	page zéro indexé (Y)	B6	N,Z
LDX HHLL,Y	étendu indexé (Y)	BE	N,Z
LDY #xx	immédiat	AØ	N,Z
LDY HHLL	étendu	AC	N,Z
LDY LL	page zéro direct	A4	N,Z
LDY LL,X	page zéro indexé (X)	B4	N,Z
LDY HHLL,X	étendu indexé (X)	BC	N,Z

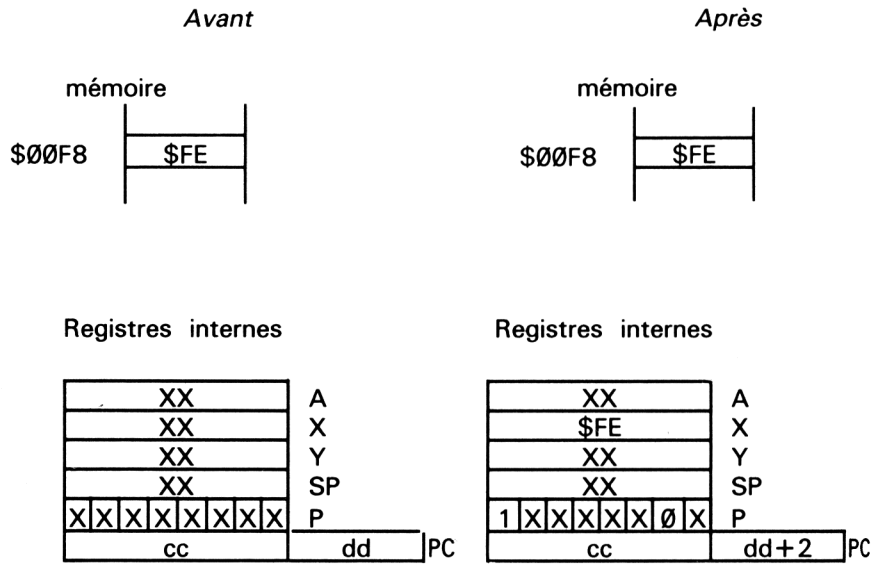
D'ores et déjà nous remarquons quelques points :

– Dans le cas de LDX, l'adressage indexé ne peut l'être que par Y et inversement pour LDY.

– Le fonctionnement est exactement le même que pour l'instruction LDA. Nous prenons ici l'exemple d'un adressage en page-zéro.

Soit l'instruction suivante : LDX $F8

Le contenu des registres est le suivant (la case-mémoire d'adresse $ØØF8 est supposée contenir la valeur $FE) :



Après l'exécution de cette instruction, le contenu du registre d'index X est $FE

or $FE = 1111111Ø

En notation en complément à 2, $FE est négatif et le bit N du registre P est positionné à 1.

5.2.2. Les instructions de chargement mémoire

Il s'agit des instructions

STA, STX, STY

qui sont exactement les équivalents des instructions
LDA, LDX, LDY

Les deux lettres " ST " veulent dire " store " (ranger)

Donc STA range le contenu de l'accumulateur A en mémoire, STX et STY les contenus des registres X et Y.

a) **L'instruction STA**

Instruction	Mode-d'adressage	Code-opération	Indicateurs affectés
STA HHLL	étendu	8D	aucun
STA LL	page zéro	85	aucun
STA HHLL,X	étendu indexé (X)	9D	aucun
STA HHLL,Y	étendu indexé (Y)	99	aucun
STA LL,X	page zéro indexé (X)	95	aucun
STA (LL,X)	pré-indexé indirect	81	aucun
STA (LL),Y	post-indexé indirect	91	aucun

En regardant ce tableau, on peut remarquer tout de suite les choses suivantes :

– bien sûr il n'y a pas d'adressage immédiat puisque l'instruction STA range le contenu de l'accumulateur dans une case-mémoire d'adresse définie.

– aucun des indicateurs du registre d'état P n'est affecté par cette instruction.

Nous allons illustrer son fonctionnement par un exemple d'adressage étendu indexé par X sachant que le problème serait exactement le même dans le cas d'une indexation par Y.

Soit à exécuter l'instruction suivante :

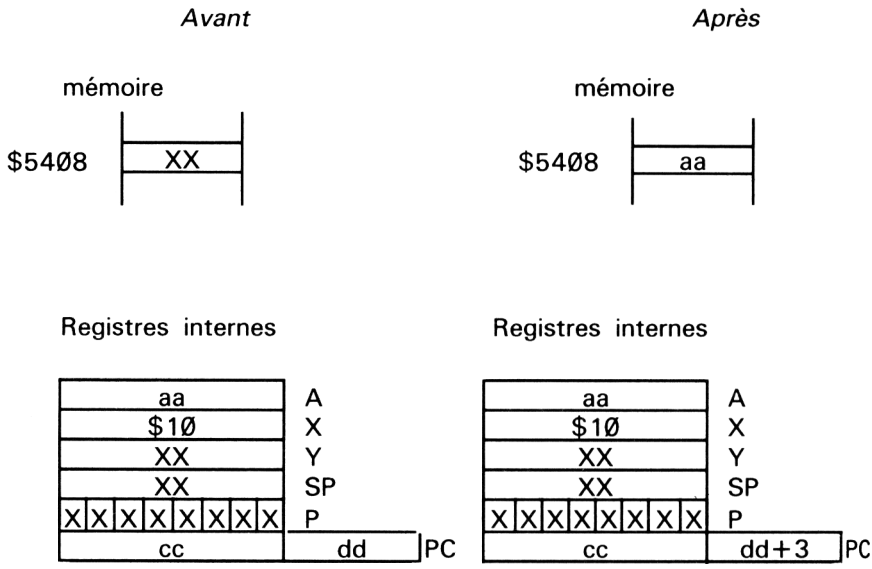
STA $53F8,X

et supposons que le registre X contienne initialement la valeur hexadécimale $1Ø.

Cette instruction range donc le contenu de l'accumulateur à l'adresse suivante :

$53F8 + $1Ø = $54Ø8

Examinons le contenu des registres internes du 65Ø2 avant et après l'exécution de l'instruction.



b) **Les instructions STX, STY**

De même que dans le cas des instructions LDX et LDY nous traitons ces instructions simultanément puisqu'elles sont en tous points identiques.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
STX HHLL	étendu	8E	aucun
STX LL	page zéro	86	aucun
STX LL,Y	page zéro indexé (Y)	96	aucun
STY HHLL	étendu	8C	aucun
STY LL	page zéro	84	aucun
STY LL,X	page zéro indexé (X)	94	aucun

Nous allons illustrer le fonctionnement de l'instruction STX par un exemple utilisant l'adressage page-zéro indexé (par Y bien sûr puisqu'il s'agit de l'instruction STX).

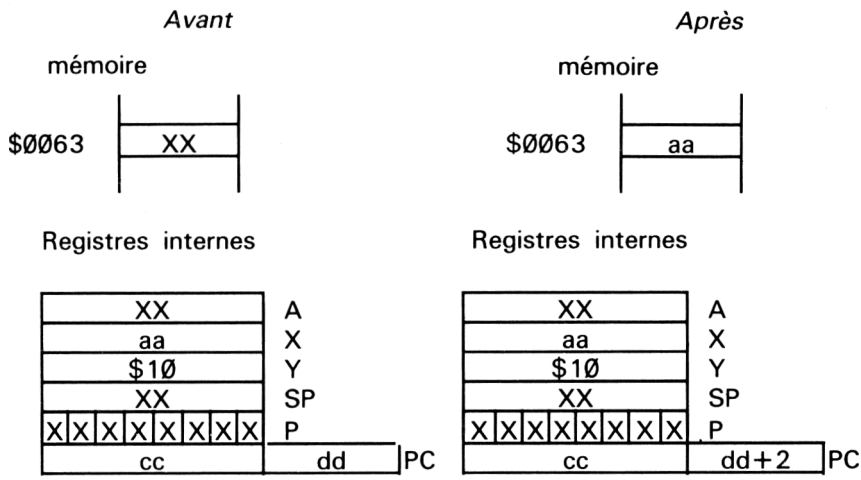
Soit à réaliser :

STX $53,Y

et supposons que Y contienne la valeur $1Ø. Alors le microprocesseur rangera le contenu du registre d'index X dans la case-mémoire d'adresse.

$53 + $1Ø = $63

Ceci est résumé dans les diagrammes suivants :



Nous venons de décrire le fonctionnement des instructions les plus employées dans tout programme " tournant sur 65Ø2 ". Inutile de vous dire donc qu'il est nécessaire et même vital que vous en ayez saisi, sinon toutes les subtilités, du moins les bases de fonctionnement.

Vous avez dû remarquer que nous n'avons jamais utilisé les modes d'adressage un peu particuliers que sont l'adressage pré-indexé indirect et l'adressage post-indexé indirect. Nous préférons laisser ceci à plus tard quand nous étudierons le fonctionnement d'autres

instructions. Pour l'instant contentez-vous de vous familiariser avec les modes d'adressage les plus courants qui sont ceux que nous avons déjà rencontrés.

5.2.3. Les instructions d'échange de registres

Nous regroupons sous cette appellation les instructions suivantes :

- TAX : transfert de l'accumulateur dans le registre X
- TAY : transfert de l'accumulateur dans le registre Y
- TSX : transfert du registre SP dans X
- TXA : transfert du registre X dans l'accumulateur
- TYA : transfert du registre Y dans l'accumulateur
- TXS : transfert du registre X dans SP.

Ces instructions ont toutes un mode d'action similaire et leur mode d'adressage est bien entendu implicite.

Instruction	Mode d'adressage	Code opération	Indicateurs affectés
TAX	implicite	AA	N,Z
TAY	implicite	A8	N,Z
TSX	implicite	BA	N,Z
TXA	implicite	8A	N,Z
TYA	implicite	98	N,Z
TXS	implicite	9A	aucun

Exemple : Prenons l'instruction TAX

A contient initialement $1Ø

Avant

Registres

$1Ø		A
XX		X
XX		Y
XX		SP
X X X X X X X X		P
cc	dd	PC

Après

Registres

$1Ø		A
$1Ø		X
XX		Y
XX		SP
Ø X X X X X Ø X		P
cc	dd+1	PC

Il faut noter que l'instruction TSX permet de connaître la valeur du pointeur de pile. Au contraire, l'instruction TXS permet de fixer le contenu de ce registre (SP) à une valeur déterminée.

Nous allons passer maintenant aux instructions arithmétiques traitées par le 65Ø2. Mais nous reviendrons bien évidemment sur toutes les instructions de chargement car, comme nous l'avons déjà dit, elles interviennent dans tout programme écrit en assembleur.

5.3. LES INSTRUCTIONS ARITHMETIQUES

Nous appellerons instructions arithmétiques les instructions suivantes :

- l'addition ADC
- la soustraction SBC
- l'incrémentation INC, INX, INY
- la décrémentation DEC, DEX, DEY.

5.3.1. L'instruction ADC

1) **Notion d'addition sur les nombres binaires**

Soit à additionner les deux nombres hexadécimaux suivants :

$Ø5 et $17

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

Leur représentation binaire est :

$Ø5 = ØØØØ Ø1Ø1
Ø 5

$17 = ØØØ1 Ø111
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	= ØØØ1 1ØØ (1 C)

Cela n'a donc rien de bien compliqué.

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

2) Retour sur le registre P

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

Nous avons vu dans le chapitre précédent le rôle succint de ces indicateurs.

V = indicateur de débordement (overflow)
C = indicateur de retenue (carry)
D = indicateur de mode décimal.

a) *Le bit C*

Supposons que nous ayons à effectuer la somme des 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
-----------------------------
 = $15 F        = (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 V*

Il faut tout d'abord retenir une chose : le microprocesseur effectue toujours de la même manière l'addition de deux nombres binaires qu'ils soient signés ou non. 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 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 la somme des bits 6 de deux nombres considérés donnera lieu à une retenue.

En fait l'indicateur V joue sur le bit 6 le même rôle que l'indicateur C sur le bit 7.

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

$4B = Ø1ØØ1Ø11 = 75

$71 = Ø1111ØØ1 = 113

```
   $4B          Ø1ØØ1Ø11      75
 + $71        + Ø1111ØØ1    + 113
-----------  -------------  -------
 = $BC        = 1Ø1111ØØ    = 188
```

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 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 (retenue sur l'addition des 2 bits 6).

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

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

```
  $-Ø1          11111111
+ $-Ø5        + 11111Ø11
---------     ----------
= $-Ø6        (1) 11111Ø1Ø
                  ↑
                bit de retenue
```

Dans ce cas il y a positionnement à 1 des deux indicateurs N et V. 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 N et V sont égaux, le résultat de l'addition est correct alors que quand ils sont différents, le résultat est faux et nécessite une correction adéquate afin de pouvoir être utilisé ultérieurement.

c) *le bit D*

Nous dirons seulement que lorsque ce bit est positionné à 1, le microprocesseur travaille en mode décimal donc avec des nombres codés BCD.

Exemple :

```
  Ø3          ØØØØØØ11
+ 15        + ØØØ1Ø1Ø1
---------   ----------
= 18        = ØØØ11ØØØ
```

Lorsque la somme des deux nombres est supérieure à 99, le bit C est positionné à 1.

3) **L'instruction ADC** *(en anglais Add with Carry)*

Cette instruction permet d'additionner à l'accumulateur le contenu d'une case-mémoire déterminée et le bit de retenue. Le résultat est donc :

R = Accumulateur + Mémoire + Retenue.

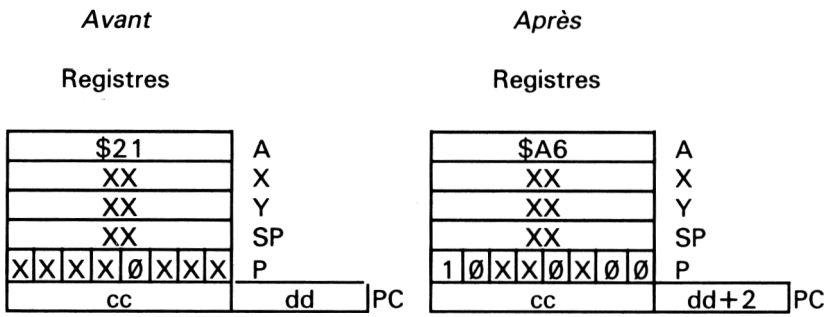
La particularité du 65Ø2 est que cette instruction peut s'effectuer aussi bien en mode décimal (BCD) qu'en mode binaire ce qui nous donne en fait l'équivalent de deux instructions.

