Crackers Velus edition

INTRODUCTION

Ce livre fait partie des séries Dr Watson Langage assembleur. Il a été écrit pour fournir une méthode de programmation en langage assembleur Z80, entièrement à votre rythme. Tout au long de la méthode, de nouvelles instructions sont illustrées par des programmes exemples. De plus, de nombreux exercices ont été inclus afin de vérifier la compréhension des concepts expliqués.

Après avoir lu et travaillé les deux premiers chapitres, le lecteur trouvera la lecture de l'annexe 6 fort utile. Cette annexe résume l'utilisation de l'assembleur, et explique également commet sauvegarder et charger des programmes sur cassettes.

Tout en travaillant sur ce livre, n'hésitez pas à expérimenter vos idées, vous n'abimerez absolument pas l'ordinateur ou l'assembleur.

Trêves de discours: vous avez désormais 10 chapitres remplis d'applications pratiques, sans parler des annexes, pour travailler. J'espère que vous y prendrez plaisir.

T. Hebertson
Londres
Février 1985

TABLE DES MATIERES

CHAPITRE 1

POUR COMMENCER

Ayant déjà utilisé un ordinateur auparavant, vous savez probablement que des choses appelées 'code machine' et 'langage machine' existent. Code machine, c'est simplement le langage que le microprocesseur de votre ordinateur, un Z80, comprend. Comme exemple prenez une addition toute simple - ajouter un à 83.

En français vous diriez :

Ajoutez un à 83 ; quelle est la réponse ?

En Basic vous pourriez dire quelque chose comme :

```
10 LET A=83
20 LET A=A+1
30 PRINT A
```

En code machine Z80 vous pourriez dire:

```
3E 53
C6 01
CD 5A BB
C9
```

C'est parfaitement inintelligible, n'est-ce pas ? hé bien c'est pour cela que nous utilisons le langage assembleur. Le même problème est donné ci-dessous en langage assembleur avec un bref commentaire pour chaque ligne.

```
LD A,83     Charge 83 dans l'Accumulateur 'A'
ADD A,1     Ajoute 1 à l'Accumulateur
CALL 47962  Mémorise le contenu de A sur l'écran
RET         Retour du sous-programme code machine
```

C'est beaucoup plus facile à lire que le code machine, n'est-ce pas? Avec un assembleur vous pouvez entrer votre programme en langage assembleur et vous pouvez le lire et le comprendre plus facilement. tout ce que fait l'assembleur, c'est de convertir le langage assembleur en code machine. Ainsi, lorsqu'il voit 'LD A' il convertit cette commande en code machine '3E' et place cela en mémoire au bon endroit.

Le microprocesseur Z80 contient divers registres, qui sont en réalité des cases mémoire internes, où les données peuvent être stockées. La plupart des instructions du Z80 utilisent ces registres. Un des registres les plus souvent utilisés est l'accumulateur. L'accumulateur est capable de mémoriser des nombres de 0 à 255. Observons maintenant une instruction qui nous permet de charger (LoaD) une valeur dans l'accumulateur.

L'instruction pour cela est :

```
LD A,n            Charger (LoaD) dans l'accumulateur la valeur immédiate n
```

Vous remarquerez que les lettres choisies pour l'instruction reflètent (en anglais du moins) ce qu'elle accomplit. Les instructions de ce type sont appelées 'mnémoniques'.

Il y a plusieurs façons de mettre et de sortir des nombres de l'accumulateur et d'autres parties de l'ordinateur; ces diverses manières sont connues sous le nom "modes d'adressages". la méthode ci-dessus, qui utilise un nombre entré non-modifé, s'appelle "mode immédiat" d'adressage

L'instruction suivante du programme est ADD:

```
ADD A,n     Ajouter (ADD) la valeur immédiate n à l'accumulateur
```

Cette instruction perd le nombre qui suit la virgule, dans laquelle '1' (page 1), et l'ajoute à la valeur contenue actuellement dans l'accumulateur (A). La réponse est placée dans l'accumulateur, en effaçant l'ancien nombre, en l'occurrence 83.

Revenons à cet exemple. L'instruction la plus déroutante est vraisembablement:

```
      CALL 47962
```

Elle ordonne à l'ordinateur de stocker le contenu de l'accumulateur, 84, sur l'écran - c'est-à-dire d'afficher ce contenu. bien que celui soit tout à fait vrai, ce n'est pas tout.

L'ordinateur possède de nombreux programmes ou 'routines" en code machine intégrés, mémorisés dans ses chips. Sans ceux-ci, il ne serait, par

exemple, pas capable de comprendre le BASIC. L'une de ces routines est conçue pour prendre une valeur de l'accumulateur et l'afficher sur un moniteur.

Pour atteindre ces routines afin de les utiliser, on emploie CALL. Dans l'ordinateur, la première instruction de toute routine incorporée est mémorisée dans une case mémoire spécifique, celles ci sont numérotées de 0 à 65535. Le numéro cité précédemment, 47962, est l'adresse en mémoire de la première instruction dans la routine 'inscrivez la valeur de l'accumulateur sur l'écran'.

Bien ! Essayons maintenant d'exécuter un programme en code machine !

Quelques précisions sur l'assembleur : lorsque vous commencez à écrire un programme, il faut dire à l'assembleur à quel endroit vous voulez que le programme soit placé dans la mémoire de l'ordinateur. Pour le moment, l'assembleur décidera où mémoriser le programme. On lui indique cela par la commande ENT.

Ainsi, la première ligne du programme sera :

```
10 ENT
   '
   '
   programme
   '
   '
```

Les programmes en langage assembleur peuvent être lancés (ou appelés) à partir du BASIC ou appelés à partir de l'assembleur en utilisant l'option appel (Call) programme.

Dans l'un ou l'autre cas, il faut dire à l'ordinateur de revenir du code machine au BASIC (le programme assembleur). La commande qui fait cela est RET.

RET Retour (RETurn) de la routine code machine

Bon, pour faire notre exemple dans un programme, il faut:

1. Dire à l'assembleur de décider de stocker le programme

2. Charger (LoaD) la valeur immédiate '83' dans l'Accumulateur, soit : 'LD A,83'

3. Ajouter 1 au contenu de l'Accumulateur. La mnémonique pour cela est 'ADD A,1'

4. Appeler (CALL) la routine code machine incorporée pour l'affichage sur l'écran de la nouvelle valeur de l'accumulateur, soit : 'CALL 47962'

5. Retour (RETurn) du programme code machine au BASIC, soit 'RET'

6. Dire à l'assembleur (pas au Z80) que le programme est terminé, en utilisant " @ "

ou

PROGRAMME 1.1 (Ne le tapez pas pour le moment)

```
10 ENT
20 LD A,83
30 ADD A,1
40 CALL 47962
50 RET
```

Maintenant pour entrer cela :

a) Charger le programme assembleur dans l'ordinateur de la manière suivante:

1) Faites un Reset (réinitialisation) complet de l'ordinateur en tenant appuyées en même temps les touches : SHIFT, CTRL et ESC, puis relâchez-les.

2) Rembobinez complètement la cassette (face A).

3) Appuyez sur CTRL et la petite touche bleue ENTER, puis relâchez-les. Le message suivant devrait apparaître alors sur l'écran:

RUN"

Appuyez sur PLAY puis sur n'importe quelle touche:

Si cela n'apparaît pas recommencer à l'étape 1.

4) Appuyez sur la touche PLAY du lecteur de cassette et appuyez ensuite sur la barre d'espace.

5) Un message apparaîtra alors sur l 'écran après quelques secondes, vous informant que l'assembleur est en train d'être chargé.

b) Appuyez sur CAPS LOCK afin de taper en lettres majuscules, puis tapez 'I', sans les guillemets bien sûr, pour commencer à entrer un nouveau programme. Entrez 10 en réponse au message "Entrez début et incrément",puis appuyez sur la touche ENTER.

c) Un 10 apparaîtra; tapez 'ENT' puis appuyez sur la touche ENTER.

d) Un nouveau numéro de ligne apparaîtra alors,20. Tapez la ligne suivante du programme, 'LD A,83' et appuyez sur la touche ENTER comme avant. n'oubliez pas l'espace entre 'Ld' et 'A'. Ne mettez pas d'espace entre la virgule et le nombre, ou à n'importe quel autre endroit.

e) 30 apparaîtra ensuite. Tapez 'ADD A,1', puis appuyez sur la touche ENTER. Là aussi n'oubliez pas l'espace entre ADD et A.

f) En ligne 40, tapez 'CALL 47962' puis appuyez sur la touche ENTER. Il doit y avoir un espace entre CALL et 47962

g) En ligne 50, tapez 'RE' et appuyez sur la touche ENTER.

h) Maintenant que nous avons fini de taper le programme, et que nous voulons revenir au mode commande, appuyez sur @ puis sur la touche ENTER.

i) Maintenant appuyez sur M pour revenir au menu, puis sur L pour lister le programme, en entrant 10 comme numéro de ligne de départ du programme. (Remarque: vous auriez pu sélectionner l'option Liste tout de suite sans passer par le menu.) Le programme devrait alors apparaître comme ci-dessous

```
10 ENT
20 LD A,83
30 ADD A,1
40 CALL 47962
50 RET
```

Si pour quelque raison qu ce soit, le programme n'apparaît pas comme ci-dessus, sélectionnez alors le mode de remplacement 'R' et remplacez la ou les lignes contenant une ou des erreurs. Lorsque votre programme apparaît comme ci-dessus, une fois listé, appuyez sur A pour assemblage . L'assembleur
assemblera alors le programme en code machine. Sélectionnez l'option 2.

j) Après que l'assembleur ait assemblé le programme, il retournera à '>'. S'il y avait une erreur dans le programme, l'assembleur vous en aurait informé. La fonction remplacer ou insérer serait alors utilisée pour corriger ces erreurs.

k) Pour voir le programme fonctionner, sélectionnez l'option appel de programme.

Si vous vous demandez pourquoi le programme met un T majuscule sur l'écran et non la réponse 84, remportez vous à l'appendice III du manuel du Basic pour l'Amstrad qui indique le 'jeu de caractère ASCII'. Vous verrez que le code de 'T' est 84.

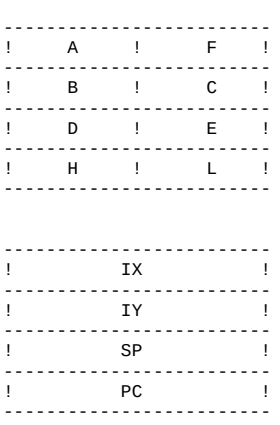
ASCI signifie : American Standard Code for Information Interchange , et
chaque symbole que l'ordinateur peut afficher sur l'écran à son propre numéro de code ASCII. Ainsi, lorsqu'on lui demande d'afficher 84 sur l'écran, la routine appelée par CALL 47962 met le caractère numéro 84 sur l'écran.

Ceci est également vrai pour le BASIC, même pour une addition telle que 'PRINT 83+1'. Dans ce cas, bien que l'ordinateur afficher '84' sur l'écran, il a dû chercher ce que signifiaient les caractères '83+1', faire la somme appropriée et convertir la réponse en utilisant les codes ASCII pour les caractères '8' et '4' . Ceci est en fait relativement compliqué à faire. L'ordinateur a une routine incorporée qui fait cela automatiquement pour le BASIC.

ARCHITECTURE DU MICROPROCESSEUR Z80

Le Z80 possède différents registres autres que l'accumulateur, et nous vous montrons ci-dessus en figure 1.1 un diagramme simplifié de leur disposition :

figure 1.1



Architecture simplifiée du Z80

L'accumulateur, ou registre A, vous l'avez déjà rencontré. Il est un peu spécial dans le sens où il y a plus d'instruction Z80 qui peuvent agir sur lui qu'il n'y en a pour les autres registres.

Tous les registres (seulement des positions mémoire dans le microprocesseur Z80) sont des registres de 'huit bits", c-à-d qu'ils ne peuvent mémoriser que des nombres de 0 à 255. Cela ne semble pas terriblement utile, mais il existe des moyens d'utiliser les registres en les combinant pour mémoriser des nombres bien plus grands, comme nous le verrons par la suite.

Les registres H et L pris ensemble sont répertoriés sous le nom de 'pointeur de données principal'. Les registres B et C, et D et E lorsqu'ils sont utilisés en paires sont appelés 'pointeur de données auxiliaires'. Le plus important pour le moment est de savoir que des instructions utilisant H et L sont souvent plus courtes lorsqu'elles sont assemblées en code machine, et par conséquent fonctionnent plus rapidement.

ADRESSAGE IMMEDIAT DES REGISTRES

L'adressage immédiat de l'accumulateur a été observé auparavant, lorsqu'un nombre était chargé dans l'accumulateur en utilisant l'instruction

```
LD A,83
```

Une instruction de forme exactement identique peut être utilisée pour charger des nombres de 0 à 255 dans tout autre registre, par exemple :

```
LD B,83
```

pourrait charger le nombre 83 dans le registre B.

En général,

Ld r,n Charger (LoaD) dans le registre r, la valeur immédiate n.

Le 'r' représente ici n'importe quel registre, soit A, B, C, D, E, H ou L.

ADRESSAGE DE REGISTRE A REGISTRE

Non seulement on peut charger un nombre dans tout registre, mais tout registre peut être recopié dans un autre registre, ou dans l'accumulateur. Par exemple, le programme 1.2 inscrit le nombre 43 dans le registre L, le recopie dans l'accumulateur, et l'affiche sur l'écran : un '+' ASCII.

PROGRAMME 1.2

```
10 ENT
20 LD L,43              Chargement en mode immédiat dans L de 43
30 LD A,L               Charger dans A le contenu de L.
40 CALL 47962           Afficher le contenu de l'accumulateur.
50 RET                  Revenir au BASIC (assembleur)
```

La ligne 30 contient la nouvelle instruction:

```
LD A,L
```

Cela recopie le contenu du registre L dans l'accumulateur:

```
LD A,L
```

Cela recopie le contenu du registre L dans l'accumulateur.

un autre exemple pourrait être :

```
LD B,E
ou
LD C,A
```

En général,

```
LD r1,r2    Recopier le contenu du registre r2 dans le registre r1
```

Quelques auteurs se réfèrent à l'adressage de registre à registre en tant que 'adressage inhérent' et d'autres s'y réfèrent en tant qu'adressage 'implicite'. Si vous utilisez d'autres livres méfiez-vous de leur terminologie, car cela peut prêter à confusion.