Instruction	Mode d'adressage	Code opération	Indicateurs affectés
ADC # xx	immédiat	69	N,V,Z,C
ADC HHLL	étendu	6D	N,V,Z,C
ADC LL	page zéro	65	N,V,Z,C
ADC HHLL,X	étendu indexé (X)	7D	N,V,Z,C
ADC HHLL,Y	étendu indexé (Y)	79	N,V,Z,C
ADC LL,X	page zéro indexé (X)	75	N,V,Z,C
ADC (LL,X)	pré-indexé indirect	61	N,V,Z,C
ADC (LL),Y	post-indexé indirect	71	N,V,Z,C

Soit l'instruction ADC #$85 qui consiste à additionner la valeur hexadécimale $85 au contenu de l'accumulateur ($21 par exemple). On suppose que l'on est en mode binaire.

On a $85 + $21 = $A6

Représentons les registres internes du 65Ø2 avant et après l'exécution de .l'instruction.



Nous allons maintenant vous proposer un petit programme qui effectue une addition sur 8 bits de deux nombres situés à deux adresses mémoire en page-zéro, par exemple $ØØØØ et $ØØB6. Le résultat sera stocké à l'adresse $ØØØ1.

```
CLC          ; MISE A ZERO DU BIT C
LDA $ØØ      ; CHARGEMENT DU 1ER NOMBRE
ADC $B6      ; ADDITIONNE LE SECOND
STA $Ø1      ; RANGEMENT EN MEMOIRE
BRK
```

L'instruction CLC, nous le verrons plus loin, sert à annuler le bit C qui, s'il était positionné à 1, fausserait le résultat de l'addition. En effet, on se souvient que l'instruction effectue : Accumulateur + mémoire + retenue. L'instruction BRK est une interruption logicielle que nous décrirons également plus loin et qui sert à arrêter le déroulement du programme.

5.3.2. L'instruction SBC

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 :

$2Ø - $15 = $2Ø + ($-15)

on a
$2Ø = ØØ1ØØØØØ
$15 = ØØØ1Ø1Ø1
$-15 = 111Ø1Ø11

donc $2Ø - $15 = (1)ØØØØ1Ø11 = $1ØB
retenue

La différence $2Ø-$15 est un nombre positif ($ØB) et la retenue est alors égale à 1.

Supposons maintenant que nous voulions faire $15-$2Ø

$ - 2Ø = 111ØØØØØ

donc $15 - $2Ø = 1111Ø1Ø1 = $F5 = $ - ØB

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 = Ø = le résultat est négatif.

Examinons le cas de la soustraction en BCD.

Nous savons qu'avec un codage BCD un octet représente 2 chiffres et donc nécessairement un nombre positif. Comme dans le cas d'une soustraction binaire, la soustraction décimale se résume à une addition.

On additionne en effet au premier nombre le complément à 1ØØ du deuxième.

Donc pour effectuer 1Ø-28 on fait 1Ø + 72 = 82 avec une retenue égale à zéro qui indique un résultat négatif. Le résultat trouvé est le complément à 1ØØ de 18 (1Ø-28 = -18).

En arithmétique BCD comme en arithmétique binaire l'état de la retenue indique le signe du résultat :

si C = 1 le résultat est positif
si C = Ø le résultat est négatif.

2) **L'instruction SBC** *(en anglais substract with carry)*

Cette instruction soustrait de l'accumulateur le contenu d'une case-mémoire déterminée et le complément de la retenue (c'est-à-dire 1-C). Comme dans le cas de l'instruction ADC, elle peut s'effectuer aussi bien en mode binaire qu'en mode décimal selon l'état du bit D.

Instruction	Mode d'adressage	Code opération	Indicateurs affectés
SBC #xx	immédiat	E9	N,V,Z,C
SBC HHLL	étendu	ED	N,V,Z,C
SBC LL	page zéro	E5	N,V,Z,C
SBC HHLL,X	étendu indexé (X)	FD	N,V,Z,C
SBC HHLL,Y	étendu indexé (Y)	F9	N,V,Z,C
SBC LL,X	page zéro indexé (X)	F5	N,V,Z,C
SBC (LL,X)	pré-indexé indirect	E1	N,V,Z,C
SBC (LL),Y	post-indexé indirect	F1	N,V,Z,C

Nous allons tout d'abord illustrer cette instruction par un exemple d'adressage post-indexé indirect.

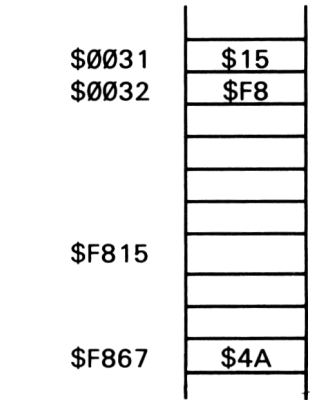
Soit donc l'instruction

SBC ($31),Y

On suppose que A contient $12, Y contient $52, les adresses $ØØ31 et $ØØ32 contiennent respectivement $15 et $F8, et C = Ø. L'état de la mémoire et des registres avant et après l'exécution de cette instruction sont les suivants ;

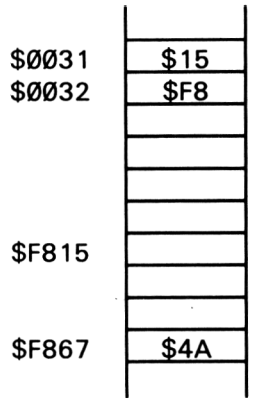
Avant

mémoire

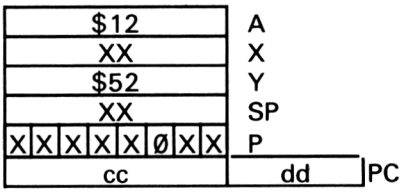


Après

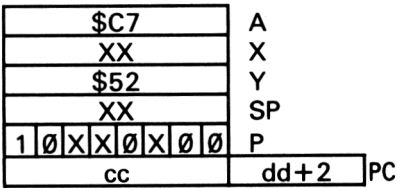
mémoire



Registres internes



Registres internes



Le microprocesseur se branche à l'adresse $F815 + $52 = $F867 qui contient par exemple la valeur hexadécimale $4A.

On effectue donc

$$\$12 - \$4A - \overline{C} = \$12 - \$4A - 1 = \$12 - \$4B$$
$$= \$C7$$
$$= 11ØØØ111$$

Donc les bits du registre P sont égaux à :

N = 1
V = Ø
Z = Ø
C = Ø
D = Ø

Exemple de programme de soustraction sur 8 bits

On suppose que les deux nombres à soustraire sont situés à deux adresses en page-zéro, par exemple $ØØØØ et $ØØB6.

```
SEC          ; MISE A 1 DU BIT C
LDA $ØØ      ; CHARGEMENT DU 1ER NOMBRE
SBC $B6      ; RETRANCHEMENT DU SECOND
STA $Ø1      ; RANGEMENT EN MEMOIRE
BRK
```

L'instruction SEC, nous le verrons plus loin, sert à positionner à 1 le bit C du registre d'état qui, s'il était à Ø, fausserait le résultat de la soustraction (on se souvient que l'instruction SBC retranche au contenu de l'accumulateur le contenu de la case-mémoire désignée par l'opérande et le complément de la retenue, c'est-à-dire Ø si C = 1).

5.3.3. Programme d'addition sur 16 bits

Nous savons que le 65Ø2 effectue des additions sur 8 bits donc sur des nombres variant de Ø à 255 ou de -128 à +127 en arithmétique signée. Mais il peut être nécessaire d'augmenter la précision des calculs et notamment passer à des opérations sur 16 bits.

Considérons deux nombres A et B de 16 bits qui nécessitent donc quatre cases mémoire pour être stockés.

Soient :

OPHA = adresse de l'octet de poids fort de A
OPBA = adresse de l'octet de poids faible de A
OPHB = adresse de l'octet de poids fort de B
OPBB = adresse de l'octet de poids faible de B
RESB = adresse de l'octet de poids faible du résultat
RESH = adresse de l'octet de poids fort du résultat.

Le programme s'écrit :

```
CLD         ; MISE EN MODE BINAIRE
CLC         ; MISE A ZERO DE LA RETENUE
LDA OPBA    ; CHARGEMENT OCTET POIDS FAIBLE DE A
ADC OPBB    ; ADDITIONNE OCTET POIDS FAIBLE DE B
STA RESB    ; RANGEMENT EN MEMOIRE PARTIE BASSE
LDA OPHA    ; CHARGEMENT OCTET POIDS FORT DE A
ADC OPHB    ; ADDITIONNE OCTET POIDS FORT DE B
STA RESH    ; RANGEMENT EN MEMOIRE PARTIE HAUTE
BRK
```

Au début, on met à Ø (voir plus loin l'instruction CLD) le bit de mode décimal (on est donc en arithmétique binaire), puis on met de même à Ø le bit C. On additionne les parties basses des nombres A et B et selon le cas il y a génération ou non d'une retenue. La partie basse du résultat est stockée en mémoire. Puis on fait la somme des parties hautes des nombres A et B et du bit C (qui contient toujours la retenue générée par l'addition des 2 octets de poids faible). La partie haute de A + B est alors stockée en mémoire.

La somme de A et B peut être, suivant les cas, un nombre de 16 ou de 17 bits. Il faudra donc éventuellement tester le bit C après la deuxième addition. De plus, si l'on travaille en arithmétique signée il sera nécessaire de tester le bit V et de corriger, le cas échéant, la partie haute du résultat.

5.3.4. Les instructions d'incrémentation

Nous regroupons sous cette appellation les instructions INC, INX et INY. L'instruction INC effectue une incrémentation en mémoire tandis que les instructions INX et INY incrémentent directement les contenus des registres X et Y. Il s'agit donc pour ces deux dernières d'un adressage implicite.

1) **Notion d'incrémentation**

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

Exemple : $Ø5 → $Ø6

Attention : L'incrémentation ne marche de façon convenable qu'en arithmétique binaire. En effet $09 → $0A et non pas $10

$09 = 00001001

après incrémentation cela donne 00001010 = $0A

Donc ces instructions ne marchent pas en BCD

2) Instructions INC, INX et INY

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
INX	implicite	E8	N,Z
INY	implicite	C8	N,Z
INC HHLL	étendu	EE	N,Z
INC LL	page zéro	E6	N,Z
INC HHLL,X	étendu indexé (X)	FE	N,Z
INC LL,X	page zéro indexé (X)	F6	N,Z

Exemple d'utilisation de l'instruction INX

On suppose que X = $FF = ($-01)

L'état des registres internes du 6502 est le suivant :

Avant

Registres internes

XX		A
$FF		X
XX		Y
XX		SP
X X X X X X X X		P
cc	dd	PC

Après

Registres internes

XX		A
$00		X
XX		Y
XX		SP
0 X X X X X 1 X		P
cc	dd+1	PC

5.3.5. Les instructions de décrémentation

Nous regroupons sous cette appellation les instructions DEC, DEX et DEY.

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.

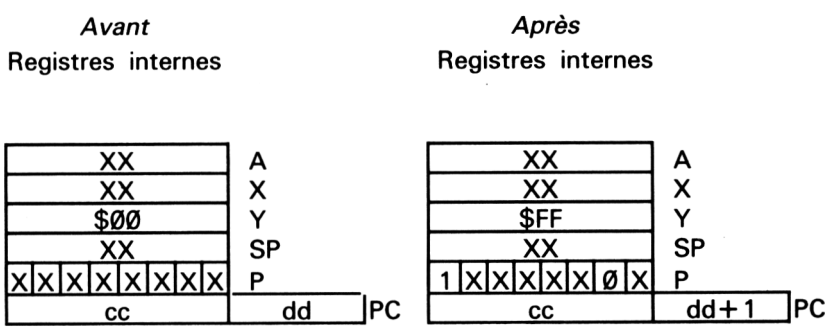
Exemple : $Ø5 → $Ø4

2) **Instructions DEC, DEX et DEY**

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
DEX	implicite	CA	N,Z
DEY	implicite	88	N,Z
DEC HHLL	étendu	CE	N,Z
DEC LL	page zéro	C6	N,Z
DEC HHLL,X	étendu indexé (X)	DE	N,Z
DEX LL,X	page zéro indexé (X)	D6	N,Z

Exemple d'utilisation de l'instruction DEY

On suppose que Y contient initialement $ØØ
L'état des registres du 65Ø2 est le suivant :



Nous en avons maintenant terminé avec les instructions arithmétiques. Nous allons maintenant passer aux instructions logiques qui sont également très utilisées.

5.4. LES INSTRUCTIONS LOGIQUES

Nous appellerons instructions logiques les instructions suivantes :

- " ET " : AND
- " OU " : ORA
- " OU exclusif " : EOR
- décalage à gauche ASL, ROL
- décalage à droite LSR, ROR

5.4.1. L'instruction AND

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 sont le " ET ", le " OU ", le " OU exclusif " et le " NON ". Dans le 65Ø2 seules les trois premières sont présentes.

Le " ET " logique : On considère deux chiffres A et B binaires. L'opération " ET " (notée souvent $\wedge$) est définie de la manière suivante :

Si A = B = 1 alors A $\wedge$ B = 1
sinon A $\wedge$ B = Ø

On peut définir une table de vérité pour cette opération.

B \ A	Ø	1
Ø	Ø	Ø
1	Ø	1

Les chiffres inscrits dans les 4 cases centrales donnent la valeur de A ∧ B pour chaque combinaison de A et de 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
```

2) L'instruction AND

Cette instruction effectue le " ET " logique entre le contenu d'une case-mémoire déterminée et l'accumulateur, le résultat final étant stocké dans ce dernier.

Instruction	Mode d'adressage	Code opération	Indicateurs affectés
AND #xx	immédiat	29	N,Z
AND HHLL	étendu	2D	N,Z
AND LL	page zéro	25	N,Z
AND HHLL,X	étendu indexé (X)	3D	N,Z
AND HHLL,Y	étendu indexé (Y)	39	N,Z
AND LL,X	page zéro indexé (X)	35	N,Z
AND (LL,X)	pré-indexé indirect	21	N,Z
AND (LL),Y	post-indexé indirect	31	N,Z

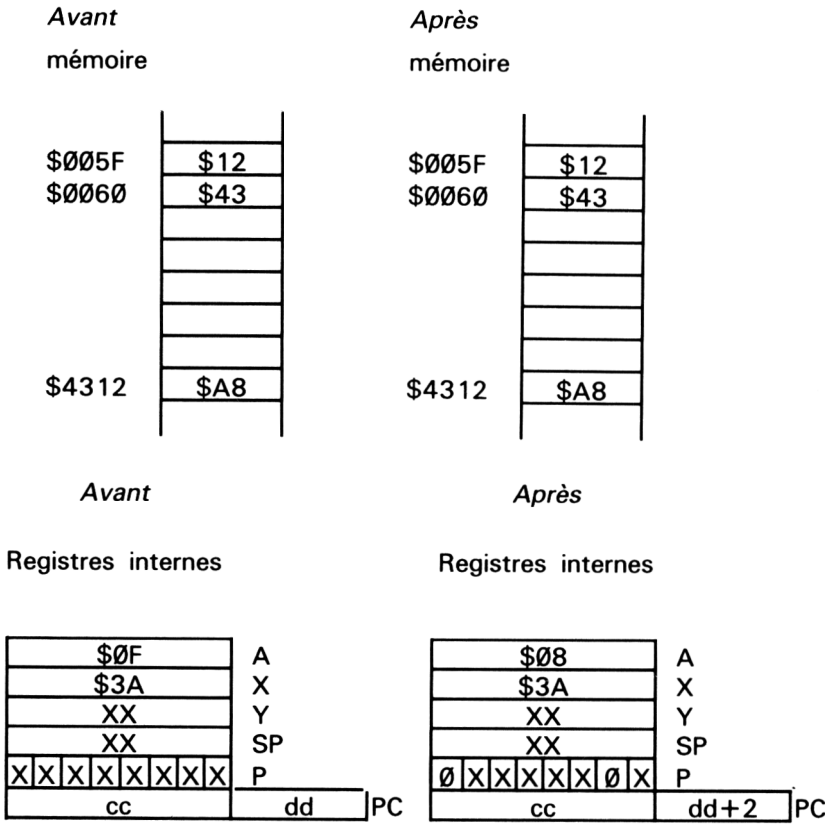
Exemple d'application utilisant un adressage pré-indexé indirect :

Soit l'instruction

```
AND ($25,X)
```

On suppose que X contient $3A et que les adresses $25 + $3A = $5F et $25 + $3A + 1 = $6Ø contiennent respectivement $12 et $43.

On suppose de plus que l'adresse $4312 contient la valeur $A8 et que l'accumulateur contient initialement $ØF.



Après l'exécution de l'instruction, l'accumulateur contient la valeur $Ø8.

En effet

$ØF = ØØØØ1111
$A8 = 1Ø1Ø1ØØØ

$ØF∧$A8 = ØØØØ1ØØØ = $Ø8

Dans cet exemple, nous voyons apparaître une technique très utilisée qui est celle du masquage. En effet, en faisant un " ET " entre $A8 et $ØF, on n'a conservé dans le résultat que les 4 bits de poids faible de $A8, les 4 autres ayant été mis à zéro (en effet x ∧ Ø = Ø quel que soit x).

Le technique du masquage est utilisée en général pour ne conserver dans un octet que les bits " utiles ".

On prendra souvent pour l'instruction AND un adressage immédiat avec comme opérande un " masque " donné sous forme binaire.

Exemple : on veut faire directement l'opération effectuée précédemment.

$A8 → $Ø8

on écrira alors
```
LDA #$A8
AND #%ØØØØ1111
```

5.4.2. L'instruction ORA

1) **Le " OU " logique** (noté souvent V)

L'opération " OU " est définie de la manière suivante :

si

A = B = Ø alors A V B = Ø

sinon

A V B = 1

La table de vérité de cette opération est la suivante :

B \ A	Ø	1
Ø	Ø	1
1	1	1

Comme dans le cas de l'opération " ET ", le " OU " entre 2 octets s'effectue bit par bit.

Exemple : Soit à effectuer $12 V $35

```
    $12 = ØØØ1ØØ1Ø
    $35 = ØØ11Ø1Ø1
    ---------------
$12 V $35 = ØØ11Ø111 = $37
```

2) L'instruction ORA

Cette opération effectue le " OU " logique entre le contenu d'une case-mémoire et l'accumulateur.

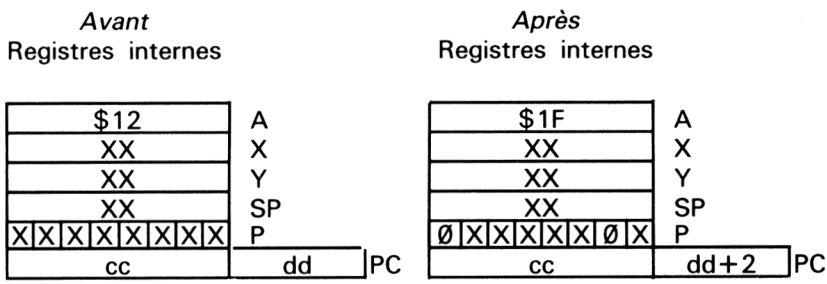
Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
ORA #xx	immédiat	Ø9	N,Z
ORA HHLL	étendu	ØD	N,Z
ORA LL	page zéro	Ø5	N,Z
ORA HHLL,X	étendu indexé (X)	1D	N,Z
ORA HHLL,Y	étendu indexé (Y)	19	N,Z
ORA LL,X	page zéro indexé (X)	15	N,Z
ORA (LL,X)	pré-indexé indirect	Ø1	N,Z
ORA (LL),Y	post-indexé indirect	11	N,Z

Exemple : Soit à effectuer $12 V $ØF

L'instruction s'écrit (adressage immédiat)

```
                ORA #$ØF
ou bien         ORA #%ØØØØ1111
```

L'état des registres du 65Ø2 est le suivant :



Après l'exécution de l'instruction, l'accumulateur contient la valeur $1F

En effet

```
    $12 = ØØØ1ØØ1Ø
    $ØF = ØØØØ1111
    ---------------
$12 V $ØF = ØØØ11111 = $1F
```

Nous avons vu précédemment que l'instruction AND pouvait permettre de masquer certains bits et donc de les mettre à zéro.

Au contraire l'instruction ORA, nous le voyons ici, peut permettre de positionner à 1 certains bits d'un octet (car x V 1 = 1 quel que soit x)

5.4.3. L'instruction EOR

1) **Le " OU " exclusif** *(noté souvent ⊻)*

L'opération " OU exclusif " est définie de la manière suivante :

si A = B alors A ⊻ B = Ø
si A # B alors A ⊻ B = 1

La table de vérité de cette opération est la suivante :

B \ A	Ø	1
Ø	Ø	1
1	1	Ø

Le " OU exclusif " sur un octet s'effectue bit par bit comme dans le cas du " ET " et du " OU ".

Exemple : Soit à effectuer $12 ⊻ $35

```
$12 = ØØØ1ØØ1Ø
$35 = ØØ11Ø1Ø1
-------------
$12 ⊻ $35 = ØØ1ØØ111 = $27
```

2) **L'instruction EOR**

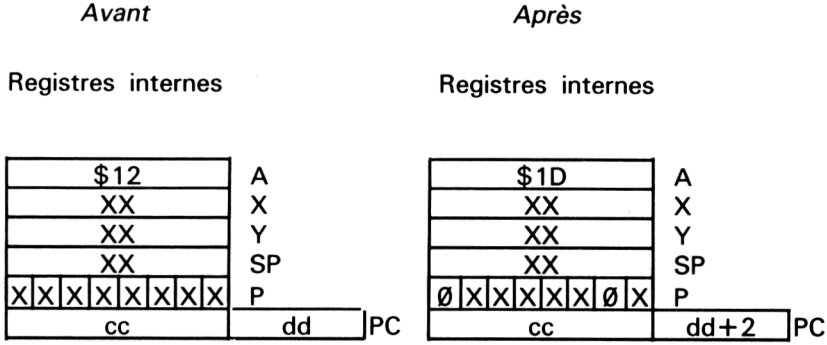
Cette opération effectue le " OU exclusif " entre le contenu d'une case-mémoire et l'accumulateur.

Instruction	Mode d'adressage	Code opération	Indicateurs affectés
EOR #xx	immédiat	49	N,Z
EOR HHLL	étendu	4D	N,Z
EOR LL	page zéro	45	N,Z
EOR HHLL,X	étendu indexé (X)	5D	N,Z
EOR HHLL,Y	étendu indexé (Y)	59	N,Z
EOR LL,X	page zéro indexé (X)	55	N,Z
EOR (LL,X)	pré-indexé indirect	41	N,Z
EOR (LL),Y	post-indexé indirect	51	N,Z

Exemple : Soit à effectuer $12 ⊻ $ØF

L'instruction s'écrit EOR #%ØØØØ1111

L'état des registres est le suivant :



Après l'exécution de l'instruction l'accumulateur contient la valeur $1D

($12 ⊻ $ØF) = $1D

Nous voyons que l'instruction EOR peut être utilisée pour inverser certains bits d'un octet.

Ici par exemple les 4 bits de poids faible de $12 ont été inversés à cause du " F " de $ØF pour donner $1D (en effet X ⊻ 1 = $\overline{X}$, $\overline{X}$ désignant le complément de X, ceci quel que soit X).

5.4.4. Les instructions de décalage

Parmi les instructions logiques, les instructions de décalage sont également très utilisées. On pourra citer comme exemple d'application des programmes de multiplication et de division.

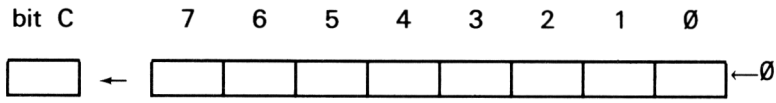
Les instructions de décalage sont au nombre de 4.

– ASL : Arithmetic Shift Left : décalage arithmétique vers la gauche

- LSR : Logical Shift Right : décalage logique vers la droite
- ROL : rotate left = rotation vers la gauche
- ROR : rotate right = rotation vers la droite.

1) **Fonctionnement des instructions de décalage** *(ASL et LSR)*

a) *ASL*



Chaque bit est décalé d'un rang vers la gauche. Le bit Ø est alors remplacé par un " Ø " et le bit 7 devient le bit de retenue (bit C).

On notera que le bit de signe (bit 7) est conservé dans la retenue d'où le nom de décalage " arithmétique ".

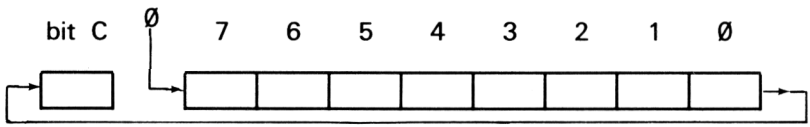
Exemple : Soit l'instruction ASL A (adressage accumulateur)

on suppose que A = $9D initialement.

$9D = 1ØØ111Ø1

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

b) *LSR*



Ici chaque bit est décalé vers la droite. Le bit 7 est remplacé par un Ø et le bit " Ø " est transféré dans la retenue. Ici le bit de signe est perdu d'où le nom de décalage " logique ".

Exemple : Soit l'instruction LSR A (adressage accumulateur) et A = $9D initialement.

Après décalage cela nous donne

$4E = Ø1ØØ111Ø et C = 1

2) Les instructions ASL et LSR

Ces instructions effectuent un décalage soit sur l'accumulateur soit sur une case-mémoire donnée.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
ASL A	Accumulateur	ØA	N,Z,C
ASL HHLL	étendu	ØE	N,Z,C
ASL LL	page zéro	Ø6	N,Z,C
ASL HHLL,X	étendu indexé (X)	1E	N,Z,C
ASL LL,X	page zéro indexé (X)	16	N,Z,C
LSR A	Accumulateur	4A	N,Z,C N = Ø
LSR HHLL	étendu	4E	N,Z,C N = Ø
LSR LL	page zéro	46	N,Z,C N = Ø
LSR HHLL,X	étendu indexé (X)	5E	N,Z,C N = Ø
LSR LL,X	page zéro indexé (Y)	56	N,Z,C N = Ø

Exemple 1 : Soit l'instruction ASL A avec A = $9D

L'état des registres internes est le suivant :

Avant

Registres internes

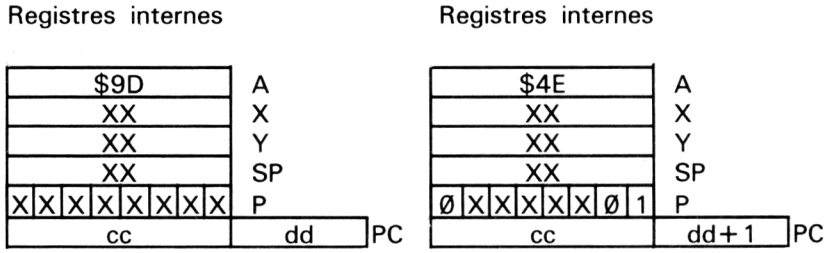
$9D	A
XX	X
XX	Y
XX	SP
X X X X X X X X	P
cc dd	PC

Après

Registres internes

$3A	A
XX	X
XX	Y
XX	SP
Ø X X X X X Ø 1	P
cc dd+1	PC

Exemple 2 : Soit l'instruction LSR A avec A = $9D



Dans le cas de l'instruction LSR, le bit de signe N du registre d'état P est toujours positionné à Ø.

REMARQUE : Un décalage à droite d'un rang équivaut à une division par 2.

Un décalage à gauche d'un rang équivaut à une multiplication par 2.

3) **Programme de conversion BCD → binaire**

On considère un octet d'adresse $ØØØØ contenant deux chiffres BCD et que l'on veut convertir en sa valeur binaire.
On prend comme exemple

23 = ØØ1ØØØ11 (en BCD)

on a

23 = 2 × 1Ø + 3

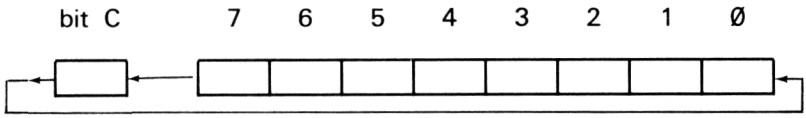
Le problème est donc d'obtenir le " 2 ", le " 3 " et de multiplier 2 par 1Ø. Or on sait que 1Ø = 8 + 2 donc on pourra obtenir facilement le résultat escompté par des décalages adéquats.

Le listing du programme est le suivant :