Bien, tapons maintenant le programme 1.2. Si vous faites une erreur en tapant une ligne, avant d'appuyer sur la touche ENTER, utilisez la touche DELETE pour effacer l'erreur et tapez la correction. Si vous remarquez une erreur dans la ligne après avoir appuyé sur la touche ENTER, vous pouvez remplacer ces lignes après avoir appuyé sur @ et avant d'assembler le programme.

1. Charger et lancer le programme d'assemblage s'il n'est pas déjà chargé; si c'est le cas, revenez au menu avec la commande M si nécessaire.
2. Sélectionner 'I' pour commencer à entrer le programme, et appuyez sur CAPS LOCK pour mettre le mode en majuscule.
3. Dites à l'assembleur à quel endroit de la mémoire le programme doit commencer, soit : taper 'ENT' et appuyer sur ENTER.
4. Taper 'LD L,43' et appuyer sur ENTER (appuyer sur la touche ENTER après chaque entrée.)
5. Taper 'LD A,L'
6. Taper 'CALL 47962'
7. Taper 'RET'
8. Appuyer sur @
9. Sélectionner 'L' pour lister le programme: vérifiez-le !
10. Sélectionner 'A' pour assembler le programme.
11. Sélectionner 'C' pour lancer le programme.

Après cela, vous devriez être capable d'écrire quelques programmes tout seul. Essayez le petit exercice suivant. N'oubliez pas d'entrer l'instruction 'RET', autrement l'ordinateur se promènera dans sa mémoire après le programme, à la recherche d'instructions à exécuter. Il trouvera assurément ou presque quelque chose, et ce quelque chose entrainera un 'plantage' du système. En d'autres termes il n'y aura plus d'autre solution que d'éteindre et de recommencer à zéro - et par conséquent de recharger le programme d'assembleur.

EXERCICE 1.1

En utilisant l'adressage immédiat, charger 65 dans l'accumulateur, puis afficher cela sur l'écran. 65 est le code ASCII pour A majuscule. Une réponse possible est donnée dans le chapitre solutions.

CALL 47962 PLUS EN DETAIL

Si cet appel est utilisé plusieurs fois à la suite, plusieurs choses peuvent être affichées sur l'écran dans des positions consécutives.

Essayez ce programme :

```
10 ENT
20 LD A,72        Charge l'accumulateur avec 'H'
30 CALL 47962     Afficher 'H'
40 LD A,69        Charge l'accumulateur avec 'E'
50 CALL 47962     Afficher 'E'
60 LD A,76        Charge l'accumulateur avec 'L'
70 CALL 47962     Affiche 'L'
80 CALL 47962     Affiche 'L' à nouveau
90 LD A,79        Charge l'accumulateur avec 'O'
100 CALL 47962    Afficher 'O'
110 RET           Revenir au BASIC
```

Ce programme devrait afficher le mot 'HELLO" sur l'écran, après avoir été assemblé et lancé.

EXERCICE 1.2

Ecrivez votre nom en haut à gauche de l'écran. Une réponse pour 'FRED' est donnée dans le chapitre solutions.

Ce chapitre est terminé. Ce n'était pas si terrible, non! Maintenant vous devriez savoir ou pouvoir traduire ce qui suit :

```
LD A,n
LD r1,r2
RET
@

CALL 47962
ADD A,n
```

Adressage de mode immédiat
Adressage registre-registre.

Et cela ? (r est n'importe quel registre):

```
ADD A,r
```

CHAPITRE 2

SAUTS, SOUS-PROGRAMME ET ETIQUETTES

Peu de programmes réels se déroulent en une phase régulière et interrompue sans saut ou branchement en quelque étape. Ce chapitre traite de ces commandes de sauts et de leurs utilisations. Puis, le chapitre en vient à étudier les flags qui permettent à ces sauts d'être conditionnels et inconditionnels. les étiquettes symboliques sont également abordées. Celles-ci sont une caractéristique puissante et utile de l'assembleur et aident considérablement au développement des programmes.

SAUTS INCONDITIONNELS

Ceux-ci ordonnent au programme de sauter bon gré mal gré - pas de conditions.
Le jeu d'instructions du Z80 contient cinq sauts de cette catégorie; pour le moment nous nous occuperons seulement de deux de ceux-ci. Les trois autres seront présentés plus loin.

Le premier à prendre en considération est :

JP nn Sauter (JumP) à l'adresse spécifiée nn

Par exemple, JP 200 signifie de sauter à la case mémoire 200.

Mis dans un programme JP se présente de la manière suivante :

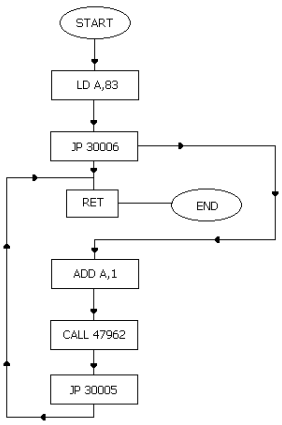


Figure 2.1

Utilisation de JP dans un programme

Une telle routine de saut n'accomplit en fait pas grand chose, mais elle pourrait, par exemple, être utilisée pour imbriquer un peu de code dans un programme. En figure 2.1 par exemple, les commandes ' ADD A,1', 'CALL 47962' et 'JP 30005' on été effectivement insérées dans le programme.

Nous allons maintenant taper ce programme sur l'ordinateur. N'oubliez pas de tapez ENTER à la fin de chaque ligne, et de sélectionner ' I' pour commencer à entrer un programme, et d'entrer 10 comme numéro de ligne de départ.

Notez bien que la première ligne du programme n'est pas ENT. Etant donné que le programme saute à des cases mémoire spécifique, le départ exact du programme doit être connu. ORG 30000 fait que l'assembleur stocke le code machine en mémoire, en commençant à la case mémoire 30000. Reportez-vous à l'annexe 6 pour plus de détails.

PROGRAMMES 2.1

```
ORG 30000
LD A,83
JP 30006
RET
ADD A,1
CALL 47962
JP 30005
```

Maintenant assemblez le programme avec l'option A, sélectionnez l'option 2.

L'affichage de l'écran du programme devrait ressembler à quelque chose comme ça :

```
-----------------------------------------
!  CASE    !  NUMERO  !  CODE  !  CODE     !
! MEMOIRE  ! DE LIGNE ! OBJET  ! SOURCE    !
-----------------------------------------
!  7530    !   10     !        ! ORG 30000 !
!  7530    !   20     ! 3E53   ! LD A,83   !
```

```
!  7532    !   30     ! C33673 ! JP 30006  !
!  7535    !   40     ! C9     ! RET       !
!  7536    !   50     ! C601   ! ADD A,1   !
!  7538    !   60     ! CD5ABB ! CALL 47962!
!  753B    !   70     ! 70     ! JP 30005  !
-------------------------------------------
```

FIGURE 2.2

Le premier nombre dans la colonne case mémoire semble ne pas correspondre à 30000, là où le programme devrait commencer. C'est parce que les cases mémoire sont représentées en hexadécimal. Ne vous en occupez pas pour le moment, rappelez-vous simplement que 30000 est 7530 en hexadécimal, 30006 est 7536 et 30005 est 7535.

Quand des sauts sont utilisés de cette manière, il est nécessaire d'indiquer qu programme l'endroit où il doit sauter, soit une adresse, d'où JP 30006. Le calcul de ces adresses est tout ce qu'il y a de plus simple si on le fait systématiquement, comme nous le démontrerons plus loin.

Observez la figure 2.2, les nombres et les lettres dans la deuxième colonne, le code objet, sont ce à quoi ressemble le langage assembleur après qu'il ait été assemblé - soit le code machine. Chaque paire de caractère alphanumériques (autrement dit de nombre et/ou de lettres) dans cette colonne représente une partie du langage assembleur initial. Ainsi, '3E' représente 'LD A' et '53' représente '83' (Ne vous demandez pas pour quelle raison! C'est en hexadécimal et nous l'expliquerons dans un chapitre ultérieur.) Pour les nombres en langage assembleur plus grands que 255, quatre alphanumériques - code machine sont requis. Ainsi, 'C3' représente 'JP' et '3075' représente '30000'. Comment cela peut-il nous aider à calculer les adresses auxquelles il faut sauter ? Hé bien, chaque paire de chiffre remplit une case mémoire - un octet de mémoire. Par conséquent, puisque l'ont sait que la première instruction du programme se trouve ne case mémoire 30000 (grâce à ORG 30000), il nous suffit de compter les paires d'alphanumérique - code machine.

Par exemple, pour sauter à l'instruction ' ADD A,1', il nous faudrait sauter à la case mémoire contenant 'C6'. Donc, en commençant par 30000, qui contient '3E', nous avons 53, C3, 36, 75, C9, et puis C6. Il y a donc 6 cases mémoire à compter de 30000, c'est donc en 30006. Par conséquent le programme devra sauter à 30006. Il est convenu qu'avant d'entrer 30006 dans les instructions JP, ce ne sera pas à vous de calculer '3675'. Malgré cela, comme vous savez que l'adresse en code machine sera composée de quatre caractères (deux octets), cela ne vous posera pas de problème à condition d'écrire votre programme sur une feuille avant de le taper, - ce qui est une bonne méthode de toute façon.

Les annexes contiennent des tables indiquant combien d'octets sont requis
par chaque instruction, ce qui vous aidera à calculer les adresses de sauts.

Voici une décomposition du programme:

ORG — N'utilise pas de mémoire, il est là pour être utilisé par l'assembleur, et on l'appelle une 'directive assembleur'

LD A,83 — Demande 2 octets - le premier est le code objet pour LD A, et le second est la donnée.

JP 30006 — Demande 3 octets - un pour son code objet, C3, les deux autres sont la case mémoire à laquelle il faut sauter.

RET — Ne demande qu'un octet pour son code objet, C9.

ADD A,1 — Demande 2 octets. Le premier, C6, est le code objet pour 'ADD A', et le second est la donnée.

CALL 47962 — Cette instruction demande 3 octets, CD est le code objet et 5A et BB la case mémoire qui est appelé.

JP 30005 — C'est la même chose que JP 30006, sauf que la case mémoire à laquelle il faut sauter est différente.

En comptant le nombre d'octets nous pouvons calculer la longueur du programme: le programme 2.1 est long de 14 octets.

En utilisant l'opérateur JP nous pouvons sauter dans toute case mémoire de 0 à 65535. Si la case mémoire à laquelle il faut sauter est située entre +129 ou -126 octets par rapport à la case mémoire actuelle, on peut utiliser le saut relatif au lieu de l'instruction de saut. L'avantage de cette instruction alternative est qu'elle est plus rapide, puisqu'elle utilise deux octets seulement en mémoire par opposition aux trois octets qu'utilise l'instruction JP.

JR e — Saut (JumP) relatif à l'adresse e

Par exemple, si vous êtes à l'adresse 30000,et que vous voulez sauter à l'adresse 30045, vous écrirez

JR 30045

Vous ne direz pas JR 39, l'ordinateur calculera automatiquement le décalage sur un octet. Notez bien que JR ne peut être utilisé que si vous sauter à une adresse allant de +129 ou -126 de la case mémoire contenant le 'JR' proprement dit.

LE REGISTRE D'INSTRUCTION

Avant d'aller plus loin, il faut observer la façon dont l'instruction de saut accomplit sa tâche. Observez la figure 2.2, on peut voir que le programme commence à la case mémoire 30000. Dans le corps du programme nous lui demanderons de sauter aux cases mémoires 30006 et 30005. Comment fait l'ordinateur pour repérer l'endroit où il se trouve? Dans le microprocesseur Z80 il y a un registre de 16 bits appelé le compteur de programme (Program Counter:PC). Ce registre contient l'adresse de l'instruction en cours. Lorsqu'un programme machine est lancé, le compteur de programme est fixé sur la première case mémoire du programme. Après exécution de la première instruction, le compteur de programme est mis à jour, de façon à ce qu'il pointe sur l'instruction suivante. Ainsi, une instruction de saut charge dans PC l'adresse à laquelle il faut sauter, autrement dit : il le pointe sur la nouvelle instruction. L'effet en est que le programme continue l'exécution à partir d'une instruction contenue dans la case mémoire sur laquelle l'adresse de saut était pointée. D'autres instructions sont disponibles pour modifier le contenu du PC et de ce fait modifient le cours du programme, nous vous les
présenterons lorsque cela sera nécessaire.

La figure 2.3 illustre l'exécution du programme précédent.

programme	PC avant exécution	PC après exécution
ORG 30000	?	30000
LD A,83	30000	30002
JP 30006	30002	30006
ADD A,1	30006	30008
CALL 47962	30008	30011
JP 30005	30011	30005
RET	30005	Retour à l'assembleur

Jusqu'ici l'ordinateur a sauté inconditionnellement à une autre section de code. Bien qu'utile, cela le serait encore plus si l'on pouvait faire sauter l'ordinateur si s'est vérifiée une certaine condition. On peut faire cela avec le groupe d'instruction de sauts conditionnels.

SAUTS CONDITIONNEL

Tout programme demandant un test de certaines conditions nécessite des sauts conditionnels. En BASIC, l'analogie est la commande IF...THEN.

Soit: 10 IF X=Y THEN GOTO 500

Cette ligne fait que l'ordinateur compare les variables X et Y et si elles sont identiques il va à la ligne 500.

Le Z80 accomplit cette opération en utilisant un registre spécial, le registre de flags. Le registre de flags est une registre de huit bits comme l'accumulateur et les registres B, C, D, E, H et L, mais il est utilisé tout différemment. Alors que les autres registres sont utilisés pour mémoriser et manipuler des octets, 3e registre de flags est traité comme s'il contenait huit bits individuels qui sont utilisés comme signaux ou flags. Le Z80 ne traite normalement qu'un flag a la fois, en fixant la valeur du bit soit à 0 soit a 1. Il peut également tester un flag pour déterminer s'il est mis (1) ou annulé (0).