```
LDA  $ØØ          ; CHARGE CHIFFRE BCD DE POIDS FAIBLE (LSB)
AND  #%ØØØØ1111
STA  $Ø1          ; SAUVEGARDE EN MEMOIRE
LDA  $ØØ          ; CHARGE CHIFFRE BCD DE POIDS FORT (MSB)
AND  #%1111ØØØØ
LSR  A            ; A CONTIENT 8 FOIS MSB
STA  $Ø2          ; SAUVEGARDE DE A
LSR  A            ; A CONTIENT 4 FOIS MSB
LSR  A            ; A CONTIENT 2 FOIS MSB
CLC
CLD               ; MODE BINAIRE
ADC  $Ø2          ; A CONTIENT 10 FOIS MSB
ADC  $Ø1          ; A CONTIENT 10 FOIS MSB + LSB
STA  $Ø2          ; RANGE EN MEMOIRE LE RESULTAT
BRK
```

4) **Fonctionnement des instructions de rotation** *(ROR et ROL)*

a) *ROL*



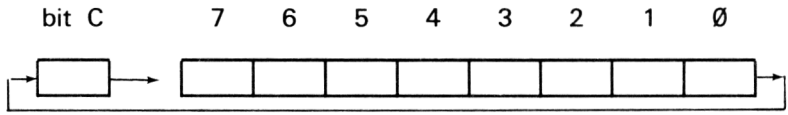
Dans l'instruction ROL, chaque bit est décalé d'un rang vers la gauche. Mais, contrairement à l'instruction ASL, le bit " Ø " n'est pas remplacé par un Ø mais par le bit de retenue. Il y a donc la rotation suivante :

Ø → 1 → 2 → 3 → 4 → 5 → 6 → 7 → C → Ø etc...

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

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

b) *ROR*



Le fonctionnement est exactement le même que pour l'instruction ROL sauf que le décalage se fait vers la droite.

Exemple : Soit l'instruction ROR A avec A = $9D et C = 1 cela nous donne alors

$CE = 11ØØ111Ø et C = 1

5) **Les instructions ROR et ROL**

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
ROL A	Accumulateur	2A	N,Z,C
ROL HHLL	étendu	2E	N,Z,C
ROL LL	page zéro	26	N,Z,C
ROL HHLL,X	étendu indexé (X)	3E	N,Z,C
ROL LL,X	page zéro indexé (X)	36	N,Z,C
ROR A	Accumulateur	6A	N,Z,C
ROR HHLL	étendu	6E	N,Z,C
ROR LL	page zéro	66	N,Z,C
ROR HHLL,X	étendu indexé (X)	7E	N,Z,C
ROR LL,X	page zéro indexé (X)	76	N,Z,C

5.4.5. L'instruction BIT

1) **Fonctionnement de cette instruction**

En fait il s'agit d'un " ET " logique un peu particulier dans le sens où ni l'accumulateur ni la case-mémoire désignée par l'opérande ne sont modifiés. Seul le registre d'état P est affecté et traduit le résultat de l'opération.

Nous décrirons un exemple dans le prochain paragraphe. D'ores et déjà nous pouvons dire que l'instruction BIT sert, comme son

nom l'indique, à tester un ou plusieurs bits d'un octet. En général, cette instruction est suivie par une instruction de branchement appropriée traduisant une condition sur un des bits du registre d'état P.

2) L'instruction BIT

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
BIT HHLL	étendu	2C	N,V,Z
BIT LL	page zéro	24	N,V,Z

Nous allons expliquer en détail la manière dont sont affectés les bits N, V et Z du registre d'état P.

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 ∧ (M) = Ø
Z = Ø si A ∧ (M) # Ø

b) *le bit N*

Le bit N est la recopie du bit 7 de la case-mémoire désignée par l'opérande.

c) *le bit V*

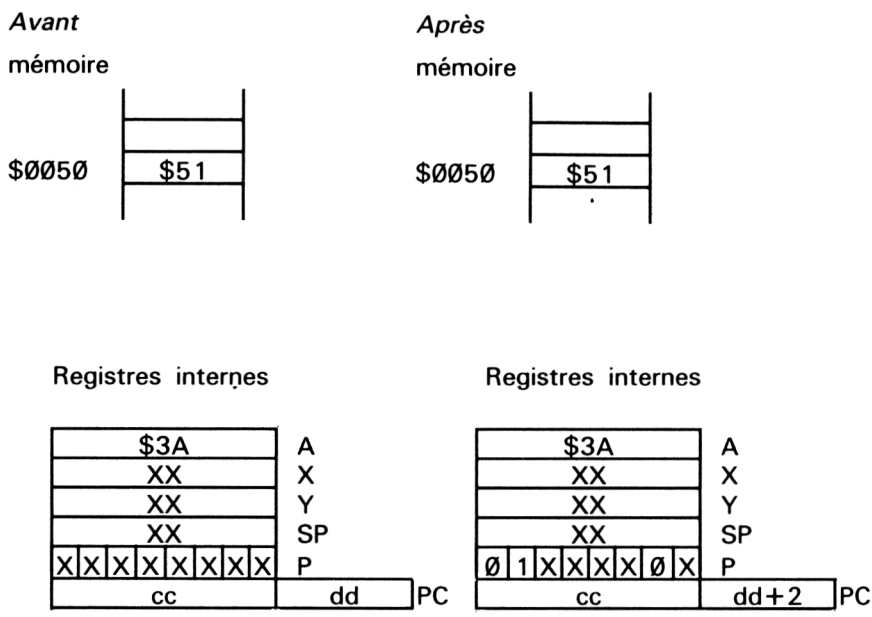
Le bit V est la recopie du bit 6 de la case-mémoire désignée par l'opérande.

Exemple : Soit l'instruction BIT $5Ø
on suppose que ($5Ø) = $51 et que l'accumulateur contient $3A

$51 = Ø1Ø1ØØØ1
$3A = ØØ111Ø1Ø

$51 ∧ $3A = ØØØ1ØØØØ

Etat des registres internes du 6502 :



Le bit 7 de $51 est égal à 0 donc N = 0

Le bit 6 de $51 est égal à 1 donc V = 1

$$\$51 \wedge \$3A = \$10 \neq 0 \text{ donc } Z = 0$$

Nous reviendrons sur toutes ces opérations logiques lorsque nous aurons vu les instructions de branchement et nous vous proposerons alors divers exemples d'application.

5.5. LES INSTRUCTIONS SUR LE REGISTRE D'ETAT

Elles sont au nombre de 7. Elles permettent de positionner certains bits de ce registre à une valeur déterminée initialement.

Nous en avons déjà vu deux (CLC et SEC) lors du chapitre concernant les instructions arithmétiques.

Nous allons passer en revue ces différentes instructions :

- CLC : Clear carry = mise à zéro du bit de retenue
- SEC : Set carry = mise à un du bit de retenue
- CLD : mise à zéro du bit indicateur de mode décimal (donc on est en mode binaire)
- SED : mise à un du bit indicateur de mode décimal (on est donc en mode décimal)
- CLI : mise à zéro du masque d'interruptions → interruptions autorisées
- SEI : mise à un du masque d'interruptions → interruptions interdites
- CLV : mise à zéro du bit de dépassement.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
CLC	implicite	18	C = Ø
SEC	implicite	38	C = 1
CLD	implicite	D8	D = Ø
SED	implicite	F8	D = 1
CLI	implicite	58	I = Ø
SEI	implicite	78	I = 1
CLV	implicite	B8	V = Ø

Il est probable que vous ne vous servirez jamais des instructions CLI et SEI puisque la maîtrise des interruptions (dont nous parlerons lors de la description de l'instruction RTI) ne peut être acquise qu'avec une très bonne connaissance du matériel que l'on utilise.

Programme d'addition en BCD (sur 4 chiffres)

On considère deux nombres compris entre Ø et 9999. Le premier est situé aux adresses $ØØØØ et $ØØØ1, le deuxième aux adresses $ØØØ2 et $ØØØ3. En $ØØØØ et $ØØØ2 on trouve les chiffres des unités et des dizaines tandis qu'en $ØØØ1 et $ØØØ3 on trouve les chiffres des centaines et des milliers. Le résultat sera stocké aux adresses $ØØØ4 et $ØØØ5.

Le programme est le suivant :

```
SED         ; MISE EN MODE DECIMAL
CLC
LDA $ØØ     ; CHARGE LES 2 CHIFFRES DE POIDS FAIBLE
ADC $Ø2     ; EFFECTUE L'ADDITION (DIZAINES + UNITES)
STA $Ø4     ; STOCKAGE EN MEMOIRE
LDA $Ø1     ; CHARGE LES 2 CHIFFRES DE POIDS FORT
ADC $Ø3     ; EFFECTUE L'ADDITION (CENTAINES + MILLIERS)
STA $Ø5     ; STOCKAGE EN MEMOIRE
BRK
```

Nous aurions pu utiliser également un adressage indexé et le programme aurait été le suivant :

```
        SED           ; MISE EN MODE DECIMAL
        CLC
        LDX #$ØØ     ; CHARGEMENT DU COMPTEUR DE BOUCLE
BOUCLE  LDA $ØØ,X     ; POINTE SUR LES CHIFFRES DES DIZAINES + UNITES
        ADC $Ø2,X     ; EFFECTUE L'ADDITION
        STA $Ø4,X     ; STOCKAGE EN MEMOIRE
        INX
        CMP $Ø2       ; FIN DE BOUCLE
        BNE BOUCLE    ; NON, CHARGE CHIFFRES CENTAINES + MILLIERS
        BRK           ; OUI, TERMINE
```

L'instruction BEQ sera expliquée un petit peu plus loin. Le programme boucle tant que le compteur représenté par le registre d'index X est inférieur ou égal à 2. Lorsqu'il passe à la valeur $Ø2, il y a exécution de l'instruction BRK.

5.6. LES INSTRUCTIONS DE COMPARAISON

Il s'agit des instructions CMP, CPX et CPY. Elles fonctionnent toutes les trois de manière identique sauf qu'elles agissent respectivement sur l'accumulateur, le registre X et le registre Y. Dans l'instruction CMP, le contenu de la case-mémoire désignée par l'opérande est retranché au contenu de l'accumulateur mais sans

en affecter la valeur (un peu comme pour l'instruction BIT). Il y a alors positionnement des indicateurs du registre d'état P en fonction du résultat de cette différence.

Ces trois instructions sont la plupart du temps utilisées avant des instructions de branchement conditionnel.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
CMP #xx	immédiat	C9	N,Z,C
CMP HHLL	étendu	CD	N,Z,C
CMP LL	page zéro	C5	N,Z,C
CMP HHLL,X	étendu indexé (X)	DD	N,Z,C
CMP HHLL,Y	étendu indexé (Y)	D9	N,Z,C
CMP LL,X	page zéro indexé (X)	D5	N,Z,C
CMP (LL,X)	pré-indexé indirect	C1	N,Z,C
CMP (LL),Y	post-indexé indirect	D1	N,Z,C
CPX #xx	immédiat	EØ	N,Z,C
CPX HHLL	étendu	EC	N,Z,C
CPX LL	page zéro	E4	N,Z,C
CPY #xx	immédiat	CØ	N,Z,C
CPY HHLL	étendu	CC	N,Z,C
CPY LL	page zéro	C4	N,Z,C

Exemple : Soit l'instruction CMP $ØØ

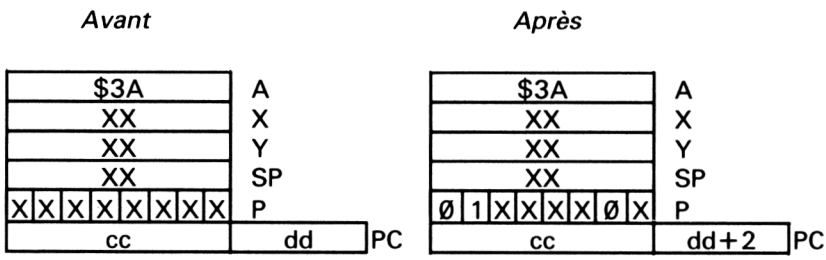
on suppose que l'adresse page-zéro $ØØ contient $12 et que l'accumulateur contient $3A.

Le 65Ø2 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, le bit Z à Ø et le bit N à Ø.



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 1ere chaîne et à partir de $2Ø pour la seconde.

Le programme est le suivant :

```
        LDY #$FF    ; INDICATEUR DE CHAINES DIFFERENTES
        LDX $ØØ     ; INITIALISE COMPTEUR DE CARACTERES
COMPT   LDA $ØØ,X   ; CHARGE CARACTERE
        CMP $2Ø,X   ; COMPARE AVEC 2eme CHAINE
        BNE FIN     ; SI DIFFERENTS, TERMINE
        INX         ; SINON, RECOMMENCE AVEC CARACTERE SUIVANT
        CPX $5Ø     ; DERNIER CARACTERE?
        BNE COMPT   ; NON, CONTINUE
        LDY #$ØØ    ; OUI, CHAINES SEMBLABLES
FIN     STY $51     ; CHARGE RESULTAT EN MEMOIRE
        BRK
```

Ce programme fonctionne de la manière suivante :

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

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

Y = $FF si les deux chaînes sont différentes

Y = $ØØ si les deux chaînes sont identiques.

5.7. LES INSTRUCTIONS DE BRANCHEMENT

IL s'agit d'une part de l'instruction de branchement inconditionnel JMP et des instructions de branchement conditionnel qui sont les suivantes :

- BEQ : Branchement si égalité à zéro
- BNE : Branchement si non-égalité à zéro
- BMI : Branchement si inférieur à zéro
- BPL : Branchement si supérieur à zéro
- BCC : Branchement si C = Ø
- BCS : Branchement si C = 1
- BVC : Branchement si V = Ø (pas de débordement)
- BVS : Branchement si V = 1 (débordement)

5.7.1. L'instruction de branchement inconditionnel JMP

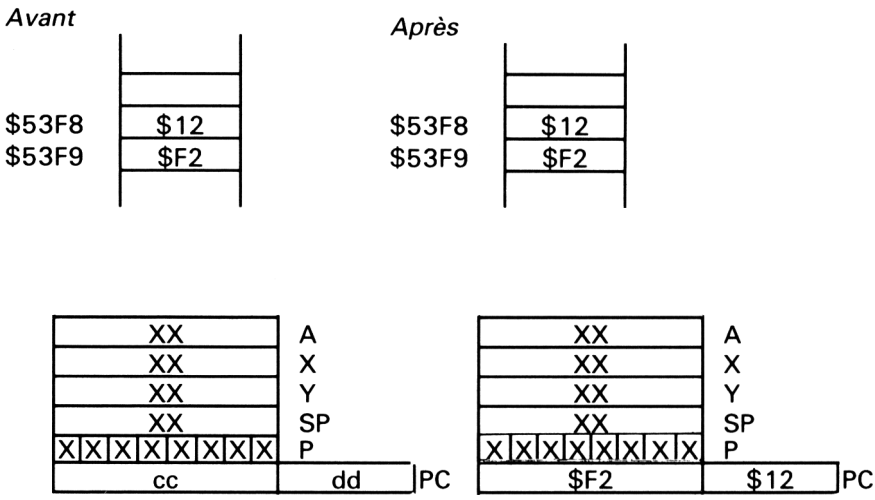
Cette instruction est l'équivalent du GOTO en Basic (du moins pour l'adressage étendu) sauf qu'en assembleur l'étiquette peut avoir un nom (exemple " BOUCLE "). Lorsque le microprocesseur rencontre le code-opération de l'instruction JMP, il charge son compteur ordinal avec l'adresse étendue ou bien l'adresse indirecte spécifiée par l'opérande.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
JMP HHLL	étendu	4C	aucun
JMP (HHLL)	indirect	6C	aucun

Nous allons illustrer cette instruction par un exemple d'adressage indirect puisqu'elle est la seule à posséder ce mode d'adressage.

Soit l'instruction suivante : JMP ($53F8)
et supposons que les contenus des adresses $53F8 et $53F9 soient respectivement $12 et $F2.

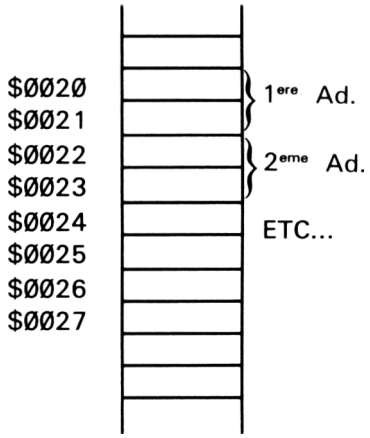
L'octet de poids faible du PC sera chargé par $12 alors que l'octet de poids fort sera chargé par $F2.



L'adressage indirect dans le cas d'un JMP peut être utile lorsque l'on veut accéder à une adresse différente suivant le résultat d'une opération. On range donc un ensemble d'adresses indirectes dans une table en mémoire et on y accède avec le petit programme suivant :