Par exemple un des flags est le flag 1 ou flag Zéro, Chaque fois qu'une opération arithmétique est réalisée et qu'elle produit un résultat de zéro, le flag zéro est mis a 1 - sinon il sera annulé à '0', indiquant ainsi que l'opération précédente n'a pas eu zéro pour résultat.

De nombreuses autres instructions peuvent fixer ce flag, l'une d'elles étant;

DEC d DECrémente le registre d
Cette instruction décrémente le contenu de 'd' où 'd' est un des registres suivants:

B,C,D,E,H,L,A

Si après décrémentation du registre spécifié, le résultat est égal à
zéro, alors le flag zéro est mis, ou sinon annulé.

Notez que l'instruction DEC est appelée OPERATEUR, et d, l'OPERANDE; ainsi l'OPERATEUR travaille sur l'OPERANDE. Quelques OPERATEURS tels que LD A,10 demandent deux OPERANDES, ici, A et 10.

Continuons la programmation. Le programme 2.2 affichera dix A sur l'écran.

PROGRAMME 2.2

```
ORG 30000
LD C,10
LD A,65
CALL 47962
DEC C
JR NZ,30004
RET
```

Tapez ce programme et assemblez-le avec l'option A, puis l'option 2, Pour lancer le programme, revenez en BASIC avec X et tapez CALL 30000. Vous verrez dix A s'afficher sur l'écran. Pour revenir à l'assembleur appuyez sur la touche 'point décimal' du pavé numérique.

Observons le programme, la seule ligne qui n'a pas été encore rencontrée est JR NZ. Cet opérateur teste l'état actuel du flag zéro et fait un saut relatif (Jumps Relative) à 30001 si la dernière instruction arithmétique a un résultat non-zéro. Ainsi le programme met le contenu de l'accumulateur sur l'écran, DECremente le registre C et vérifie si le flag zéro a été mis par l'instruction de DECrémentation. S'il ne l'a pas été, autrement dit si c'est Non Zéro, il fait alors un saut relatif a 30004, sinon il va à la ligne suivante du programme où il RETourne au programme principal (ici le Basic).

JR NZ,e Saut relatif, si résultat Non Zéro, à l'adresse e

EXERCICE 2.1

Réécrivez le Programme 2.2 pour utiliser le registre B à la place du registre C.

Une réponse possible est donnée dans le chapitre solutions.

EXERCICE 2.2

Pourquoi avons-nous utilisé JR NZ,e au lieu de l'instruction alternative JP NZ,e?

La réponse de cet exercice se trouve dans le chapitre solutions.

Remarque: comme on pouvait le Penser, une instruction pour INCrémenter un opérande existe également:

INCrémenter l'opérande d

Jusque la l'ordinateur a été programmé pour sauter inconditionnellement ou en cas de résultat zéro d'une opération. Dans les deux cas il est nécessaire de savoir à quelle case mémoire il faut sauter. Alors que ceci est possible pour des programmes courts, cela devient de plus en plus difficile de calculer ces adresses de sauts, sans parler du gaspillage de temps, lorsque la longueur des programmes s'accroît. Pour surmonter cela on utilise des étiquettes.

ETIQUETTES

L'utilisation des étiquettes permet au programme d'être dirigé sur une Instruction nommée, sans qu'il soit nécessaire de calculer les adresses de sauts. Un terme plus fantaisiste pour étiquette est ETIQUETTE SYMBOLIQUE, car l'étiquette elle-même est symbolique de la Position en mémoire. Par exemple, l'instruction

JP LOOP:

fera que l'assembleur remplacera LOOP (boucle) par une adresse, qui a été assignée auparavant à LOOP, chaque fois qu'il rencontrera cette étiquette.

Pour dire à l'assembleur qu'une étiquette est une étiquette, Il est nécessaire de faire suivre l'étiquette d'un double point (:). D'autre part les étiquettes doivent être inférieures ou égales à six caractères, Ainsi, par exemple, si le programme devait sauter à la case mémoire contenant l'instruction DEC C, les lignes de programme suivantes seraient utilisées.

.

```
    .
  JP LOOP
    .
    .

LOOP : DEC C
    .
    .
```

D'autres conventions doivent être observées lorsque l'on utilise les étiquettes. Le double point, par exemple, doit suivre le dernier caractère de l'étiquette (pas d'espace entre eux). D'autre part, un espace doit suivre le double point. C'est simplement pour permettre à l'assembleur de trouver où l'étiquette se termine. Ne vous tracassez pas si vous oubliez cela, l'assembleur repérera toute erreur et vous en informera.

En résumé:

ETIQUETTES

1 Une étiquette doit être constituée de 6 caractères au maximum,

2 Un double point doit immédiatement suivre l'étiquette.

3 Un espace doit suivre le double point.

4 L'étiquette ne doit pas contenir d'espace.

Par exemple :

```
LOOP:
TEST:
NXTCHR:
```

sont des étiquettes correctes. Les suivantes ne le sont pas :

```
LOOP :
BACKONE:
NEXT L:
```

Pour illustrer l'utilisation des étiquettes, nous réécrirons le programme 2.1 en utilisant des étiquettes.

Notez que la directive ENT peut être utilisée au lieu de la directive ORG , puisque aucun saut absolu n'est utilisé.

PROGRAMME 2.3

```
ENT

      LD A, 83
      JP NXT:
END:  RET
NXT:  ADD A,1
      CALL 47962
      JP END:
```

Lorsque vous listez le programme ci-dessus, vous verrez qu'il est affiché avec les étiquettes sur la gauche du corps principal du programme, le rendant plus facile à lire, il se présente ainsi:

```
      ENT
      LD A,83
      JP NXT:
END:  RET
NXT:  ADD A,1
      CALL 47962
      JP END
```

Pour faciliter la frappe des programmes, ceux-ci seront présentés de cette manière tout au long du livre. Néanmoins, vous les taperez bien sur normalement.

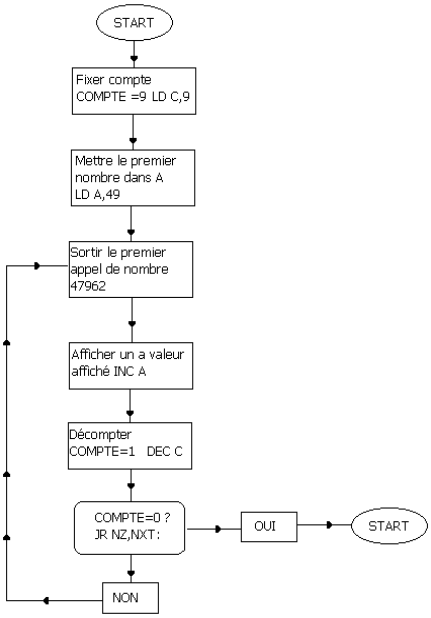
En utilisant les Instructions présentées Jusqu'ici, il est possible de sauter inconditionnellement ou conditionnellement à n'importe quel endroit de la mémoire, ou en utilisant une commande de saut relatif, à n'importe quelle adresse située entre +129 ou -126 de l'adresse actuelle. Des étiquettes peuvent également être utilisées pour faciliter le calcul des adresses de sauts.

Toutes ces commandes seront désormais combinées pour afficher les nombres de un à neuf sur l'écran. Rappelez-vous, au chapitre 1 nous avons vu que le code ASCII pour T était 84. Bien, les nombres de un a neuf ont aussi un code. Regardez la table ASCII (dans le manuel BASIC appendice 3) et regardez si le code de 1 est bien 49 et celui de neuf est 57. Vous êtes d'accord?

Maintenant combinez ce que l'on a appris jusque la pour résoudre le problème, c'est à dire; Comment mettre le contenu de l'accumulateur sur l'écran, le charger, et l'incrémenter. Et aussi, comment décrémenter un registre et tester si zéro. Si vous avez une idée pour amener l'ordinateur à afficher les nombres de 1 à 9 sur l'écran, fermez le livre et essayez; si vous n'en avez pas, ne vous tracassez pas nous expliquerons tout cela.

L'organigramme suivant explique la solution:

Problème : Afficher les nombres de 1 à 9 sur l'écran.



Traduit en un programme, cela donne :

PROGRAMME 2.4

```
      ENT         Fixe le début du programme,
      LD C,9      Charger dans C le nombre de caractères à inscrire, soit 9,
      LD A,49     Charge dans A le code de "1",
NXT:  CALL 47962  Affiche le contenu de l'accumulateur sur l'écran,
      INC A       Ajoute 1 au contenu de A, chargeant ainsi dans A le code de "2"
      DEC C       Décrémente le registre d'instructions,
      JR NZ,NXT:  Si DEC C a un résultat autre que zéro, faire un saut relatif à la case adressée
                  Par l'étiquette NXT.
      RET         Si C=0 alors retour a l'assembleur.
```

Avant de le lancer, examinons-le étape par étape:

Numéro d'étape	Accumulateur	Registre C	Flag Z
1	49	9	0
2	50	8	0
3	51	7	0
4	52	6	0
5	53	5	0
6	54	4	0
7	55	3	0
8	56	2	0
9	57	1	0
10	58	0	1

le flag Z est mis (1)le programme se termine et revient à l'assembleur, (Etape 10)

Maintenant assemblez et lancez le programme.

EXERCICE 2.3

Ecrivez un programme qui inscrira l'alphabet sur l'écran. (Pour vous aider, sachez que le code de A est 65). Une réponse possible est donnée au chapitre solutions.
Jusque là, seul le flag zéro, un des 9 flags possibles, a été utilisé. Voici les autres.

LES FLAGS

Le flag de retenue (Carry Flag (C))

Ce flag est mis lorsque une addition ou une soustraction a pour résultat une retenue. Il est aussi utilisé dans certaines instructions de décalage et de rotation.

Flag de soustraction (N)

Ce flag est plutôt utilisé par le Z80 que par des programmes pour certaines opérations arithmétiques.

Flag de Parité/Dépassement (P/V)

Ce flag a deux fonctions distinctes, la première pour indiquer la parité d'un résultat. La "parité" du résultat est obtenue en ajoutant tous les Uns de sa représentation binaire. Si le résultat est pair alors la parité est mise, sinon elles est annulée, c'est à dire = 0. Le flag est également mis lorsque des dépassements se produisent lors de certaines opérations arithmétiques.

Le flag de demi-retenue (H)

Le flag de demi-retenue est utilisé par le Z80 pour des Instructions "Décimales Codées en Binaires", Ne vous en occupez pas pour le moment.

Le flag Zéro (Z)

Le flag zéro est mis lorsque une opération a pour résultat zéro, et en tant que tel, est utilisé très souvent pour des comparaisons.

Le flag Signe (S)

Ce flag indique le signe du résultat arithmétique ou d'un octet qui est transmis. Fondamentalement, il teste le bit 7 d'un octet, et il est mis s'il est égal a un et annulé dans l'autre cas.

SOUS-PROGRAMMES

De même que nous avons comparé l'Instruction JP avec l'instruction BASIC IF X=Y THEN 500, il existe une instruction Z80 similaire à l'instruction BASIC GOSUB 500. Cette instruction, CALL, a déjà été utilisée pour inscrire le contenu de l'accumulateur sur l'écran.

```
CALL nn Appel      (CALL) de la routine commençant à l'adresse mémoire nn
                   (nn peut être représenté par une étiquette).
```

Comme en BASIC, une instruction de retour (return) est requise a la fin du sous-programme.

```
RET                Retour du sous-programme appelé en dernier.
```

Voyons comment ces instructions peuvent être utilisées dans un programme.

Problème : Inscrire sur l'écran les caractères ASCII de 200 a 250, en utilisant un sous-programme 'imprimer les caractères'.

PROGRAMME 2.5

```
            ENT
            LD A,200          A= 1er code ASCII
            LD B,50           B= compte
NXT:        CALL PRINT:       Appel routine d'impression
            DEC B             Décrémenter le compte
            INC A             Code ASCII suivant
            JR NZ,NXT:        Si B=0 sauter a NXT
            RET
PRINT:      CALL 47962
            RET
```

Assemblez-le puis lancez-le.

Pour aider à la compréhension du programme l'étiquette PRINT a été assignée au sous-programme d'impression. Outre les appels inconditionnels il existe des appels conditionnels, de même qu'il y a des sauts conditionnels et inconditionnels.

```
CALL cc,nn          Appel du sous-programme commençant à l'adresse mémoire nn, si la
                    condition ce est remplie.
```

Les conditions oui font que la routine est appelée sont exactement les mêmes que celles utilisées pour des sauts conditionnels. Lorsqu'une instruction CALL est exécutée, le contenu actuel du registre d'instructions est sauvegardé. Le registre d'instructions reçoit alors l'adresse d'appel nn, Quand le Z80 a terminé le sous-programme, c'est à dire a atteint une instruction RET, le registre d'instructions reçoit la valeur sauvegardée du registre d'instructions.

```
Problème :   Afficher 100 "A" sur l'écran, mais afficher un espace pour chaque
             dizaine de A affichée.
```

Problème : Inscrire sur l'écran les caractères ASCII de 200 à 250, en utilisant un sous-programme 'imprimer les caractères'.

PROGRAMME 2.5

```
            ENT
            LD A,200          A= 1er code ASCII
            LD B,50           B= compte
NXT:        CALL PRINT:       Appel routine d'Impression
            DEC B             Décrémenter le compte
            INC A             Code ASCII suivant
            JR NZ,NXT:        Si B=0 sauter à NXT
            RET
            CALL 47962
            RET
```

Assemblez-le puis lancez-le.