```
LDA   #$nn      ; CHARGE NUMERO DE L'ADRESSE
ASL   A         ; MULTIPLIE PAR 2
TAX
LDA   $2Ø,X     ; POINTE SUR L'OCTET DE POIDS FAIBLE
STA   $Ø1
LDA   $21,X     ; POINTE SUR L'OCTET DE POIDS FORT
STA   $Ø2
JMP   ($Ø1Ø2)   ; SAUT A L'ADRESSE INDIRECTE
```

L'implantation en mémoire des adresses indirectes est la suivante :



L'index représente le numéro de l'adresse considérée. Il est multiplié par 2 étant donné qu'une adresse est codée sur deux octets.

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'occurence ici, c'est le contenu d'un bit du registre d'état P qui importe. Toutes ces instructions ont un fonctionnement identique c'est pourquoi nous n'en expliciterons qu'une seule.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
BEQ	relatif	FØ	aucun, condition Z = 1
BNE	relatif	DØ	aucun, condition Z = Ø
BMI	relatif	3Ø	aucun, condition N = 1
BPL	relatif	1Ø	aucun, condition N = Ø
BCS	relatif	BØ	aucun, condition C = 1
BCC	relatif	9Ø	aucun, condition C = Ø
BVS	relatif	7Ø	aucun, condition V = 1
BVC	relatif	5Ø	aucun, condition V = Ø

Reprenons l'exemple de l'addition en BCD sur 4 chiffres. Le programme était le suivant :

```
        SED             ; MISE EN MODE DECIMAL
        CLC
        LDX #$ØØ       ; CHARGEMENT DU COMPTEUR DE BOUCLE
BOUCLE  LDA $ØØ,X       ; POINTE SUR LES CHIFFRES DES DIZAINES + UNITES
        ADC $Ø2,X       ; EFFECTUE L'ADDITION
        STA $Ø4,X       ; STOCKAGE EN MEMOIRE
        INX
        CMP $Ø2         ; FIN DE BOUCLE
        BNE BOUCLE      ; NON, CHARGE CHIFFRES CENTAINES + MILLIERS
        BRK             ; OUI, TERMINE
```

Au premier passage, le contenu du registre X est égal à $Ø1 donc Z = Ø. L'instruction BNE s'exécute donc correctement et il y a branchement à l'adresse BOUCLE (déplacement relatif vers l'arrière). Au 2e passage, le registre X est incrémenté et comparé à $Ø2. A la fin de ce 2e passage, le branchement à l'étiquette " BOUCLE " n'est pas effectué puisqu'il y a égalité entre X et $Ø2. Le programme s'arrête alors avec un BRK.

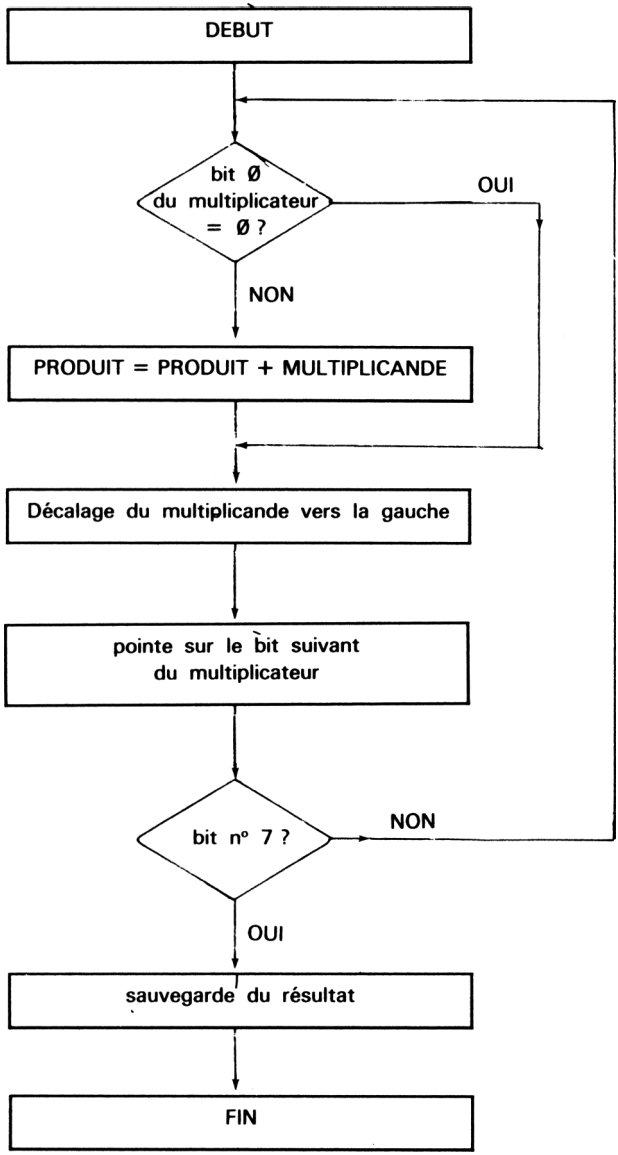
Nous allons maintenant décrire un programme de multiplication de deux nombres binaires non signés de 8 bits (le résultat sera alors donné sur 16 bits). Rappelons tout d'abord le principe d'une multiplication en décimal.

```
  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
-------
ØllØØ    (12)
```

Elle s'effectue exactement de la même manière qu'en décimal. L'organigramme d'un programme de multiplication sur 8 bits sera donc le suivant :



Cet organigramme parle de lui-même et représente bien le déroulement des calculs dans le cas d'une multiplication.

Notons qu'au lieu de décaler le multiplicande vers la gauche on aurait très bien pu décaler le résultat (PRODUIT) vers la droite.

Le listing du programme est le suivant :

```
        LDX #$Ø8     ; INITIALISE COMPTEUR DE BOUCLE
        LDA #$ØØ     ; ANNULE LE RESULTAT
        STA PRODB    ; PARTIE BASSE
        STA PRODH    ; PARTIE HAUTE
        STA VAR      ; INTERMEDIAIRE DE CALCUL
DECAL   LSR MTEUR    ; BIT Ø = Ø?
        BCC SUITE    ; OUI, DECALAGE MUTIPLICANDE
        LDA PRODB    ; NON, ADDITIONNE MULTIPLICANDE
        CLC
        ADC MCANDE ; PARTIE BASSE
        STA PRODB    ; SAUVEGARDE PARTIE BASSE DU RESULTAT
        LDA PRODH    ; PARTIE HAUTE
        ADC VAR
        STA PRODH    ; SAUVEGARDE PARTIE HAUTE
SUITE   ASL MCANDE ; DECALAGE MULTIPLICANDE A GAUCHE
        ROL VAR      ; SAUVEGARDE BIT 7 DANS VAR
        DEX          ; DERNIERE ITERATION?
        BNE DECAL    ; NON, RECOMMENCE
        BRK          ; OUI, TERMINE
```

Les adresses des variables sont les suivantes :

PRODB = partie basse du résultat

PRODH = partie haute du résultat.

VAR = intermédiaire de calcul temporaire

MTEUR = multiplicateur

MCANDE = multiplicande

Nous allons maintenant donner la valeur de ces variables et du bit C à chaque itération pour un exemple donné.

Soit à calculer le produit suivant : $12 × $5E

Initialisation :

PRODB = $ØØ
PRODH = $ØØ
VAR = $ØØ
MTEUR = $12
MCANDE = $5E
C = -
X = $Ø8

1er itération

PRODB = $ØØ
PRODH = $ØØ
VAR = $ØØ
MTEUR = $Ø9
MCANDE = $BC
C = Ø
X = $Ø7

2e itération

PRODB = $6C
PRODH = $ØØ
VAR = $Ø1
MTEUR = $Ø4
MCANDE = $78
C = 1
X = $Ø6

3e itération

PRODB = $6C
PRODH = $ØØ
VAR = $Ø2
MTEUR = $Ø2
MCANDE = $FØ
C = Ø
X = $Ø5

4e itération

PRODB = $6C
PRODH = $ØØ
VAR = $Ø5
MTEUR = $Ø1
MCANDE = $EØ
C = Ø
X = $Ø4

5e itération

PRODB = $9C
PRODH = $Ø6
VAR = $ØB
MTEUR = $ØØ
MCANDE = $CØ
C = 1
X = $Ø3

6e itération

PRODB = $9C
PRODH = $Ø6
VAR = $17
MTEUR = $ØØ
MCANDE = $8Ø
C = Ø
X = $Ø2

7e itération

PRODB = $9C
PRODH = $Ø6
VAR = $2F
MTEUR = $ØØ
MCANDE = $ØØ
C = Ø
X = $Ø1

8e itération

PRODB = $9C
PRODH = $Ø6
VAR = $5E
MTEUR = $ØØ
MCANDE = $ØØ
C = Ø
X = $ØØ

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.

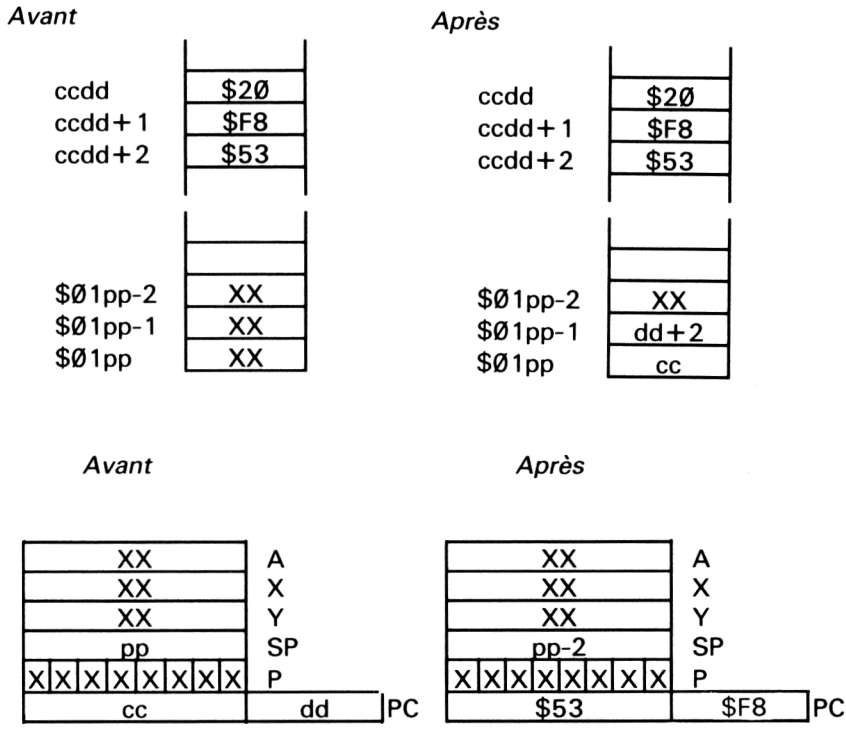
L'instruction RTS est l'équivalent du RETURN en Basic et est donc une instruction de retour de sous-programme.

Lorsque le microprocesseur rencontre l'instruction JSR, il charge le contenu du compteur ordinal dans la pile (lorsqu'il pointe sur le 3e octet de l'instruction JSR), puis se branche à l'adresse spécifiée par l'opérande.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
JSR HHLL	étendu	2Ø	aucun
RTS	implicite	6Ø	aucun

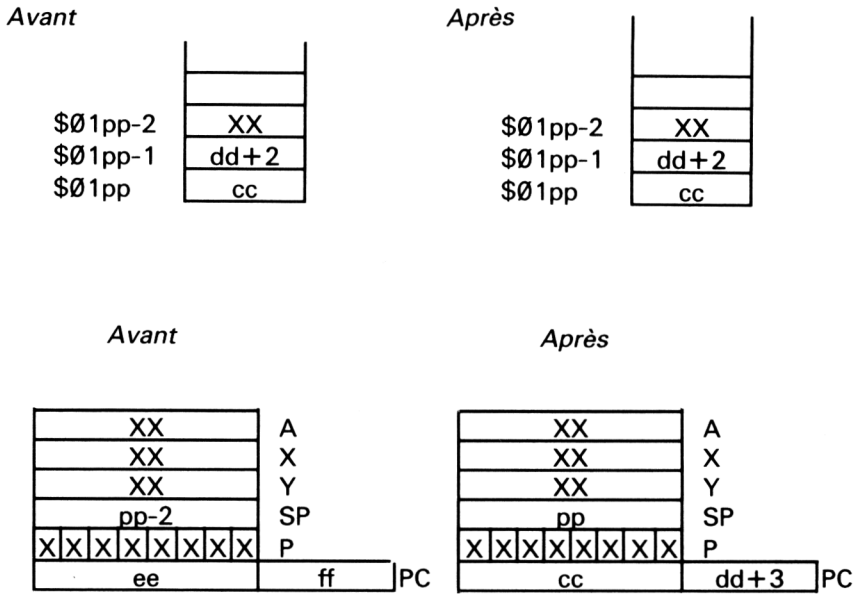
Lorsque le microprocesseur rencontre l'instruction RTS, il va chercher l'adresse qui se trouve en haut de la pile, l'incrémente et la charge ensuite dans le compteur ordinal.

Exemple : Soit l'instruction JSR $53F8



Le contenu du PC, lorsque le microprocesseur pointe sur l'instruction JSR (code-opération $2Ø), est égal à ccdd ce qui fait que la pile est chargée par ccdd+2 dans les octets d'adresse $Ø 1pp et $Ø1pp-1. Il est à noter que l'octet de poids fort du PC est chargé en premier et donc se trouve à l'adresse $Ø1pp. Le PC est alors en chargé avec la valeur $53F8 et il y a exécution du sous-programme.

A la fin de celui-ci on rencontre l'instruction RTS qui permet de revenir au programme principal.



Après avoir rencontré l'instruction RTS, le PC est restauré. Le pointeur de pile est est alors incrémenté de 2 (sans que le contenu de la pile ne soit modifié).

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 Y la longueur du bloc à scruter et dans les cases mémoire $ØØ et $Ø1 l'adresse de début de ce bloc.