Pour aider à la compréhension du programme l'étiquette PRINT a été assignée au sous-programme d'impression. Outre les appels inconditionnels il existe des appels conditionnels, de même qu'il y a des sauts conditionnels et inconditionnels.

```
CALL cc,nn          Appel du sous-programme commençant à l'adresse mémoire nn,
                    si la condition ce est remplie.
```

Les conditions qui font que la routine est appelée sont exactement les mêmes que celles utilisées pour des sauts conditionnels. Lorsqu'une instruction CALL est exécutée, le contenu actuel du registre d'instructions est sauvegardé. Le registre d'instructions reçoit alors l'adresse d'appel nn. Quand le Z80 a terminé le sous-programme, c'est à dire a atteint une instruction RET, le registre d'instructions reçoit la valeur sauvegardée du registre d'instructions.

```
Problème    Afficher 100 "A" sur l'écran, mais afficher un espace pour
            chaque dizaine de A affichée.
```

Représentation de la solution d'un organigramme :

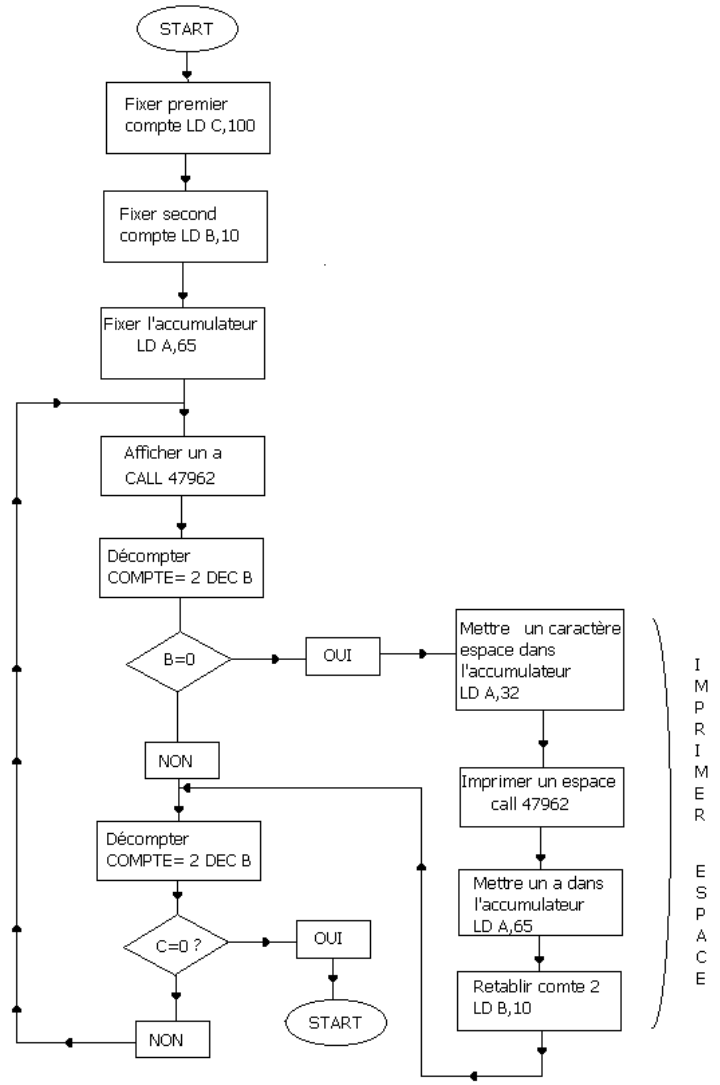


Figure 2.5

Conversion en programme :

```
            ENT
            LD C,100          C= 1er compte
            LD B,10           B= 2ème compte
            LD A,65           A= "A"
NXT:        CALL 47962        Affiche un 'A'
            DEC B
            CALL Z,PSPC       Si B=0 afficher un espace
            DEC C
            JR NZ,NXT:        100 'A' sont affichés? Si non, sauter à l'espace
                             suivant (NXT).
            RET               Retour a l'assembleur

PSPC:       LD A,32           A=ASCII pour un espace.
            CALL 47962        Afficher un espace
            LD A,65           A="A"
            LD B,10           Recharger B comme compte
            RET               Retour au programme principal
```

Essayez maintenant de lancer le programme et voilà, une centaine de A apparaissent sur l'écran séparés tous les 10 par un espace. Cette forme d'appel conditionnel est très pratique car elle facilite considérablement récriture de programmes structurés.

RESUME

Le compteur de programme (PC) est un registre de 16 bits, contenant l'adresse mémoire de l'instruction en cours à l'intérieur du programme.

Le registre de flags contient 8 bits, qui reflètent l'état de la section arithmétique du Z80.

Des étiquettes peuvent être utilisées dans les Instructions de saut. Une étiquette doit contenir moins de sept caractères, et être terminée par un double point immédiatement suivi par un espace.

Il existe deux formes principales de sauts, conditionnelle et inconditionnelle. Ces deux formes peuvent être encore divisées chacune en sauts absolus, c'est a dire, JP NXT ou sauts relatifs: JR NXT.

Deux types d'appel de sous-programmes (CALL) existent, conditionnel et inconditionnel. Toute instruction d'appel doit être associée à une instruction RET.

Il existe deux formes principales de sauts, conditionnelle et inconditionnelle. Ces deux formes peuvent être encore divisées en sauts absolus, c'est à dire, JP NXT ou saut relatif :JR NXT.

Deux types d'appel (CALL) existent, conditionnel et inconditionnel. Toute instruction d'appel doit être associée à une instruction RET.

Dans un contexte, vous comprendrez la liste suivante d'instructions, même si vous ne comprenez pas tout a fait les conditions de flag.

```
JP    nn
JP    cc,nn
JR    e
JR    Z,e
JR    NZ,e
nn    est une adresse absolue ou une étiquette
ce    est l'un des opérandes indiquant quel  flag doit être testé
```

est une adresse absolue, sauf qu'elle est limitée à -126 ou +129 octets à partir de l'adresse en cours. Une étiquette peut également être utilisée.

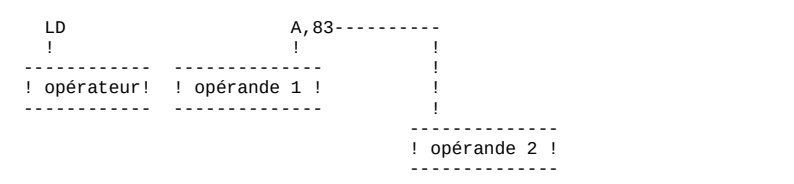
```
ce       condition
NZ       Non-Zéro
Z        Zéro
NC       Pas de retenue
C        Retenue
PO       Parité impaire
PE       Parité paire
P        Positif
M        Négatif
```

Et aussi :

```
INC r
DEC r
```

Où r est un registre de 8 bits parmi ceux-ci:
B,C,D,E,H,L,A.

On peut mieux définir opérande et opérateur en utilisant un exemple:



CHAPITRE 3

REGISTRES DOUBLES ET MODES D'ADRESSAGE

Jusque là nous avons pu seulement charger dans un registre un nombre a 8 bits. Le Z80 permet à certains registres d'être combinés en paires, permettant ainsi la mémorisation et la manipulation de nombres a 16 bits.

Les registres suivants peuvent être couplés.

```
B et C
D et E
H et L
```

Nous avons déjà étudié les instructions qui nous permettent de charger une donnée de 8 bits dans des registres individuels. L'instruction qui permet le chargement de 16 bits est:

LD dd-nn Charge dans le registre double dd la donnée de 16 bits nn.

Ici ,dd est un des registres suivants: BC, DE, HL, ou SP. 'nn' est un nombre à 16 bits.

NOTE: Le registre SP est un registre spécialisé, appelé pointeur de pile. Nous y reviendrons ultérieurement.

Lorsque l'on charge un registre double de cette manière, on le charge immédiatement avec la valeur de donnée Indiquée, d'où le terme employé pour ce type d'adressage, MODE IMMEDIAT.

Tous les programmes ont jusqu'ici mémorisé des donnée dans des registres. Tout cela va très bien tant que l'on n'a pas beaucoup de données à traiter. Mais dans des situations réelles, nous aurons besoin d'une mémorisation de données plus grande que les sept registres a 8 bits les trois registres à 16 bits qui sont disponibles.

Nous avons donc besoin maintenant d'instructions pour charger dans la mémoire des données, et pour lire le contenu d'une case mémoire. Le Z80 permet aux contenus des registres d'être stockés en mémoire, et aux contenus des cases mémoire d'être recopiés dans des registres.

Cette forme d'adressage est appelée MODE DIRECT, parce qu'elle utilise les contenus des cases mémoire et des registres directement, c'est à dire sans en faire quelque chose auparavant.

LD (nn),dd Charge dans la case mémoire nn le contenu du registre double dd.

Voici un exemple d'instruction :

```
LD (200),BC
```

Cela recopiera le contenu du registre double BC dans la case mémoire 200. Remarquez que la case mémoire doit être placée entre parenthèses.

L'instruction opposée, pour recopier le contenu de la case mémoire dans un registre double est :

LD dd,(nn) Charge dans le registre double dd le contenu de la case mémoire nn.

Voici un exemple d'instruction :

```
LD BC,(200)
```

Qu'est-ce qui va se passer? Cela va charger le contenu de la case mémoire 200 dans BC.

Bien, utilisons maintenant ces instructions.

Jusqu'ici, les caractères ont été affichés sur l'écran avec une instruction CALL. Ce n'est pas la seule routine Incorporée disponible. La mémoire interne (ROM) contient aussi des routines de graphisme, qui entre autre chose nous permettent de dessiner des lignes sur l'écran. Pour le moment nous n'approfondirons pas comment cela est fait, mais nous nous en tiendrons à leur utilisation. Alors qu'en BASIC, pour dessiner une ligne allant Jusqu'au point 400,200 (en supposant que le curseur graphique soit situé à 0,0).
Il faudrait taper la commande suivante:

```
DRAW 100,200
```

Si vous l'essayez, assurez-vous de remettre le curseur graphique à 0,0 avec la commande:

```
PLOT 0,0
```

En utilisant la routine graphique commençant en case mémoire 48118, nous pouvons aussi tracer une ligne Jusqu'à 400,200 en utilisant un code machine. En BASIC, les coordonnées sont tapées après l'instruction DRAW. Quand on utilise des routines code-machine, les données (ici, les coordonnées X,Y) sont transmises à travers les registres. Dans le registre DE est chargée la coordonnée X, dans HL est chargée la coordonnée Y.

Résumons les éléments nécessaires à l'utilisation de cette routine :

Adresse de départ 48118

Paramètres X,Y

X mémorisé dans le registre double DE
Y mémorisé dans le registre double HL

Bien, traçons donc sur l'écran une ligne jusqu'au point 400,200.

Voici le programme:

PROGRAMME 3.1

```
ENT
LD DE,400
LD (35000),DE
LD HL,200
LD (35002),HL
LD DE,(35000)
LD HL,(35002)
CALL 48118
RET
```

Tapez-le et lancez-le ...

Observons le programme. La Première chose à noter est que le contenu de DE est mémorisé en case mémoire 35000, mais le contenu de HL l'est en 35002. Pourquoi ne pas mémoriser le contenu de HL en 35001? Parce que toute case mémoire ne peut mémoriser qu'un nombre à 8 bits, alors que nous mémorisons le contenu d'un registre à 16 bits donc 16 bits. Ce qui se passe, c'est que le nombre à 16 bits est partagé en deux nombres à 8 bits, qui sont alors mémorisés simultanément. Par conséquent, la mémorisation d'un nombre à 16 bits demande deux cases mémoire.

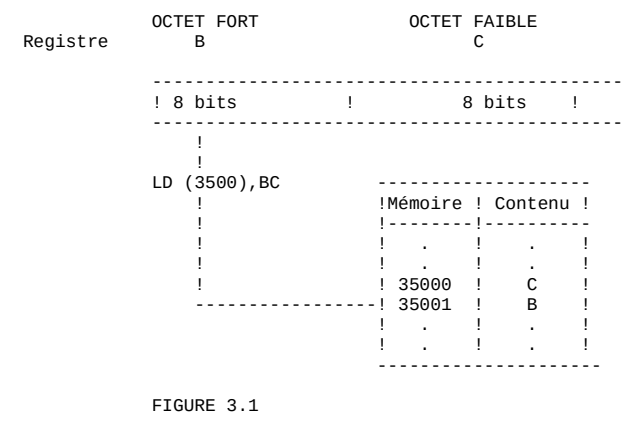


FIGURE 3.1

L'octet faible 'Least Significant Byte' (LSB) est mémorisé en premier, suivi immédiatement de l'octet fort, 'Most Significant Byte' (MSB).

EXERCICE 3.1

Ecrivez l'ordre dans lequel les registres seraient mémorisés si le sous-programme suivant était exécuté.

```
LD (200),DE
LD (202),HL
```

Reportez-vous au chapitre solutions pour la réponse.
Lorsqu'ils sont couplés, les registres C, E et L sont appelés octets faibles (Least Significant Bytes), et par conséquent les registres B, D et H sont appelés octets forts (Most Significant Bytes).

Maintenant un autre exercice:

EXERCICE 3.2

Ecrivez un programme pour charger 100 dans la case mémoire 35000 et 400 dans la case 35002 et ensuite tracez une ligne vers ces points.
Notez que le curseur graphique devra être remis à 0,0. Sautez au BASIC en sélectionnant X pour le faire avec PLOT 0,0 ou pour les plus audacieux: l'Amstrad contient une routine machine pour placer un point, nous pourrions donc incorporer celle-ci au programme. Cela éviterait d'utiliser le BASIC pour restaurer le curseur graphique.

Le format de cet appel est :

Adresse d'appel 48106.

DE = coordonnée X
HL = coordonnée Y

(Pratiquement identique à la routine de traçage de ligne.)

Donc pour placer un point a 0,0 il nous faut charger 0 dans DE et dans HL, puis appeler 48106.

Voir la réponse au chapitre solutions

EXERCICE 3.3

Ecrivez un programme qui joindra par des lignes les points suivants :

```
---------------
!   X  !   Y  !
---------------
! 200  ! 300  !
! 400  ! 200  !
! 0    !  0   !
---------------
```

N.B. Commencez avec 200/300, rappelez-vous que vous chargez dans DE la coordonnée X et dans HL la coordonnée Y.

Les solutions possibles sont données au chapitre solutions.

Le microprocesseur Z80 possède aussi d'autres modes d'adressage. Jusqu'ici, le mode Immédiat pour des données de 8 bits et de 16 bits, ainsi que le mode direct pour des données de 16 bits, ont été utilisés. Il en existe un autre : le mode INDIRECT.

MODE D'ADRESSAGE INDIRECT

Dans ce mode d'adressage, le contenu d'un registre est utilisé pour pointer sur une case mémoire. Ce mode est très pratique, puisque l'on peut charger dans un registre de 16 bits la case mémoire de départ de nos données. On peut également lire les données, puis incrémenter (soit ajouter 1) le registre double et accéder à la prochaine donnée.