```
LDA   DEBB      ;STOCKE ADRESSE DE DEBUT DU BLOC(PARTI
                 BASSE)
STA   $ØØ       ;
LDA   DEBH      ;STOCKE ADRESSE DE DEBUT DU BLOC(PARTI
                 HAUTE)
STA   $Ø1
LDY   LONG      ;LONGUEUR DU BLOC
JSR   MAX       ;RECHERCHE DU MAX
STA   $Ø2       ;SAUVEGARDE DU RESULTAT
BRK             ;FIN
```

```
MAX   INY
      LDA   #$00       ;VALEUR MAXIMALE=0
SUITE DEY              ;DERNIER ELEMENT?
      BEQ   FIN        ;OUI, RETOUR AU PROGRAMME PRINCIPAL
      CMP   ($00),Y    ;NOUVELLE VALEUR=MAXIMUM?
      BPL   SUITE      ;NON, RECOMMENCE
      LDA   ($00),Y    ;OUI, CHARGE CE NOUVEAU MAXIMUM
      JMP   SUITE      ;CONTINUE
FIN   RTS              ;SAUVEGARDE  DU RESULTAT
```

Ce programme est simple et se comprend aisément. Le sous-programme est utilisable quelles que soient l'adresse de début du bloc et sa longueur (inférieure à 256 mots à cause du registre X).

5.9. L'INSTRUCTION RTI

En pratique, vous n'aurez probablement jamais à l'utiliser (au-même titre que les instructions SEI et CLI vue précédemment).

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).

Lors d'une interruption, à supposer que le bit I du registre P soit à Ø (interruptions autorisées), le microprocesseur met à 1 celui-ci afin de ne pas être interrompu par une nouvelle interruption. Ensuite il pousse le compteur ordinal et le registre P sur la pile (octet de poids fort du PC en premier, puis octet de poids faible, puis registre d'état P).

Il y a ensuite branchement aux adresses $FFFE et $FFFF qui contiennent respectivement l'octet de poids faible et l'octet de poids fort de l'adresse de la routine de traitement des interruptions (interruption IRQ).

REMARQUE : il existe également une interruption non-masquable (NMI) qui donc a lieu quel que soit l'état du bit I. Les adresses de branchement sont alors $FFFA et $FFFB.

A la fin de la routine de traitement de l'interruption on trouve l'instruction RTI qui entraîne la restauration du PC et du registre P. Notons que contrairement au cas de l'instruction RTS, le PC n'est pas incrémenté avant d'être chargé.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
RTI	implicite	4Ø	registre d'état restauré

Nous n'entrerons pas plus dans les détails concernant les interruptions car, comme nous l'avons déjà dit, il y a très peu de chances que vous ayiez à vous en servir un jour.

5.10 LES INSTRUCTIONS SUR LA PILE

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

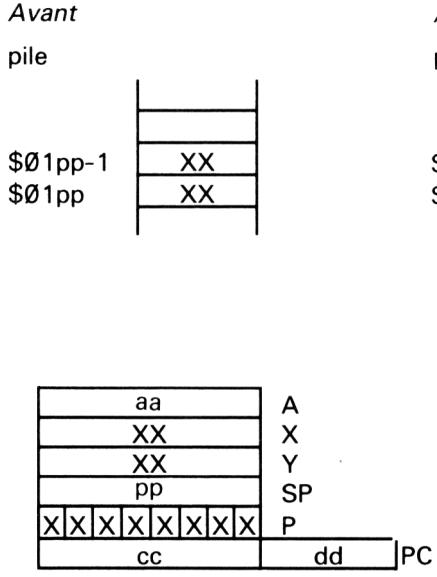
- empilement PHA, PHP
- dépilement PLA, PLP

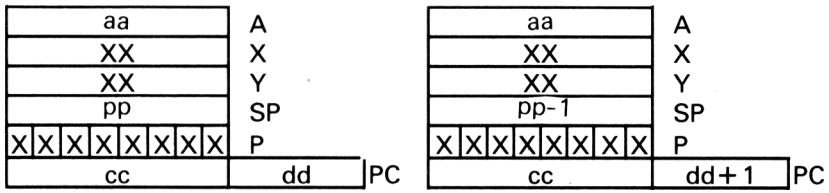
Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
PHA	implicite	48	aucun
PHP	implicite	Ø8	aucun
PLA	implicite	68	N,Z
PLP	implicite	28	registre d'état restauré

1) Fonctionnement des instructions d'empilement

Selon le cas, il y a stockage du registre concerné (A ou P) au sommet de la pile. Le pointeur de pile est ensuite décrémenté de 1.

Exemple : Soit l'instruction PHA





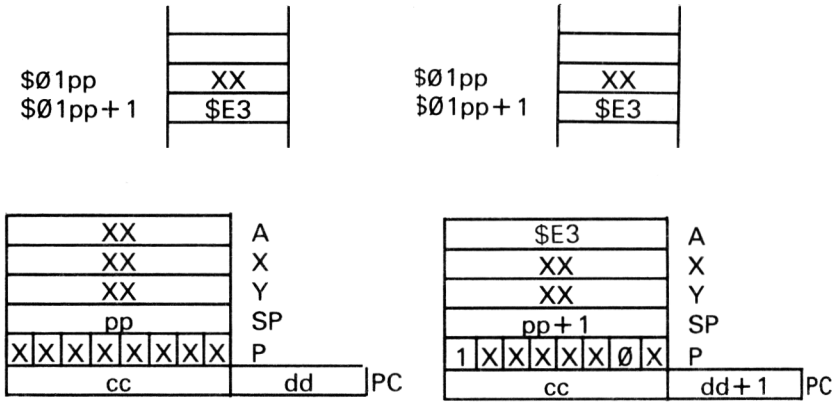
2) Fonctionnement des instructions de dépilement

Ces instructions incrémentent le pointeur de pile et ensuite chargent l'accumulateur ou le registre d'état (selon le cas) avec l'octet situé en haut de la pile.

Exemple : Soit l'instruction PLA

on suppose que le sommet de la pile (après incrémentation) contient $E3.

L'état des registres est le suivant :



Comme $E3 = 11100011, le bit N est positionné à 1 et Z à Ø.

Il est souvent nécessaire, lors d'un appel de sous-programme, de " sauvegarder le contexte ", c'est-à-dire de stocker l'état des registres quelque part en mémoire avant son exécution.

Cette opération doit souvent pouvoir se faire rapidement de même que la récupération des données lors du retour au programme principal. Les deux petits morceaux de programme qui suivent indiquent la démarche à suivre.

Sauvegarde du contexte :

```
PHA    ; SAUVEGARDE ACCUMULATEUR
PHP    ; SAUVEGARDE REGISTRE P
TXA    ; SAUVEGARDE REGISTRE X
PHA
TYA    ; SAUVEGARDE REGISTRE Y
PHA
```

Restauration du contexte :

```
PLA    ; RESTAURATION REGISTRE Y
TAY
PLA    ; RESTAURATION REGISTRE X
TAX
PLP    ; RESTAURATION REGISTRE P
PLA    ; RESTAURATION ACCUMULATEUR
```

Etant donné qu'il n'y a pas d'instruction d'empilement ou de dépilement direct pour les registres X et Y, il est nécessaire de transférer tout d'abord le contenu de ces registres dans l'accumulateur.

Pour sauvegarder le pointeur de pile SP on écrirait :

```
TSX
TXA
PHA
```

et inversement pour le restaurer :

```
PLA
TAX
TXS
```

5.11. LES INSTRUCTIONS SPECIALES

Il s'agit des instructions NOP et BRK.

5.11.1. L'instruction NOP

Le fonctionnement de cette instruction est très facile à comprendre puisqu'elle ne fait ... rien ou presque : elle se contente juste d'incrémenter le compteur ordinal. Mais quelle est l'utilité d'une instruction qui ne fait rien ? En fait elle peut servir à beaucoup de choses et ceux d'entre vous qui ont utilisé des calculatrices programmables doivent le savoir :

– remplacer une instruction non utile par un NOP permet de ne pas avoir à réécrire tout le programme lors d'un assemblage à la main ou pendant la mise au point. Sans cela, il faudrait recalculer tous les branchements.

– provoquer un délai 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 généralement être débarrassé de ces instructions inutiles afin d'en diminuer le temps d'exécution.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
NOP	implicite	EA	aucun

5.11.2. L'instruction BRK

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 s'agissait d'interruptions matérielles.

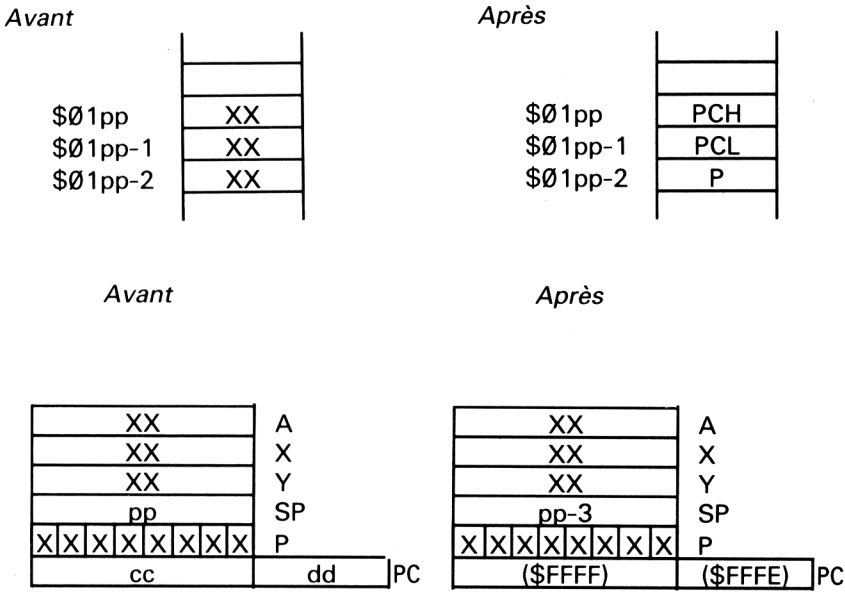
Il est possible de réaliser la même chose grâce à une instruction que l'on appelle interruption logicielle : BRK. Lorsqu'un programme est exécuté et que le microprocesseur rencontre cette instruction, celui-ci se branche aux adresses $FFFE et $FFFF où il trouve l'adresse de début d'un sous-programme.

Selon le contenu de ces deux cases mémoire, il y aura retour au programme moniteur du système, au Basic, ou à tout autre programme. L'instruction BRK est très utilisée pour mettre au point des programmes car elle permet d'en arrêter le déroulement là où on le désire.

Instruction	Mode d'adressage	Code-opération	Indicateurs affectés
BRK	implicite	ØØ	B,I

Lors de l'exécution d'un Break (BRK), le PC est incrémenté de 2 et le bit B du registre P mis à 1. Ces deux registres sont ensuite poussés au sommet de la pile (PCH en premier, puis PCB, puis registre P). Après cela le masque d'interruptions est positionné (bit I = 1) afin d'interdire les interruptions matérielles.

L'état des registres est le suivant :



Après l'exécution du BRK, le PC est donc chargé avec ($FFFE) et ($FFFF).

Nous venons de voir le fonctionnement de toutes les instructions du 65Ø2. Vous devriez donc pouvoir dès maintenant commencer à écrire vos propres programmes en assembleur.

Dans le prochain chapitre nous aborderons le problème des Entrées-Sorties qui sont en fait les moyens de communiquer avec l'extérieur.

Mais si vous voulez avant cela commencer à programmer, nous vous suggérons de lire le chapitre VII qui traite de l'écriture et de la mise au point d'un programme en assembleur.

Vous pouvez trouvez en annexe un récapitulatif de toutes les instructions du 65Ø2.

6
Les entrées-sorties

6.1. GÉNÉRALITÉS

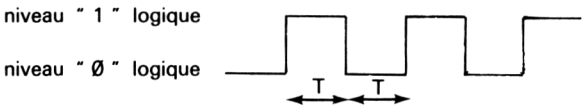
Nous avons vu jusqu'à présent des instructions qui permettaient d'effectuer des transferts entre Mémoire et Registres et des opérations sur ceux-ci. Le problème est qu'il est nécessaire également de communiquer avec l'extérieur : c'est ce que nous appellerons les Entrées-Sorties. Les entrées permettent 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 65Ø2, 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.

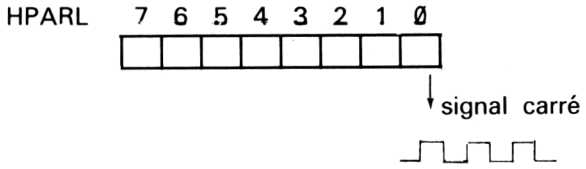
Plutôt que de faire des discours abstraits nous allons examiner quelques exemples.

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 :

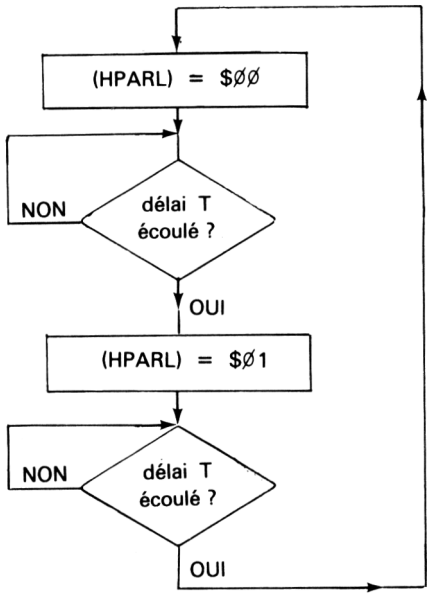


Ce signal pourra être présent par exemple sur le bit Ø de l'octet présent à l'adresse HPARL



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.

1ere méthode : on utilise un sous-programme pour générer le délai de durée T.