La famille d'instructions est la suivante :

LD r,(HL) Charge dans le registre r le contenu de la case mémoire pointée par HL. (Cela signifie en fait que nous chargeons dans HL l'adresse mémoire).

LD (HL),r Charge dans la case mémoire pointée par HL le contenu du registre r.

LD A,(BC) Charge dans l'accumulateur le contenu de la case mémoire pointée par BC.

LD A,(DE) Charge dans l'accumulateur le contenu de la case mémoire pointée par DE.

LD (BC),A Charge dans la case mémoire pointée par DE le contenu de l'accumulateur.

La table ci-dessus peut sembler redoutable au premier abord, mais heureusement elle ne l'est pas; nous expliquerons tout cela au fur et à mesure que les Instructions seront utilisées dans des exemples.

Maintenant, pour Illustrer l'utilisation de quelques unes de ces commandes, nous allons stocker la représentation ASCII de trois A en mémoire, puis les afficher sur l'écran en utilisant le registre HL comme un 'pointeur'.

Pour simplifier la mémorisation des trois A, nous utiliserons BC comme un pointeur variable. Ainsi, si l'accumulateur contient le code de A (65), nous pouvons, en utilisant LD (BC),A et INC BC, charger trois A dans les adresses mémoires consécutives, en commençant à l'adresse contenue dans BC, en incrémentant BC trois fois.

Voici le programme:

PROGRAMME 3.2 (ne l'assemblez pas pour le moment)

```
ENT

LD BC,35000     BC=début des données en mémoire
LD A,65         A=65, le code ASCII de 'A'
LD (BC),A       Mémorise le premier A
INC BC          Incrémente le pointeur
LD (BC),A       Mémorise le second A
INC BC          Incrémente le pointeur
LD (BC),A       Mémorise le troisième A
```

Maintenant, pour écrire le programme afin de lire la donnée, qui commence à 35000, et la sortir sur l'écran.

```
LD HL,35000     HL=adresse départ
LD A,(HL)       A=contenu de la case mémoire adressée par HL.
CALL 47962      Affiche le contenu de A sur l'écran
INC HL          Incrémente le pointeur
LD A, (HL)      Recharge A
CALL 47962      Affiche le second caractère
INC HL          Incrémente le pointeur
LD A, (HL)      Recharge A
CALL 47962      Affiche le troisième caractère
RET             Retour
```

Assemblez et lancez ce programme maintenant. Ce programme atteint son but mais n'est pas très efficace. Voyons comment nous pouvons l'améliorer.

Observons la première partie.

Nous écrivons trois fois les commandes suivantes pour lire les données:

```
LD (BC),A
INC BC
```

Pourquoi ne pas mettre une boucle? Si nous chargeons 3 dans un registre inutilisé, disons E, alors chaque fois que nous chargeons une case mémoire, nous décrémentons E et le testons pour voir si le résultat est zéro. Si oui, nous terminons la boucle, si non, nous revenons au début de la boucle par un saut. Cette méthode accomplit la même tâche de façon plus élégante et plus efficace.

Voici la première partie de programme modifiée:

PROGRAMME 3,3 (Ne l'assemblez pas pour l'instant)

```
      ENT
      LD BC, 35000
      LD A,65
      LD E,3    Charge le compteur de programme
NXT:  LD (BC),A
      INC BC
      DEC E     Décrémente le compte
      JR NZ,NXT:  Si pas zéro saut a NXT
```

Notez l'utilisation de l'étiquette NXT pour simplifier le calcul de l'adresse de saut. On gagne peu de temps en tapant ce programme, mais imaginez que vous chargiez 200 A avec la méthode précédente !

La même technique peut être appliquée à la deuxième partie.

EXERCICE 3.4

Ajoutez une deuxième partie au programme en utilisant une boucle pour afficher les trois A, puis assemblez-le et lancez-le.

Une réponse possible est donnée dans le chapitre solutions.

EXERCICE 3.5

Ecrivez un programme pour stocker l'alphabet en mémoire, puis extrayez-le de la mémoire pour le sortir sur l'écran.

Voir le chapitre solutions pour les réponses possibles.

Récapitulons brièvement les modes d'adressages utilisés jusqu'ici.

Registre-registre	Immédiat
LD B,A	LD A, 83 ou LD BC,300
Direct	Indirect
LD A, (200)	LD B, (HL)

Vous trouverez une liste complète des mnémoniques du Z80 dans les appendices. La plupart des groupes de chargement de 8 et 16 bits devraient être compréhensibles maintenant.

ADRESSAGE INDEXE

Dans le programme 3.3 nous avons utilisé un registre double pour accéder à un bloc de données stocké séquentiellement. Alors que dans certaines applications cette méthode est très efficace, le Z80 offre une alternative, En plus des registres déjà présentés, le Z80 contient deux registres d'index de 16 bits, IX et IY. Si vous cherchez le mot 'mémoire' dans un dictionnaire, vous commencerez par trouver le début des M, puis vous chercherez le mot 'mémoire'. En d'autres termes vous avez pris le début des M pour base et à partir de là vous avez commencé votre recherche. Ceci est tout à fait analogue à l'Adressage Indexé. Dans le registre IX ou IY est chargée une adresse de base. Ensuite, pour accéder à une information, un décalage est ajouté à cette base. Le format général de la partie indexée d'une instruction est représenté ci-dessous:

```
      (IX+d)
      !  !
      !  !------ décalage
      !
      ! Registre d'index (base), pourrait aussi être IY.
```

Ici d est un nombre de -127 à +128. Voici un exemple d'instruction qui utilise le registre d'index IX:

```
LD A,(IX+3)
```

Qu'est-ce qui va se passer?
Supposons que IX contienne 200. Quelle serait la valeur chargée dans A?

```
                !-------------------------!
                !   Adresse  !  Contenu   !
                !-------------------------!
LD A,(IX+3)     !  200       !      1     !
                !  201       !      2     !
                !  202       !      3     !
LD A,(200+3) --> !  203       !      4     !
                !  203       !      5     !
                !-------------------------!
```

FIGURE 3.2

L'accumulateur contiendra donc 4 après cette instruction. Quel décalage aurait-on eu à utiliser pour charger 3 dans A ? Réponse : (IX+2)

Au lieu d'utiliser le registre IX comme base, nous aurions pu utiliser IY avec l'instruction :

```
LD A,(IY+3)
```

Revoyons maintenant les instructions de chargement qui utilisent soit IX soit IY.

INSTRUCTIONS D'ADRESSAGE INDEXE

Il existe une méthode rapide de décrire l'opération d'une instruction, que l'on appelle son opération symbolique. Voici un exemple :

Instruction Opération symbolique

LD r,(IX+d) r <-- (IX+d)

Cela charge dans le registre r le contenu de la case mémoire pointée par l'addition de IX et d. Les appendices contiennent une table complète des Instructions Z80 ainsi que des opérations symboliques.

Voici les instructions de chargement indexé :

Instructions	Opération symbolique
LD r, (IX+d)	r <-- (IX+d)
LD r, UY+d)	r <-- (UY+d)
LD (IX+d),r	(IX+d) <-- r
LD (IY+d),r	(UY+d) <-- r
LD (IX+d),n	(IX+d) <-- n
LD (IY+d),n	(lY+d) <-- n

Maintenant voyons comment on peut charger les registres IX ou IY. Regardez la table du groupe de chargement de 16 bits, et essayez de repérer les instructions à utiliser.

Les voici; le format devrait vous sembler familier.

Instructions	Opération symbolique
LD IX, nn	IX <-- nn
LD IY,nn	IY <-- nn
LD IX, (nn)	IX <-- (nn)
LD IY, (nn)	IY <-- (nn)
LD (nn),IX	(nn) <-- IX
LD (nn),IY	(nn) <-- IY

Remarque : Il n'est pas possible de calculer la valeur de 'd', il faut absolument qu'elle soit donnée, c'est à dire comme une valeur immédiate.

Pour illustrer l'utilisation des registres d'index, le Programme 3.2 a été réécrit.

Remarque: Dorénavant les instructions ENT seront omises de tous les programmes nous considérerons que vous les insérerez automatiquement en première ligne.

Voici le programme :

PROGRAMME 3.4

```
LD IX,35000       IX=base
LD A,65
LD (IX+0),A       Premier 'A' stocké
LD (IX+1),A       Deuxième 'A' stocké
LD (IX+2),A       Troisième 'A' stocké
```

La seconde partie, qui affiche le contenu de la mémoire sur l'écran, se présente ainsi :

```
LD A,(IX+O)
CALL 47962 Afficher premier caractère
LD A,(IX+1)
CALL 47962 Afficher deuxième caractère
LD A,(IX+2)
CALL 47962 Afficher troisième caractère
RET
```

Maintenant essayez-le.

Parce que le Z80 doit additionner le décalage à la base pour des instructions indexées, il demande plus de temps pour l'exécution qu'une instruction immédiate ou indirecte. D'autre part, comme 'd' doit être donné comme une valeur absolue, l'adressage indexé n'est pas très bon en boucle, puisque 'd' ne peut pas être incrémenté. Cependant, l'avantage principal de IX et IY est leur capacité d'accéder a une donnée, stockée en mémoire, relativement à des adresses mémoire connues (c'est-à-dire pour accéder aux tables de donnée), à l'endroit où diverses informations sont stockées en utilisant des décalages connus.

EXERCICE 3.6

Ecrivez un programme en utilisant IX pour stocker cinq lettres, par exemple un nom. Ensuite affichez la troisième et la quatrième lettres sur l'écran.

La table suivante vous aidera:

Adresse mémoire	Lettre	Code de la lettre
35000	S	83
35001	U	85
35002	S	83
35003	A	65
35004	N	78

Une solution possible est donnée dans le chapitre solutions.

Parfait, fin d'un autre chapitre! Tout ce qui reste maintenant c'est un résumé de ce que vous avez appris dans ce chapitre. Si vous vous sentez à la hauteur, écrivez vos propres programmes. Pourquoi ne pas afficher un caractère, l'effacer par l'affichage d'un espace, et l'afficher a une autre position. Cela pourrait être la base d'un jeu !

RESUME

Vous devriez maintenant comprendre les termes suivants :

1. Registres doubles

2. Modes d'adressage

   a. Registre-Registre
   b. Immédiat
   c. Direct
   d. Indirect
   e. Indexé

3. Octet fort et octet faible (M.S.B. et L.S.B.)

4. Opération Symbolique

5. Bien que les commandes suivantes n'aient pas été mentionnées de manière spécifique, vous les comprendrez sûrement (ss est un registre double).

```
INC ss
DEC ss
```

6. Vous devriez désormais pouvoir comprendre ces trois instructions qui ont été mentionnées brièvement dans le chapitre 2.

```
DEC (HL)
DEC (IX+d)
DEC (IY+d)
```

où ss est un des registres suivants (ignorez SP pour l'instant)

BC,DE,HL,SP

d est un nombre de -126 à +129.

CHAPITRE 4

OPERATIONS ARITHMETIQUES

La plus part des situations réelles demandent la manipulation de nombres. Jusqu'ici, nous pouvons seulement incrémenter et décrémenter le contenu des registres ou des cases mémoire. Le Z80 nous fournit quelques Instructions arithmétiques supplémentaires, oui, bien que peu étendues, forment les bases d'opérations plus puissantes.

Ce chapitre explique ces instructions. Si vous n'êtes pas familiarisé avec les représentations binaire ou hexadécimale des nombres, reportez-vous à l'appendice 5 et travaillez-le.

Les concepts binaire et hexadécimal vous sont maintenant familiers. Vous verrez bientôt pourquoi nous utilisons les nombres hexadécimaux, cela évite au moins du travail de frappe !

Nous pouvons classer les instructions arithmétiques en deux groupes, le groupe 8 bits et le groupe 16 bits. Dans ce chapitre nous nous occuperons principalement du groupe 8 bits, puis nous verrons ultérieurement le groupe 16 bits.

Le procédé arithmétique le plus commun est l'addition de deux nombres. Pour additionner entre eux deux nombres à 8 bits, l'instruction suivante est utilisée:

```
ADD A,n     Ajouter (ADD) n au contenu de l'accumulateur, en remplaçant le contenu de
            l'accumulateur par le résultat.
```

Pour illustrer cette commande nous ajouterons 65 à 20, ce qui donne 85. Quand le résultat sera affiché sur l'écran, nous verrons un U. (Le caractère correspondant à la valeur ASCII de 85).

Essayez le programme suivant

PROGRAMME 4.1

```
ENT

LD A,65          Charger 65 dans A
ADD A,20         Ajouter 20 à A
CALL 47962       Afficher le résultat sur l'écran
RET
```

Vous voyez qu'un U apparaît sur l'écran.

L'assembleur permet aussi la représentation hexadécimale des nombres. Pour permettre à l'assembleur de les reconnaître, les nombres hexadécimaux sont précédés d'un '&'.

Réécrivez le programme 4.1 en utilisant la représentation hexadécimale pour 47962, 65 et 20 :

PROGRAMME 4.2

```
ENT
LD A, &41
ADD A,&14
CALL &BB5A
RET
```

Vérifiez que ce programme est bien le même que le programme 4.1 lorsque vous le lancez.

EXERCICE 4.1

Ecrivez un programme qui ajoutera 200 a 48 puis affichera le résultat sur l'écran.

Une réponse possible est donnée dans le chapitre solutions.

EXERCICE 4.2

Ecrivez un programme qui ajoutera &41 à &10. Puis affichera le résultat sur l'écran. Qu'est-ce qui sera affiché ?

Une réponse possible est donnée dans le chapitre solutions.

Jusque là toutes nos réponses ont été inférieures à 255, le nombre maximum de 8 bits qu'un registre puisse contenir. Que pensez-vous qu'il arrive si 150 et 171 sont additionnés? Essayez le programme suivant. N'oubliez pas ENT.

PROGRAMME 4.3

```
LD A,150         Charger 150 dans A
ADD A, 171       Ajouter 171  à A
CALL 17962       Afficher le résultat sur l'écran
RET
```

Ce qui s'est passé, c'est que l'accumulateur a débordé. Il a atteint 255 puis lorsque 1 de plus a été ajouté, il est revenu à zéro et a recommencé, atteignant 65; d'où le A sur l'écran.
Ce que l'on ne peut pas voir d'après le programme, ce sont les registres de flags. Si cela était visible, nous aurions pu voir que lorsque l'accumulateur a débordé, le bit de retenue a été mis (1). Ceci peut âtre utilisé pour additionner deux nombres dont la somme est supérieure a 255. Nous i1 lustrerons d'abord ce procédé sur le papier, puis nous écrirons un programme qui accomplira la même tâche en langage assembleur.

Problème : Ajoutez 1157 à lui même, soit 1157 + 1157 = ?

Convertissez 1157 en hexadécimal.

```
1157 : 1096 = 0 reste 1157
1157 : 256  = 1 reste 133
133  : 16   = 8 reste 5
5    :  1   =  5 reste 0
```

Donc : 1157 = &0485

&0485 exprimé en binaire est un nombre a 16 bits. Il faut donc qu'il soit partagé en deux pour permettre l'utilisation d'instructions arithmétiques a 8 bits. C'est facile en hexadécimal. (Voila pourquoi on utilise la représentation en hexadécimal!)

```
-----  ------  Octet fort = &04
! 04 ! ! 85 !      !
-----  ------      !
  !      !----------
  !-------------------Octet faible = &85
```

ETAPE 2

Pour additionner entre eux les deux &0485 il nous faut d'abord additionner les octets faibles puis les octets forts, en tenant compte de toute retenue générée par l'addition des octets faibles.

```
           85          85
         +85         +85
           --          ----

+ retenue   OA  +retenue 6 10 Décimal
                +retenue 0  A Hexadécimal
```

Le résultat est donc &0A + une retenue

ETAPE 3

Additionnez les octets forts entre eux en tenant compte du bit de retenue.

```
 04
+04
 --
 08
 --
```

Maintenant ajoutez le bit de retenue :

```
  01 <-- Bit de retenue
+ 08
  --
  09
  --
```

ETAPE 4

Recombinez le nouvel octet faible et le nouvel octet fort.

1157 + 1157 = &090A

Puis vérifiez vous-même que &090A égale 2314.

La caractéristique principale à noter est que dans tout travail en DOUBLE PRECISION, c'est à dire avec l'utilisation de nombres à 16 bits, on agit d'abord sur l'octet faible, de façon à tenir compte de la retenue (s'il y en a une) .

Avant de pouvoir écrire le programme pour faire cette addition, il nous faut connaître quelques nouvelles instructions :

AND A ET logique de l'accumulateur.

Un seul effet de cette opération doit être observé pour le moment, c'est la remise a zéro du flag de retenue. Pourquoi est-ce nécessaire ? Réponse: Si le flag de retenue était mis de façon non intentionnelle, le résultat après l'addition serait incorrect a cause de l'inclusion de la retenue incorrecte dans l'addition.

Une instruction qui additionne deux registres plus le contenu du flag de retenue est à présent requise.

ADC A,s Ajoute le contenu du registre s plus le flag de retenue au contenu de l'Accumulateur. Le résultat est stocké dans l'accumulateur.

Remarquez que l'instruction ADC utilise un registre comme l'un de ses opérandes. Une instruction similaire à ADD A,n existe, il faut utiliser un registre à la place de la donnée immédiate n.

ADD A,s Ajouter le contenu du registre s au contenu de l'Accumulateur. Le résultat est stocké dans l'accumulateur.

Maintenant, le programme.

PROGRAMME 4,4 (ne l'assemblez pas encore)

```
LD C,&85          Charge dans C le 1er octet faible
LD A,&85          Charge dans A le 2ème octet faible
AND A             Annule le flag de retenue
ADD A,C           A = octet faible + octet faible
LD (87000),A      Sauvegarde le nouvel octet faible
LD C,804          Charge dans C le 1er octet fort
LD A, 804         Charge dans A le 2ème octet fort
ADC A,C           A = octet fort + octet fort + retenue
LD (87001),A      Sauvegarde le nouvel octet fort
```

Bon, ce programme additionne deux nombres. Mais comment vérifier le résultat ?
Ce qu'il nous faut maintenant, c'est un programme pour afficher la réponse dans un format reconnaissable. Pourquoi les contenus des deux cases mémoire ne peuvent-ils être affichés sur l'écran ? Rien de reconnaissable n'apparaîtrait, puisque les codes ASCII &09 et &0A ne représentent pas des caractères mais des codes de contrôle. Un décalage doit être ajouté aux deux réponses pour les amener dans le cadre de l'alphabet ASCII (65-122).

Comme le code de A est 65, cela paraît un décalage raisonnable à utiliser. Donc nous ajoutons les lignes suivantes au programme initial là où nous l'avons laissé:

PROGRAMME 4.4 (a)

```
LD C,65
LD A,(&7001)
ADD A,C
CALL &BB5A
LD A,(&7000)
ADD A,C
CALL &BB5A
RET
```

A présent assemblez et lancez tout le programme. Vous verrez un J et un K sur l'écran (correspondant à &09 + 65 et 804 + 65).

EXERCICE 4.3

Ecrivez un programme pour additionner 250 à 600, en hexadécimal, en ajoutant 65 aux octets fort et faible. Puis affichez le résultat sur l'écran.

Une réponse possible est donnée dans le chapitre solutions.

SOUSTRACTION

SUB s Soustrait (SUBtracts) le contenu du registre s de l'accumulateur. Le résultat est stocké dans l'accumulateur.

SBC A,s Soustrait le contenu du registre s plus le flag de retenue, de l'Accumulateur. Le résultat est stocké dans l'accumulateur.

L'opération de soustraction de deux nombres à 8 bits est laissé pour l'exercice 4.4. Essayez !

EXERCICE 4.4

Ecrivez un programme qui soustraira 9 de 233. Affichez le résultat sur l'écran. Remarquez qu'il n'est pas nécessaire d'ajouter un décalage de 65 pour le résultat. Pourquoi ?

Une réponse possible est donnée dans le chapitre solutions.

EXERCICE 4.5

Ecrivez un programme qui calculera le résultat de la somme suivante :

(97 + 126) - 153 = ?

Une réponse possible est donnée dans le chapitre solutions.

Comme nous l'avons vu précédemment, lors de l'addition de deux nombre: donnant un résultat supérieur à 255, un dépassement se produit. Ce dépassement fait que le flag de retenue est mis.

La situation opposée se produit lorsque l'on essaie de soustraire un grand nombre d'un petit nombre. Par exemple, essayez cela :

25 - 17

Il n'est pas possible de soustraire directement 7 de 5, alors on emprunte une unité de la colonne suivante. La première étape de la soustraction devient alors :

```
   5 + emprunt=     15
 - 7               - 7
  --                --
   ?                 8  Rép1=8
```

Puisque nous avons emprunté, il nous faut rembourser : soit soustraire 1 de la colonne suivante,

Nous complétons la soustraction.

```
      2 - emprunt =       1
    - 1                 - 1
      --                  --
      ?                   0  Rép2=0
      --                  --
```

Nous combinons les deux réponses (Rép1 et Rép2)

      0 + 8 = 8

Donc:

      27-17 = 8

En utilisant le même procédé, nous pouvons accomplir cette soustraction avec les instructions de soustraction du Z80.

Problème : Soustrayez 2000 de 2224.

ETAPE 1

Convertissez les deux nombres en hexadécimal.

```
2000 : 4096   = 0          reste  2000
2000 : 256    = 7          reste  208
208  : 16     = D          reste  0
0    :  1     = 0

2224 : 4096   = 0          reste 2224
2224 : 256    = 8          reste 176
176  : 16     = B          reste 0
0    : 1      = 0
```

2000 = &07DO
2224 = &08B0

ETAPE 2

Soustrayez les deux octets faibles

```
 B0 + emprunt =   1B0
-D0                D0
 --                --
 ??             -- E0
 --             !  --
                !   !-- 0=0-0
                !-E=1B-D
```

ETAPE 3

Puis les octets forts

```
 08       08
-07     - 07 + emprunt = 08
 --       --
 ??       00
 --       --
```

Combinez les deux résultats :

 2224-2000 = &00EO

Lorsque l'on fait ce type de soustraction, le flag de retenue agit un flag d'emprunt, qui est mis lors de l'apparition d'un emprunt. Convertissons maintenant cette soustraction en instructions Z80.

PROGRAMME 4.5

```
LD C,&D0          C= 1er octet faible
LD A,&B0          A= 2eme octet faible Annule
AND A             Annule le flag de retenue
SUB C             A= nouvel octet faible
LD (&7000),A      Sauvegarde octet faible
LD C,&07          C= 1er octet fort
LD A,&08          A= 2eme octet fort
SBC A,C           A= A-C-retenue
LD (&7001),A      Sauvegarde octet fort
LD A,(&7000)      Rappelle octet faible
CALL &BB5A        Affiche résultat
RET
```

Points à noter : premièrement, comme l'octet fort de la réponse est connu comme étant zéro, ce n'est pas la peine de l'afficher. Deuxièmement, ce n'est pas la peine d'ajouter un décalage à la réponse puisqu'elle est dans le cadre affichable des caractères ASCII (65-255).

EXERCICE 4.6

Ecrivez un programme pour faire une soustraction à 16 bits en utilisant les instructions suivantes (ss est un registre double -ici : BC ou DE)

SBC HL,ss

Par exemple : 4248 - 4008 = ?

(Indication = c'est plus facile que le programme 4.5) Réponse dans le chapitre solutions

EXERCICE 4.7

Stockez deux nombres en mémoire, puis ajoutez a ceux-ci le contenu de l'accumulateur. Remplacez les anciennes valeurs par les résultats nouvellement calculés.

Les tables suivantes vous aideront.

Case mémoire	Contenu
35000	10
35001	20

Accumulateur = 65

Après exécution du programme,

Case mémoire	Contenu
35000	75
35001	85

Utilisez l'instruction ADD A,(HL). Vérifiez que les résultats sont ceux espérés en affichant le contenu de la case mémoire sur l'écran.

Réponse au chapitre solutions.

EXERCICE 4.8

Tracez sur l'écran une ligne jusqu'au point 100,50 en utilisant la routine de tracement de ligne. Puis ajoutez 75 à chaque coordonnée et tracez une autre ligne jusqu'à ce point.

La routine de tracement de ligne est entièrement détaillée dans l'annexe approprié, mais nous vous indiquons brièvement :

Appeler adresse 48118

Paramètres X,Y

X passé en DE
Y passé en KL

Une réponse possible est donnée au chapitre solutions.

Hé bien, tout cela apporte une conclusion aux principes de base de l'addition et de la soustraction. Bien que non mentionnées explicitement, les instructions suivantes devraient pouvoir être comprises aisément :

Instructions	Opération symbolique
ADD HL,SS	HL <-- HL+SS
ADC HL,SS	HL <-- HL+ss+retenue
SBC HL,SS	HL <-- HL-ss-retenue
ADD IX,pp	IX <-- IX+pp
ADD IY,rr	IY <-- IY+rr

Ici ss est BC, DE, HL ou SP,
pp est BC, DE, IX ou SP,
rr est BC, DE, IY ou SP

D'après l'opération symbolique, vous devriez pouvoir comprendre ce que chaque instruction accomplit. Il est possible d'additionner ou de soustraire des nombres à 32 bits en utilisant les instructions ci-dessus et le flag de retenue, en utilisant la même méthode que pour l'arithmétique à 16 bits réalisée avec des instructions à 8 bits.

LES INSTRUCTIONS DE FLAG DE RETENUE

Avant toute opération arithmétique, le flag de retenue doit être annulé. Cela se fait en utilisant l'instruction AND A. Le flag de retenue peut également être annulé en utilisant les deux instructions suivantes :

SCF Met le flag de retenue
CCF Inverse le flag de retenue

Pour annuler le flag de retenue avec ces instructions, le flag de retenue est d'abord mis en utilisant l'instruction SCF, puis il est inversé par l'instruction CCF. Lorsqu'un nombre binaire est inversé, un zéro est remplacé par un un; de la même manière un un est remplacé par un zéro. Le flag de retenue lorsqu'il est mis, est égal à un, et donc après inversion, il sera égal a zéro. La raison de l'utilisation de AND A est que cela demande moitié moins de temps que SCF et CCF.

RESUME

Les principes de bases de l'addition et de la soustraction devraient maintenant être assimilés, ainsi que l'utilisation du flag de retenue.

CHAPITRE 5

DECIMAL CODé EN BINAIRE ET OPERATEURS LOGIQUES

En plus des nombres représentés en notation binaire, hexadécimale et décimale, une autre représentation existe. On lui donne ce nom assez impressionnant de décimal codé en binaire. Alors qu'en binaire un nombre (0-255) est normalement stocké par octet, en BCD (Binary Coded Décimal) l'octet est partagé en deux. Le nom donné à chaque demi-octet de 4 bits est : quartet !

Par exemple, 7564 stocké en BCD demande 2 octets (4 chiffres a coder a raison de 2 chiffres par octet).

```
             7   5
           -----------
Octet 1    !0111!0101!
           -----------

             6   4
           -----------
Octet 2    !0110!0100!
           -----------
```

Notez que l'octet 1 n'est pas nécessairement stocké en premier en mémoire. Sa position dépend du format de stockage utilisé dans le programme.

Vous avez peut être remarqué que 4 bits sont utilisés pour stocker chaque nombre (0-9) alors que 3 bits uniquement pourraient les stocker. Le pourquoi de l'utilisation de 4 bits est que BCD existe depuis plus longtemps que les micros. Il a d'abord été utilisé sur des unités centrales où il était plus efficace de stocker des chiffres BCD en 4 bits. BCD est très utilisé dans les programmes de comptabilité et est également très utile pour transmettre des informations à certaines formes de dispositifs d'affichage (par exemple les affichages à 7 segments vus dans les calculatrices et les montres a affichage digital).

Observons maintenant comment deux nombres BCD s'additionnent.

```
Décimal   BCD

   8     1000
  +2    +0010
  --    -----
  10     1010
```

C'est bien, mais 1010 c'est 10 en décimal et par conséquent trop grand pour être stocké dans un quartet BCD qui ne peut stocker que des nombres de 0 à 9. Quelques ajustements sont nécessaires pour des réponses supérieures à neuf. En ajoutant 6 à tout résultat supérieur a 9, on obtient le résultat juste :

Ainsi :

```
Premier quartet            Deuxième quartet
    0000                     1010
  + 0000                      0110 : - 6 en décimal
    ----                      -----
    0001                      0000
```

Ce qui donne la représentation correcte de 10 en BCD ; 0001 0000.