```
DEBUT  LDA  #$ØØ    ; SORTIE HAUT PARLEUR = Ø
       STA  HPARL
       JSR  DELAI   ; RESTE A CET ETAT PENDANT T
       LDA  #$Ø1    ; SORTIE HAUT PARLEUR = 1
       STA  HPARL
       JSR  DELAI   ; RESTE A CET ETAT PENDANT T
       JMP  DEBUT   ; RECOMMENCE
DELAI  LDX  #$C8    ; INITIALISATION DE T
COMPT  DEX          ; FINI?
       BNE  COMPT   ; NON, CONTINUE
       RTS          ; OUI, CHANGE L'ETAT DU HAUT PARLEUR
```

Ce programme est très simple et n'amène pratiquement aucun commentaire. La valeur $C8 chargée dans X permet d'obtenir une durée T d'environ 1ms ce qui donne une fréquence d'environ 5ØØ Hz. (Ceci est le cas d'un 65Ø2 avec fréquence d'horloge de 1MHz : dans le cas d'un 65Ø2 A tournant à 2MHz on pourra remplacer $C8 par $64).

2e méthode. Ici nous n'utilisons pas de sous-programme.

On se base sur la propriété suivante $X \veebar 1 = \bar{X}$

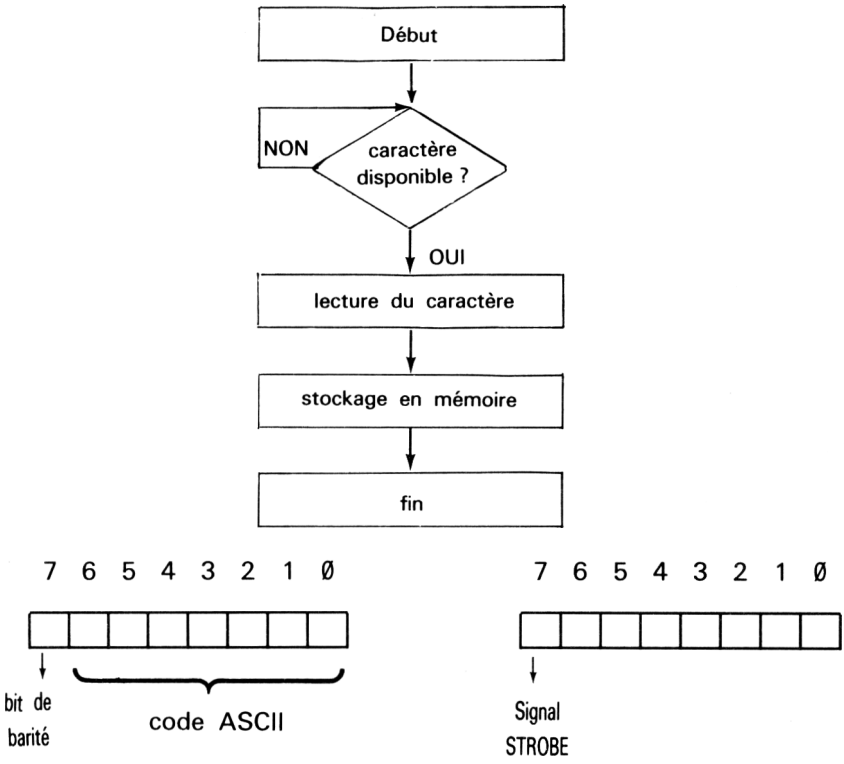
Le programme est le suivant :

```
       LDA  #$ØØ          ; MET LE HAUT PARLEUR A ZERO
CHANGE EOR  #%ØØØØØØØ1    ; CHANGE CET ETAT
       STA  HPARL          ; TRANSFERT VERS LE HAUT PARLEUR
       LDX  #$C8           ; INITIALISATION DE T
COMPT  DEX                 ; FINI?
       BNE  COMPT          ; NON, CONTINU
       JMP  CHANGE         ; OUI, CHANGE A NOUVEAU L'ETAT DU
                           HAUT PARLEUR
```

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

Beaucoup de claviers utilisés dans les microordinateurs 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   BIT   STROBE          ; CARACTERE DISPONIBLE?
        BMI   SCRUT           ; NON, RECOMMENCE
        LDA   CARACT          ; OUI, CHARGEMENT DANS A
        AND   #%Ø1111111      ; MASQUE LE BIT DE PARITE
        STA   RANGE           ; RANGEMENT EN MEMOIRE
        BRK
```

Vous pouvez remarquer que nous avons utilisé ici l'instruction BIT. 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 BIT transfère notamment le bit 7 de la case-mémoire STROBE dans le bit N du registre d'état P.

L'instruction BMI permet un branchement à l'adresse SCRUT si N = 1 donc si STROBE = 1. Sinon le caractère est rangé en mémoire après avoir été débarrassé 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 microordinateur 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 Entrée - Sorties (PIA : Peripheral Interface Adaptor, VIA : Versatile Interface Adaptor) qui se connectent facilement au 65Ø2 et qui permettent de remplir des fonctions complexes telles que Timers, générateurs d'impulsions, ports d'entrée-sorties programmables, etc... Certains microordinateurs de conception récente en sont équipés. Ceci dit leur utilisation peut paraître un petit peu difficile à comprendre à un novice.

6.2. LE 651Ø

Le 651Ø est un processeur récemment introduit sur le marché et qui fait partie de la famille 65Ø2. A ce titre, il possède un jeu d'instructions en tous points similaire et la seule différence visible pour le programmeur se place au niveau des entrée-sorties. Ce processeur est utilisé actuellement dans le COMMODORE 64.

Le 651Ø possède ce qu'on appelle un port d'entrée-sorties intégré qui a l'avantage d'être situé en page-zéro pour des raisons de rapidité.

Tout d'abord il nous faut définir ce qu'on entend par port d'entrée-sorties : il s'agit d'une case mémoire particulière dont le rôle est d'autoriser les communications entre le processeur et le monde extérieur. Une information venant d'un circuit externe peut être chargée dans cette case mémoire et ensuite être transférée dans l'accumulateur par exemple. Inversement une valeur contenue dans l'accumulateur (ou tout autre registre) et destinée à commander un périphérique peut être envoyée dans cette même case mémoire.

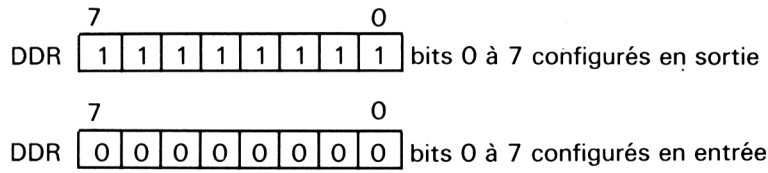
Le port d'entrée-sorties du 651Ø est situé à l'adresse page zéro $Ø1.

De plus il est nécessaire de renseigner le circuit de commande de ce port si l'on désire effectuer une entrée ou une sortie. Ce rôle est confié à un registre spécialisé appelé DDR (Data Direction Register). Comme le port lui-même, ce registre, situé à l'adresse $ØØ, possède 8 bits.

La mise à 1 de certains bits de ce registre configure en sortie le bit correspondant du port.

La mise a Ø de certains bits de ce registre configure en entrée le bit correspondant du port.

Exemple :



Nous allons maintenant examiner la façon d'aller écrire ou lire une donnée dans le port d'entrée-sorties.

Supposons que nous voulions aller lire le port, il suffira d'écrire le petit programme suivant :

```
LDA   #$00          ;PORT CONFIGURE EN ENTREE
STA   $00
LDA   $01          ;TRANSFERE LE CONTENU DU PORT DANS A
```

Si nous voulons maintenant aller écrire une valeur dans le port, il suffira d'écrire le programme suivant :

```
LDA   #$FF          ;CONFIGURE LE PORT EN SORTIE
STA   $00
LDA   VALEUR       ;VALEUR A ENVOYER SUR LE PORT
STA   $01
```

Un tel port d'entrée-sorties peut ouvrir la voie à de nouvelles applications, par exemple acquisition de données à grande vitesse, contrôles divers, etc...

Pour de plus amples détails concernant ce port et surtout sur la manière dont il est utilisé, nous vous conseillons de vous reporter au manuel de référence de votre micro-ordinateur.

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 microordinateur. Mais le problème est qu'il 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 minimum 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 Entrée-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 " Ecriture 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. Evitez au maximum les " astuces géniales " qui peuvent ne pas être comprises par d'autres et que vous-mêmes ne comprendrez peut-être plus dans quelques mois lorsque vous voudrez apporter une amélioration à votre programme.

Dans la plupart des cas le microprocesseur (65Ø2) 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.

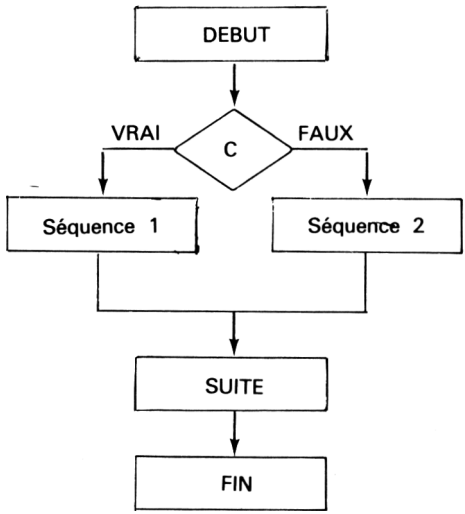
Exemple : Deux nombres sont rangés en mémoire (entrée), une séquence d'instructions en calcule le produit (sortie).

Dans chaque séquence essayez au maximum d'adopter une démarche logique. Rappelez-vous :
IF....THEN....ELSE, en BASIC, ou bien
DO....UNTIL, DO....WHILE, dans des FORTRANS évolués.

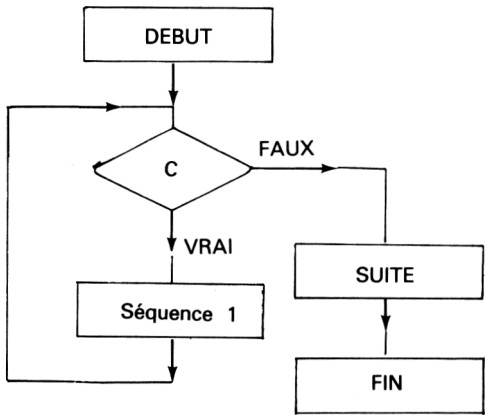
Nous vous donnons à titre indicatif les organigrammes relatifs à ces 3 instructions.

a) *IF..... THEN..... ELSE*

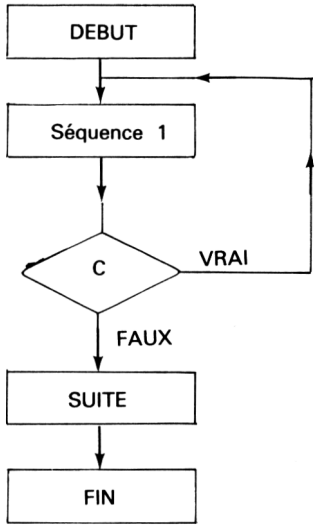
Traduisons ceci en bon français : Si la condition C est réalisée alors on exécute la séquence 1 sinon on exécute la séquence 2.



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



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



Pour des exemples d'organigrammes vous pourrez vous reporter au chapitre consacré au jeu d'instructions (en particulier programme de multiplication en binaire).

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'instruction.

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'arrêt*

Nous avons vu que l'instruction BRK était une interruption logicielle et permettait 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 65Ø2 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 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 TAY transfère le contenu de l'accumulateur dans le registre Y et non pas l'inverse.

– 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 65Ø2.

– Attention aux changements éventuels des bits du registre d'état P 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.

Economie 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 sur 1 ou 2 octets et notamment l'adressage page-zéro (surtout pour les données qui sont fréquemment utilisées).

– Utilisez la pile 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.

Economie du 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 sur 1 ou 2 octets.

– Essayez de sortir certaines instructions des boucles lorsqu'elle 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.

– 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, pré-indexé indirect, post-indexé indirect, indexés.

– Utilisez des instructions qui opèrent directement sur les registres et la page-zéro.

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 65Ø2 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 microordinateur (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.

8

Les instructions mystérieuses du 6502

Nous avons dit précédemment que le 65Ø2 possédait 56 instructions différentes qui, combinées avec les différents modes d'adressage, portaient ce nombre à 152. Or, avec un code-opération de un octet il est possible de coder 256 instructions (codes $ØØ à $FF) différentes. On peut donc se demander si les 256 - 152 = 1Ø4 codes restant n'ont pas une fonction au même titre que les autres.

La réponse est la suivante : en général non, mais il existe quelques exceptions. Ceci dit il nous faut préciser un point : ces instructions n'étant pas officielles et donc non garanties par le constructeur du 65Ø2 il se peut très bien qu'elles ne fonctionnent pas dans tous les cas ou bien que le résultat qu'elles donnent dépende du fabricant du microprocesseur ou de l'année où celui-ci a été fabriqué.