Cela deviendrait vite ennuyeux si chaque fois que nous exécutions des opérations arithmétiques BCD, il nous fallait vérifier la validité du résultat et le corriger si nécessaire. Heureusement le Z80 contient une instruction qui le fait pour nous :

DAA Décimal Adjust the Accumulator (ajustage décimal de l'accumulateur)

Revenons maintenant à la programmation

Le programme 5.1 additionne 8 et 2 pour donner la réponse en BCD.

```
PROGRAMME 5.1
LD A,8
ADD A,2
DAA
ADD A,65
CALL 47962
RET
```

Notez l'utilisation du décalage (65) pour amener le résultat dans la zone alphabet.

Le résultat, avant addition du décalage, sera la représentation BCD de 10 (0001 0000). En décimal cette valeur binaire correspond à 16. Et donc le caractère ayant le code ASCII 16+65 sera affiché sur l'écran (Q).

EXERCICE 5.1

Ajoutez 7 a 12 en BCD et affichez sur l'écran la réponse sous la forme d'une lettre.

Une réponse possible est donnée dans le chapitre solutions.

EXERCICE 5.2

Soustrayez &12 de &35 et convertissez la réponse en BCD puis affichez sur l'écran la réponse sous forme de lettre.

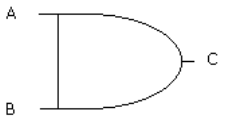
Une réponse possible est donnée dans le chapitre solutions.

Quelquefois, il est nécessaire d'extraire le quartet faible d'un octet, Cela se fait en enlevant le quartet fort. On peut le faire avec l'instruction :

AND s ET logique de l'opérande s

Ici s est A, B, C, D, E, H, L, (HL), (IX+d) ou (IY+d).

Tous ceux qui ont approché l'électronique digitale reconnaîtront le symbole suivant. C'est la représentation symbolique d'une porte logique ET.



Porte ET

FIGURE 5.1

La porte ET fonctionne ainsi si et seulement si les deux entrées A et B sont mises (1), alors la sortie C sera mise. Décrire l'opération d'une porte ET est relativement facile/ mais comme la logique des portes devient de plus en plus complexe cela finit par être de plus en plus difficile à décrire clairement. La réponse à ce problème est l'utilisation d'une table de vérité. Une table de vérité permet à une sortie de porte logique d'être Plus facilement et plus rapidement dérivée d'un jeu d'entrées donné.

Par exemple, voici la table de vérité de la porte ET.

ENTREES		SORTIE
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Table de vérité pour ET Logique

FIGURE 5.2

Notez que C est seulement mis lorsque A ET B sont mis, d'où le nom ET.

EXERCICE 5.3

En utilisant la figure 5.2 décidez si C sera mis ou non pour les entrées suivantes ;

1 A=0 B=1
2 A=1 B=1
3 A=0 B=0

Une réponse possible est donnée dans le chapitre solutions.

Lorsque le microprocesseur Z80 accomplit une instruction ET, il agit sur 8 bits en même temps.
Observez ceci :

Problème : Quel est le résultat d'une opération entre 10101101 et 00001111 ?

Solution :

```
     10101101
 AND 00001111
     --------
     00001101
```

Qu'est-ce qui s'est passé ?

Le quartet fort a été 'masqué', c'est à dire que tous les bits de ce quartet ont été mis à zéro alors que ceux du quartet faible n'ont pas été modifiés.

Ce procédé est très puissant dans le sens où il nous permet d'extraire toute portion d'un octet en utilisant un masque approprié. Par exemple, pour masquer le quartet faible de 10111011 nous ferions l'opération ET avec 11110000.

```
    1010 1101
AND 1111 0000
    ---- ----
    1101 0000
    ---- ----
```

Généralement, toute position de bit qui doit rester intacte, est ANDée avec un, le reste avec des zéros. Par exemple, si le résultat de l'opération AND d'un nombre de huit bits avec 00000011 est 2, alors nous savons que le nombre, quel qu'il soit est divisible par 2.

EXERCICE 5.4

Quel masque faudrait-il utiliser pour arriver à 00000011 en partant de 10101011 ?

Une réponse possible est donnée dans le chapitre solutions.

EXERCICE 5.5

Quel est le résultat d'une opération AND entre 253 et 75 ?

Une réponse possible est donnée dans le chapitre solutions.

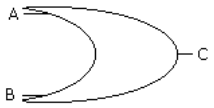
Maintenant écrivons un programme avec l'instruction AND (ET). Le programme 5.2 fait une opération logique AND avec 225 et 254, puis affiche le résultat sur l'écran.

PROGRAMME 5.2

```
LD C,225
LD A, 254
AND C
CALL 47962
RET
```

Cela reproduira une 'tête' sur l'écran, le symbole ASCII de l'Amstrad pour 224.

ET n'est pas le seul opérateur logique que possède le Z80. Etudions maintenant la porte OU (OR). Le symbole standard pour une porte OU est :



Porte OU

FIGURE 5.3

La sortie C est mise (1), si soit A soit B est mis (1) ou si A et B sont mis. Ceci est représenté dans la table de vérité suivante :

ENTREES		SORTIE
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Table de Vérité OU Logique

FIGURE 5.4

L'Instruction Z80 OR est :

OR s OU Logique de l'opérande s

Ici s est A, B, C, D, E, H, L (HL), (IX+d) OU (IY+d).

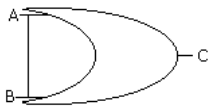
EXERCICE 5.6

Quel est le résultat des opérations logiques suivantes :

1. 1001 OR 1101 (nombres binaires)
2. 250 OR 25
3. (209 OR 20) AND 27

Les réponses se trouvent dans le chapitre solutions.

Il peut sembler étrange qu'une porte OU produise une sortie lorsque les deux entrées sont mises (1); évidemment cela peut poser un problème dans certains cas. C'est pourquoi on utilise la porte 'OU EXCLUSIF', qui résout ce problème. Le symbole utilisé pour représenter un XOR (OU EXCLUSIF) est:



PORTE XOR

FIGURE 5.5

ENTREES		SORTIE
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Table de Vérité XOR

FIGURE 5.6

L'instruction Z80 correspondante est :

XOR s OU exclusif de l'opérande s

Ici S est A, B, C, D, E, H, L, (HL), (IX+d), ou (IY+d).

EXERCICE 5.7

Quel est le résultat des opérations logiques suivantes ? Essayez les d'abord sur une feuille puis vérifiez vos

réponses en écrivant un programme.

1. 1011 XOR 1110100
2. 77 XOR 200
3. (25 OR 255) AND 200

Les réponses sont données dans le chapitre solutions.

NOMBRES SIGNES

Jusqu'ici tous les nombres utilisés étaient positifs. En de nombreuses situations les nombres négatifs sont indispensables.

Comment représenter des nombres négatifs en binaire ? La méthode utilisée permet au Z80 de traiter des nombres négatifs presque comme il traite des nombres positifs. On donne a cette méthode le nom impressionnant de "complément a deux". Avant d'étudier le complément a deux, il est auparavant nécessaire de bien comprendre le concept de complément à un !

COMPLEMENT A UN

Lorsque l'on utilise la notation complément a un, tous les nombres entiers positifs sont représentés en binaire comme d'habitude. Mais, les nombres négatifs sont représentés en remplaçant tous les '1' d'un octet par des '0', et en remplaçant tous les 0 par des 1.

Par exemple :

+9 = 1001

Remplacez maintenant tous les 1 par des 0 et tous les 0 par des 1, c'est à dire la notation complément à un de +9 :

1001 complément à un = 0110

Ainsi, en notation complément à un -9 = 0110

Mais 0110 représente aussi +6; voilà en vérité un des problèmes de la notation complément à un.

EXERCICE 5.8

Convertissez les nombres suivants sous leur forme complément à 1.

1. 1011
2. 1011101
3. 14

Réponses au chapitre solutions.

COMPLEMENT A DEUX

Comme dans le complément à 1 les nombres entiers positifs sont normalement représentés en binaire. Les nombres négatifs sont d'abord complémentés à 1 puis un est ajouté au résultat. Cela peut paraître, au premier abord, comme une façon étrange de représenter les nombres négatifs, mais ça marche. Essayons d'additionner 7 et 5 en utilisant le complément à 2 pour -5.

```
5=                0101
Complément à 1    1010
Ajouter 1       + 0001
                  ----
                  1011 = -5
```

Maintenant :

```
7             0111
+(-5)         1011
------        ----
2             0010 + retenue
--
```

Si l'on ignore le bit de retenue, la réponse est juste. Le fait Que le bit de retenue puisse être ignoré, tout en gardant un résultat juste, aide considérablement l'écriture de programmes arithmétiques simples traitant des nombres négatifs.

EXERCICE 5.9

Si l'on ignore le bit de retenue, la réponse est juste. Le fait que le bit de retenue puisse être ignoré, tout en gardant un résultat juste, aide considérablement l'écriture de programmes arithmétiques simples traitant des nombres négatifs.

EXERCICE 5.9

Calculez la réponse des problèmes suivants en utilisant la représentation complément à 2 pour les nombres négatifs.

1. -3 + 10 = ?
2. -1 + 7 = ?
3. -10 + 8 = ?

Les réponses sont données au chapitre solutions.

En utilisant 4 bits, les nombres décimaux représentés en notation complément à 2 sont les suivants :

Décimal	Binaire
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Les 16 combinaisons uniques de 1 et de 0 oui résultent de 4 bits ne représentent plus les nombres entiers 0-15 mais les nombres entiers +7 à -

8. De façon similaire/ lorsque l'on utilise 8 bits la gamme valable des nombres est de +127 à -128. Cela peut poser des problèmes lors de l'addition de deux nombres ayant pour résultat un nombre plus grand que +127.

Exemple :

```
100         01100100
 79         01001111
---         --------
179         10101011
```

En complément à 2, 10101011 représente -85; c'est tout à fait faut. On dit qu'un dépassement a eu lieu.
Cela est détecté par le flag P/V (Parité/Dépassement (Overflow)) et c'est au programmeur de réagir en fonction de cette information. Une solution facile est de déclarer le résultat non valable. Un dépassement se produira généralement dans les conditions suivantes:

Lors de :

1. L'addition de deux grands nombres positifs ou négatifs.
2. La soustraction d'un grand nombre positif d'un grand nombre négatif ou lors 'de la soustraction d'un grand nombre négatif d'un grand nombre positif.

Le Z80 contient des instructions qui convertissent le contenu de l'accumulateur en complément à 1 ou en complément à 2, de la manière suivante :

NEG "NEGative" le contenu de l'accumulateur (exactement comme le complément à 2).

CPL ComPLémente le contenu de l'accumulateur (complément à 1).

Le programme 5.3 ci-après calcule le résultat de la somme -112 +101.

PROGRAMME 5.3

```
LD A,112       A=112
NEG            A=-112
ADD A, 104     A=-112+104
CALL &BB5A
RET
```

Une fois lancé, ce programme affichera un petit bonhomme sur l'écran, le caractère de l'Amstrad pour le code ASCII 248. La représentation binaire de 248 est 11111000 ce qui représente -8 en notation complément à 2. Le point important à noter ici, est que le Z80 ne peut faire de distinction entre 248 et -8. Complément à 2 est un concept que le programmeur, et non le Z80, utilise pour la représentation de nombres négatifs, il faut donc se méfier lorsqu'on l'utilise et s'assurer que le résultat espéré est bien obtenu.

EXERCICE 5.10

Utilisez les instructions CPL et INC au lieu de NEG pour calculer le résultat de la somme suivante :

-20 + 98 = ?

Une réponse possible est indiquée au chapitre solutions.

Et voilà, encore un autre chapitre de terminé; et en résumé:

RESUME

Les notions et concepts suivants devraient désormais vous être familiers.

1. Arithmétique BCD

2. Opérateurs logiques
   1. AND (ET)
   2. OR (OU)
   3. XOR (OU EXCLUSIF)
   4. Masques logiques

3. Nombres signés
   1. Complément à 1
   2. Complément à 2
   3. Débordement

Vous devriez aussi reconnaître les instructions suivantes.

```
DAA  AND s
CPL  OR s
NEG  XOR s
```

CHAPITRE 6

MULTIPLICATION, DIVISION ET GROUPE ROTATION

Presque toutes les applications réelles des ordinateurs demandent la manipulation des nombres. Alors que quelques unes ne demandent qu'une addition ou une soustraction fondamentale/ d'autres demandent la multiplication et la division. Dans ce chapitre nous verrons comment ces fonctions arithmétiques peuvent être réalisées en langage assembleur.

MULTIPLICATION BINAIRE

Avant de nous embarquer dans la Multiplication Binaire, observons le processus de la multiplication décimale. Prenez le total 13x14. Nous définissons 13 comme le MULTIPLICANDE et 14 comme le MULTIPLICATEUR et posons la multiplication comme ceci :

```
 13 Multiplicande
 14 Multiplicateur
 --
 52
130
 --
182 Réponse
```

La multiplication est accomplie en multipliant le multiplicande par le chiffre situé à l'extrême droite du multiplicateur et en stockant ce résultat comme le "produit partiel", soit : 13x4=52. Ensuite nous multiplions le multiplicande par le chiffre suivant du multiplicateur, ce qui nous donne un deuxième produit partiel, soit : 1x13=13. On écrit ce produit partiel avec un décalage d'une position vers la gauche. Les deux produits partiels sont alors additionnés et donnent un résultat de 182, ce qui est la réponse juste. Il est tout à fait possible d'utiliser la même méthode pour effectuer une multiplication binaire.