Ceci dit nous allons vous donner quelques exemples de " codes invalides " que nous avons testés et qui semblent fonctionner dans tous les cas. Examinons tout d'abord le tableau donnant les code-opérations du 65Ø2 (les vrais cette fois-ci) dans l'ordre croissant :

ØØ BRK	1Ø BPL	2Ø JSR HHLL	3Ø BMI
Ø1 ORA (LL,X)	11 ORA (LL),Y	21 AND (LL,X)	31 AND (LL),Y
Ø2	12	22	32
Ø3	13	23	33
Ø4	14	24 BIT LL	34
Ø5 ORA LL	15 ORA LL,X	25 AND LL	35 AND LL,X
Ø6 ASL LL	16 ASL LL,X	26 ROL LL	36 ROL LL,X
Ø7	17	27	37
Ø8 PHP	18 CLC	28 PLP	38 SEC
Ø9 ORA xx	19 ORA HHLL,X	29 AND xx	39 AND HHLL,X
ØA ASL A	1A	2A ROL A	3A
ØB	1B	2B	3B
ØC	1C	2C BIT HHLL	3C
OD ORA HHLL	1D ORA HHLL,X	2D AND HHLL	3D AND HHLL,X
ØE ASL HHLL	1E ASL HHLL,X	2E ROL HHLL	3E ROD HHLL,X
ØF	1F	2F	3F
4Ø RTI	5Ø BVC	6Ø RTS	7Ø BVS
41 EOR (LL,X)	51 EOR (LL),Y	61 ADC (LL,X)	71 ADC (LL),Y
42	52	62	72
43	53	63	73
44	54	64	74
45 EOR LL	55 EOR LL,X	65 ADC LL	75 ADC LL,X
46 LSR LL	56 LSR LL,X	66 ROR LL	76
47	57	67	77
48 PHA	58 CLI	68 PLA	78 SEI
49 EOR xx	59 EOR HHLL,Y	69 ADC xx	79 ADC HHLL,Y
4A LSR A	5A	6A ROR A	7A
4B	5B	6B	7B
4C JMP HHLL	5C	6C JMP (HHLL)	7C
4D EOR HHLL	5D EOR HHLL,X	6D ADC HHLL	7D ADC HHLL,X
4E LSR HHLL	5E LSR HHLL,X	6E ROR HHLL	7E
4F	5F	6F	7F
8Ø	9Ø BCC	AØ LDY xx	BØ BCS
81 STA (LL,X)	91 STA (LL),Y	A1 LDA (LL,X)	B1 LDA (LL),Y
82	92	A2 LDX xx	B2
83	93	A3	B3
84 STY LL	94 STY LL,X	A4 LDY LL	B4 LDY LL,X
85 STA LL	95 STA LL,X	A5 LDA LL	B5 LDA LL,X
86 STX LL	96 STX LL,Y	A6 LDX LL	B6 LDX LL,Y
87	97	A7	B7
88 DEY	98 TYA	A8 TAY	B8 CLV
89	99 STA HHLL,Y	A9 LDA xx	B9 LDA HHLL,Y
8A TXA	9A TXS	AA TAX	BA TSX
8B	9B	AB	BB
8C STY HHLL	9C	AC LDY HHLL	BC LDY HHLL,X
8D STA HHLL	9D STA HHLL,X	AD LDA HHLL	BD LDA HHLL,X
8E STX HHLL	9E	AE LDX HHLL	BE LDX HHLL,Y
8F	9F	AF	BF

CØ CPY xx	DØ BNE	EØ CPX xx	FØ BEQ
C1 CMP (LL,X)	D1 CMP (LL),Y	E1 SBC (LL,X)	F1 SBC (LL),Y
C2	D2	E2	F2
C3	D3	E3	F3
C4 CPY LL	D4	E4 CPX LL	F4
C5 CMP LL	D5 CMP LL,X	E5 SBC LL	F5 SBC LL,X
C6 DEC LL	D6 DEC LL,X	E6 INC LL	F6 INC LL,X
C7	D7	E7	F7
C8 INY	D8 CLD	E8 INX	F8 SED
C9 CMP xx	D9 CMP HHLL,Y	E9 SBC xx	F9 SBC HHLL,Y
CA DEX	DA	EA NOP	FA
CB	DB	EB	FB
CC CPY HHLL	DC	EC CPX HHLL	FC
CD CMP HHLL	DD CMP HHLL,X	ED SBC HHLL	FD SBC HHLL,X
CE DEC HHLL	DE DEC HHLL,X	EE INC HHLL	FE INC HHLL,X
CF	DF	EF	FF

Les colonnes dont les code-opérations se terminent par " 3 ", " 7 ", " B " et " F " sont totalement vides. C'est parmi elles que nous allons trouver nos mystérieuses instructions.

Celles-ci n'étant pas officielles, elles ne seront répertoriées que par leur code-opération puisqu'il n'existe pas de mnémonique dans leur cas. D'autre part nous avons volontairement limité notre liste aux instructions les plus intéressantes.

1) L'instruction " Ø3 "

Elle opère en page-zéro : on écrira donc Ø3 LL, LL étant l'adresse en page-zéro.

L'opération effectuée est la suivante : ASL (LL,X) + ORA (LL,X)

Donc il y a tout d'abord un décalage arithmétique vers la gauche puis un " OU " logique, le mode d'adressage pour ces deux instructions étant l'adressage pré-indexé indirect.

Exemple : si X = $ØØ
LL = $ØØ

l'opérande désigne une adresse dont les parties basses et hautes sont rangées respectivement aux adresses $ØØØØ et $ØØØ1

Supposons que ($ØØØØ) = $2Ø
et ($ØØØ1) = $A3

L'adresse désignée par l'opérande est $A32Ø ; supposons qu'elle contienne la valeur. $A9

on a

$A9 = 1Ø1Ø1ØØ1

après un décalage cela nous donne ØlØlØØlØ = $52
Supposons que l'accumulateur contienne $FØ, le résultat final sera

ØlØlØØlØ
V 11110000

= 11110010 = $F2 qui est stocké dans l'accumulateur

2) L'instruction " A3 "

Celle-ci opère également en page-zéro.

L'opération effectuée est : LDA (LL,X) + LDX (LL,X)

Il y a donc chargement de l'accumulateur et du registre X par le contenu de l'adresse spécifiée par l'opérande. Le mode d'adressage est ici aussi pré-indexé indirect.

Exemple : si X = $ØØ
LL = $ØØ
($ØØØØ) = $2Ø
($ØØØ1) = $A3

et que ($A32Ø) = $A5 on aura, après l'exécution de l'instruction A3 ØØ : A = $A5
X = $A5

3) L'instruction " B3 "

Cette instruction est la même que la précédente sauf au niveau du mode d'adressage qui est ici post-indexé indirect.

Donc cette instruction effectue

LDA (LL),Y + LDX (LL),Y

Donc l'instruction B3 ØØ avec

Y = $1Ø
($ØØØØ) = $ØØ
($ØØØ1) = $Ø3
($Ø31Ø) = $A5

donnera X = $A5
A = $A5

4) L'instruction " 87 "

Celle-ci opère en page-zéro. Elle effectue le " ET " logique entre le contenu de l'accumulateur et le contenu du registre X. Le résultat est stocké à l'adresse page-zéro indiquée.

Exemple : si A = $A5
X = $FØ

L'instruction 87 ØØ donnera ($ØØØØ) = $AØ

REMARQUE : Les bits N et Z du registre d'état P ne sont pas positionnés normalement comme dans le cas d'un " ET " classique.

5) L'instruction " A7 "

Elle opère en page-zéro. L'opération effectuée est

(LL) ⟶ A et (LL) ⟶ X

Le contenu de l'adresse page-zéro spécifiée par l'opérande est chargé à la fois dans l'accumulateur et dans le registre X.

Exemple : si ($ØØØØ) = $3Ø

on aura, après l'exécution de l'instruction A7 ØØ : A = $3Ø
X = $3Ø

REMARQUE : Les bits N et Z sont positionnés normalement comme dans toute opération de chargement classique.

si ($ØØØØ) = $3Ø on a N = Ø, Z = Ø
si ($ØØØØ) = $ØØ on a N = Ø, Z = 1
si ($ØØØØ) = $EF on a N = 1, Z = Ø

6) L'instruction " C7 "

C'est peut-être l'instruction non officielle la plus intéressante. Elle permet d'effectuer des boucles extrêmement simplement.

Elle opère également en page-zéro et effectue l'opération suivante :

" Le contenu de l'adresse page-zéro désignée par l'opérande est décrémenté puis comparé à l'accumulateur A ".

Exemple : Soit l'instruction C7 Ø5 par exemple (LL = $Ø5).

Supposons que ($ØØØ5) = $A3

et que A = $BØ

Le contenu de l'adresse $ØØØ5 est tout d'abord décrémenté → $A2. Il est ensuite comparé à $BØ. Il y a donc positionnement des bits N,Z,C comme dans l'instruction CMP classique.

Ici après l'exécution de cette instruction, on obtient :

N = Ø (résultat > Ø)
Z = Ø (A # mémoire)
C = 1

Si on avait eu A = $BØ et ($ØØØ5) = $B1

le résultat aurait été : N = Ø
Z = 1
C = 1

Exemple de programme donnant un délai

Nous avons vu dans le chapitre sur les Entrées-Sorties qu'il était possible grâce à une boucle d'obtenir un délai de durée T.

Nous vous proposons ici une nouvelle méthode ; le programme s'écrit :

```
A9 C8          LDA #$C8
85 ØØ          STA $ØØ
A9 ØØ          LDA #$ØØ
C7 ØØ    LOOP
DØ FC          BNE LOOP
```

7) L'instruction " E7 "

Cette instruction opère en page-zéro et effectue l'opération suivante : INC LL + SBC LL

Il y a donc incrémentation de l'adresse page-zéro désignée par l'opérande puis soustraction de ce résultat au contenu de l'accumulateur (selon les règles de l'instruction SBC).

Exemple : Soit l'instruction E7 ØØ

et supposons ($ØØØØ) = $33
A = $B5

Si la retenue est mise à 1 initialement par une instruction SEC on obtient : $ØØØØ = $34 tout d'abord

puis A = $81
N = 1
V = Ø
Z = Ø
C = 1

En effet

$B5 = 1Ø11Ø1Ø1
$34 = ØØ11Ø1ØØ
$-34 = 11ØØ11ØØ

$B5 - $34 - $\overline{c}$ = 11ØØØØØØ1

8) L'instruction " F7 "

Elle opère en page-zéro et effectue l'opération suivante :

INC LL,X + SBC LL,X

Il y a, comme dans le cas de l'instruction " E7 ", incrémentation du contenu de l'adresse page-zéro puis soustraction de ce résultat au contenu de l'accumulateur. Le mode d'adressage est ici indexé par X.

Exemple : Soit l'instruction F7 ØØ

avec X = $1Ø
($ØØ1Ø) = $33
A = $B5

et si C = 1 initialement on obtient

$ØØ1Ø = $34 tout d'abord

puis A = $81
N = 1
V = Ø
Z = Ø
C = 1 comme dans le cas précédent.

9) L'instruction " 2B "

Cette instruction charge tout simplement l'accumulateur avec la valeur $ØØ.

Donc 2B peut remplacer A9 ØØ et on remarque que le bit Z est mis à 1 comme dans le cas de l'instruction A9.

Nous venons de voir quelques " codes invalides " du 65Ø2. Le problème est qu'il n'existe pas pour les instructions qu'ils représentent de mnémoniques standardisés. Pour les utiliser à l'aide d'un assembleur il faudra donc les charger en mémoire à l'aide des directives • BYTE;• DBYTE comme nous l'avons vu dans le cas de l'instruction " C7 ". Il existe probablement d'autres instructions mystérieuses dans le 65Ø2. A vous de les trouver si le cœur vous en dit !.

Cet ouvrage est maintenant terminé en même temps que notre travail. Le vôtre commence maintenant (s'il n'a pas déjà commencé). Le seul moyen d'apprendre est de pratiquer soi-même et c'est ce que nous vous engageons vivement à faire.

Nous donnons en annexe un tableau (un autre !) contenant le jeu d'instructions par ordre alphabétique (le vrai cette fois-ci).

TABLEAU RECAPITULATIF DES INSTRUCTIONS DU 6502

mode d'adressage mnémonique	IMP	IMM	ACC	ET	PZ	EIX	EIY	PZX	PZY	REL	IND	PRE	POST
ADC		69		6D	65	7D	79	75				61	71
AND		29		2D	25	3D	39	35				21	31
ASL			ØA	ØE	Ø6	1E		16					
BCC										9Ø			
BCS										BØ			
BEQ										FØ			
BIT				2C	24								
BMI										3Ø			
BNE										DØ			
BPL										1Ø			
BRK	ØØ												
BVC										5Ø			
BVS										7Ø			
CLC	18												
CLD	D8												
CLI	58												
CLV	B8												
CMP		C9		CD	C5	DD	D9	D5				C1	D1
CPX		EØ		EC	E4								
CPY		CØ		CC	C4								
DEC				CE	C6	DE		D6					
DEX	CA												
DEY	88												
EOR		49		4D	45	5D	59	55				41	51
INC				EE	E6	FE		F6					
INX	E8												
INY	C8												

TABLEAU RECAPITULATIF (SUITE)

Mode d'adressage mnémonique	IMP	IMM	ACC	ET	PZ	EIX	EIY	PZX	PZY	REL	IND	PRE	POST
JMP				4C							6C		
JSR				20									
LDA		A9		AD	A5	BD	B9	B5				A1	B1
LDX		A2		AE	A6		BE		B6				
LDY		A0		AC	A4	BC		B4					
LSR			4A	4E	46	5E		56					
NOP	EA												
ORA		09		0D	05	1D	19	15				01	11
PHA	48												
PHP	08												
PLA	68												
PLP	28												
ROL			2A	2E	26	3E		36					
ROR			6A	6E	66	7E		76					
RTI	40												
RTS	60												
SBC		E9		ED	E5	FD	F9	F5				E1	F1
SEC	38												
SED	F8												
SEI	78												
STA				8D	85	9D	99	95				81	91
STX				8E	86				96				
STY				8C	84			94					
TAX	AA												
TAY	A8												
TYA	98												
TSX	BA												
TXA	8A												
TXS	9A												

Abréviations

IMP = adressage implicite
IMM = adressage immédiat
ACC = adressage accumulateur
ET = adressage étendu
PZ = adressage page-zéro
EIX = adressage étendu indexé par X
EIY = adressage étendu indexé par Y
PZX = adressage page-zéro indexé par X
PZY = adressage page-zéro indexé par Y
REL = adressage relatif
IND = adressage indirect
PRE = adressage pré-indexé indirect (par X)
POST = adressage post-index-indirect (par Y).

Imprimé en France. — JOUVE, 18, rue St-Denis, 75001 PARIS
N° 12563. Dépôt légal : Mars 1984
N° d'Editeur : 4114

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 65Ø2, l'un des microprocesseurs les plus utilisés actuellement (sur ORIC, ATMOS, VIC-20, CQMMODORE-64, PET, CBM, ATOM, ATARI 400/800, ATARI SÉRIE XL, APPLE II, APPLE IIE, APPLE III, MPF II, BASIS 108, tous les compatibles APPLE, 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

8604

EYROLLES **L'ASSEMBLEUR FACILE DU 6502 ET DU 6510** **F. MONTEIL**