Par exemple, pour multiplier 5x7 en binaire :

5 = 0101 (Calcul en 4 bits seulement)
7 = 0111

```
             0111   = (7)
           x 0101   = (5)
                           Puis en additionnant

Produit partiel 1    0111           0111
Produit partiel 2   00000           0111
Produit partiel 3  011100         100011
Produit partiel 1 0000000         100011
```

Rep: 100011

100011 = (1x32)+(0x16)+(0x8)+(0x4)+(1x2)+(1x1)=35

Avec 1 pour chiffre à l'extrême droite du multiplicateur, le produit partiel a la même disposition de chiffres que le multiplicande, soit : 0111, Sinon le produit partiel est zéro, soit : 0000. Tout nouveau produit partiel est écrit en le décalant d'une position vers la gauche. Ainsi la multiplication binaire se réduit à des additions successives et des décalages.

MULTIPLICATION A 8 BITS

Pour effectuer une multiplication en utilisant le Z80, nous utiliserons l'accumulateur pour garder le 'total en cours', le registre C pour garder le multiplicande et le registre E pour garder le multiplicateur. Voyons maintenant comment sont formés les produits partiels.

```
                                       0 1 0 1
                                       ! ! ! !
Produit partiel   1     0111 =  0111x1 <-!-!-!-!
Produit partiel   2     0000 = 0111 x0 <-!-!-!
Produit partiel   3      0111 = 0111 x1 <-!-!
Produit partiel   4      0000 = 0111 x0 <-!
                           ^          ^
                           ^          ^
----------------------------        ^
!0111 décalé à gauche!               ^
!chaque fois (appelé !  -----------------
!le Produit Partiel  !  !Le bit suivant !
!Invisible PPI)      !  !de 0101 chaque !
----------------------  !fois.          !
                        -----------------
```

Le problème de la multiplication binaire peut être exprimé dans l'organigramme suivant.

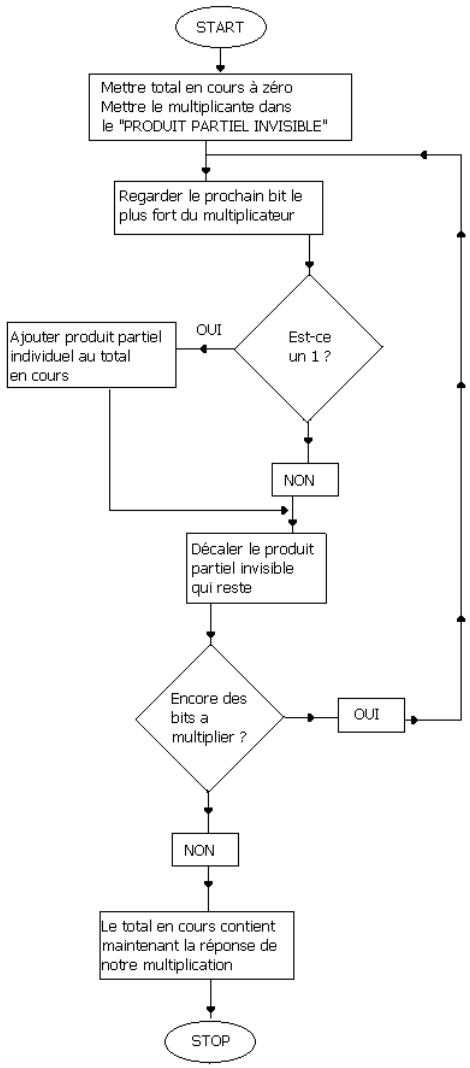


FIGURE 6.1

Ce qu'il nous faut maintenant avant d'écrire le programme, ce sont le instructions qui nous permettront de décaler les bits dans les octets.

SRI s Logical Shift Right (Décalage Logique à Droite) de l'opérande s

Représenté en diagramme :

```
      !---------!         ---
0---->! 7----->0!   ----> !?!
      !---------!         ---
```

Octet opérande Flag Carry (retenue)

Par exemple, examinons le décalage à droite de 10110111 avec l'instruction SRL,

	Octet	Flag Carry
Avant	1 0 1 1 0 1 1 1	?
Après	0 1 0 1 1 0 1 1	1

Le bit 7 a été remplacé par un 0,les bits 1 à 6 décalés à droite d'une position et le bit 0 placé dans le flag Carry (retenue).

Maintenant le programme:

PROGRAMME 6.1

```
        LD A,0          Accumulateur a Zéro
        LD C,7          C = Multiplicande
        LD E,5          E = Multiplicateur
        LD B, 4         B = Compte (pour 4 bits)
ADD:   SRL C            Décaler C à droite
        JR NC,NOADD:    Si bit a l'extrême droite = 0 alors sauter à NOADD
        ADD A,E          Ajouter PPI au total en cours dans l'accumulateur


NOADD: SLA E            Décaler PPI à gauche
        DEC B            Décrémenter le compteur
        JR NZ,ADD:       Resauter à ADD s'il y a encore des bits a multiplier
        ADD A,65         Ajouter décalage
        CALL &BB5A       Afficher le caractère sur l'écran
        RET              Retour
```

Un décalage de 65 est utilisé donnant l'affichage sur l'écran du caractère ayant pour code ASCII 100- soit la lettre minuscule d (La réponse à 7x5 était 35, 35+65=100).

EXERCICE 6.1

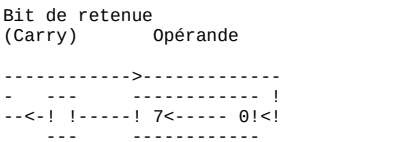
Ecrivez un programme pour multiplier 10x9. Puis afficher le résultat directement sur l'écran.

Une réponse possible est donnée au chapitre solutions.

Le programme 6.1 ne peut que multiplier deux nombres ayant pour résultat un total inférieur à 255. ta principale raison de cette limitation est que le produit partiel invisible est décalé à gauche et sera finalement décalé complètement hors du registre, Un problème similaire s'est présenté lorsque nous faisions une addition et une soustraction à 16 bits, le problème était résolu en rechargeant le registre de gauche avec le bit de retenue quand il y avait dépassement. Ce qu'il faut maintenant, c'est une instruction qui, après une instruction de décalage sur l'octet faible, décalera l'octet fort ainsi que toute retenue générée. Cela se fait avec l'instruction suivante :

RL s Rotation à gauche de l'opérande s ainsi que du bit de retenue.

En diagramme :



Le flag de retenue est chargé dans le bit 0, les bits 1-8 sont décalés à gauche d'une position et le flag Carry reçoit le bit 7.

Pour décaler un registre à 16 bits il nous faut donc deux instructions. L'octet faible est décalé normalement avec l'instruction SLA, puis l'octet fort est décalé avec RL. Par exemple, supposons que nous voulions décaler a gauche le contenu de DE. Les instructions suivantes accompliraient cela :

```
SLA E
RL D
```

Incorporons cette Instruction dans un programme.

Problème :

Affichez sur l'écran le résultat de l'opération suivante: 7x10 = ?

PROGRAMME 6.2

```
            LD C,7          C = Multiplicande
            LD E,10         E = Multiplicateur
            LD D,0          Registre D a zéro
            LD B,8          B = Nombre de bits
            LD HL,0         HL à zéro, on l'utilise pour garder le total actuel
NXTB:       SRL C           Décaler Multiplicande a droite
            JR NC,NOADD:    Si retenue=0 sauter à NOADD
            ADD HL,DE       Ajouter PPI au total en cours
NOADD: SLA E          Décaler octet faible à gauche
            RL D            Recharger la retenue et décalez D à gauche
            DEC B           Décrémenter compteur
             JR NZ,NXTB:     Resauter si encore des bits
            LD A,L          Charger dans A la réponse RET
```

NB. Bien que nous ayons utilisé un registre a 16 bits pour le résultat en sélectionnant soigneusement le multiplicateur et le multiplicande, seul l'octet faible contient une valeur, ce qui nous permet de l'afficher sur l'écran.

EXERCICE 6.2

En utilisant le programme 6.2 calculer la réponse de 146x124. Notez que cela donnera une réponse a 16 bits, donc H et L devront être affichés sur l'écran.

Une réponse possible est donnée au chapitre solutions.

Le programme 6.2 a utilisé le registre B comme compteur, Chaque fois que le multiplicateur était décalé à gauche, B était décrémenté puis testé. Cette opération demande deux instructions: DEC et JR NZ. Ces deux instructions, peuvent néanmoins être remplacées par une seule, qui rend le programme plus efficace et plus élégant.

DJNZ e Décrémenter B et si cela donne une réponse Non-Zéro sauter a l'adresse mémoire e.

EXERCICE 6.3

Remplacez les Instructions DEC B et JR NZ-NXTB par une instruction DJNZ NXTB dans le programme 6.2, ou dans votre réponse a l'exercice 6.2

Une réponse est donnée dans le chapitre solutions.

La division binaire est aussi possible avec une méthode très similaire à celle utilisée pour la multiplication.

DIVISION BINAIRE

Observons la division suivante :

```
   ----
10 !785
```

D'abord nous essayons de diviser 7 par 10; comme 10 ne rentre pas dans 7 la prochaine étape c'est de prendre le 8 et d'essayer de diviser 78 par 10. Il y va sept fois et il reste 8.

```
     7
    ----
10  !785
```

comme 10 'ne rentre pas dans' 8, la prochaine étape c'est de prendre 5 et d'essayer de diviser 85 par 10. Il y va 8 fois et il reste 5.

```
    78
```

```
   ----  reste 5
10 !785
```

Donc le résultat est 78 et le reste 5. Cette forme de division est appelée "division entière" puisque les fractions en sont exclues.

Comme pour la multiplication les différents nombres d'une division ont des noms; par exemple "reste" est déjà familier. Les autres noms sont :

```
                    quotient
                       !
              78--------!
             ----
          10 !785      reste = 5
         ^    ^
         ^-------- Dividende
Diviseur--
```

Etudions maintenant une division de 16 bits par 8 bits.

Problème Afficher sur l'écran le résultat de la division suivante dans un format reconnaissable.

2765 : 75 = ?

La division binaire se fait de la manière suivante, Si sans emprunt (retenue négative), le diviseur à 8 bits peut être soustrait de l'octet fort du dividende a 16 bits, alors le bit approprié de la réponse à 8 bits est mis. Ce procédé se répète huit fois, donnant un quotient à 8 bits et un reste à 8 bits.

PROGRAMME 6.3

```
       LD HL,2765         HL = Dividende
       LD C,75             C = Diviseur
       LD B,8              B = compte
NXT:  ADD HL,HL           Décaler dividende a gauche
       LD A,H              A = octet fort du dividende
       SUB C               Soustraire diviseur
      JR C,NXTB:          Si retenue sauter à NXT
       LD H,A              Recharger octet fort du dividende
       INC L               Incrémenter la réponse
NXTB: DEC B               Décrémenter le compteur
       JR NZ,NXT:          Si non-zéro sauter à NXT
       LD A,L              A = réponse (quotient)
       CALL &BB5A          Afficher la réponse
       LD A, H             A = reste
       CALL &BB5A          Afficher le reste RET
```

Lancez le programme. Il affichera '$A' sur l'écran, ici le code ASCII de !$! correspond à la réponse et 'A' au reste. Vérifiez que ceux-ci soient justes. Il faut noter que comme le Z80 ne possède pas une instruction
de décalage à gauche à 16 bits, on a utilisé ADD HL,HL. En binaire, ajouter un nombre à lui-même c'est la même chose que de le décaler à gauche d'une position bit, ou de le multiplier par deux. Si vous comparez cela avec la base 10, par exemple, décaler 19 à gauche, cela donnera 190 -l'équivalent d'une multiplication par 10. A votre avis, qu'est-ce qui se passerait si on décalait d'une position bit a droite un nombre binaire ? (Réponse : c'est la même chose que de diviser par 2).

Jusqu'ici, seules quelques instructions de décalage du Z80 on été utilisées pur des routines de multiplication et division d'usage général. Il est quelque fois Plus facile et plus rapide d'utiliser un petit groupe d'instructions de décalage ou de rotation pour exécuter une opération arithmétique spécifique.
Quand on utilise une instruction de rotation par opposition à une instruction de décalage, le contenu du registre est modifié, mais aucune information n'est perdue, c'est à dire qu'aucun bit ne disparaîtra sans laisser de trace. Voici une instruction qui fait Pivoter à gauche le contenu de l'accumulateur :

RLCA Rotation à gauche de l'Accumulateur et chargement d'une copie du bit 7
      dans le flag Carry.

RLCA sous forme de diagramme :

```
Flag Carry          Accumulateur
                  -------->-------
      ----        !  --------    !
      !  !-----------!7<---0!<----
       ----          --------
```

Notez que le bit 7 n'est cas perdu mais inséré dans le bit 0. Une Instruction pour faire pivoter à droite le contenu de l'accumulateur existe également :

RRCA Rotation à droite (Right) du contenu de l'Accumulateur et chargement
      d'une copie du bit 0 dans le flag de retenue.

Utilisons ces deux Instructions.

Problème : Multipliez 10 par 16, affichez sur l'écran le résultat, divisez ce résultat par 2 et
            affichez le sur l'écran.

PROGRAMME 6.5

```
LD A,10     A=10
RLCA        A=20
RLCA        A=40
RLCA        A=80
RLCA        A=160
CALL &BB5A  Affiche A ('•')
RRCA        A=80
CALL &BB5A  Affiche A('P')
RET
```

EXERCICE 6.4

Ecrivez un programme pour calculer et afficher sur l'écran les résultats des opérations suivantes en utilisant des instructions de rotation et d'addition.

1. 5 x 32 = (160)
2. 254 – 2 = (127)

Les réponses possibles sont données dans le chapitre solutions.

Le Z80 possède encore d'autres Instructions de rotation et de décalage que nous n'avons pas utilisées. Elles sont détaillées dans les appendices sous le titre 'Jeu d'instructions du Z80',

Observons maintenant un jeu d'instructions oui permet la mise, l'annulation et le test de bits isolés au sein d'un octet.

LE GROUPE MISE, ANNULATION ET TEST DE BIT

Examinez l'instruction suivante :

BIT b,r Teste le bit à la position b dans l'opérande r.

Cette instruction est utilisée chaque fois qu'un bit particulier dans un octet a besoin d'être testé, par exemple dans les opérations d'entrée/sortie.

Pour illustrer l'utilisation de cette commande, un registre sera mis à zéro, incrémenté jusqu'à la mise du bit 7, le contenu du registre qui en résultera sera alors affiché sur l'écran.

PROGRAMME 6.6

```
       LD A,0                  A=0
NXT:   INC A                   Incrémenter A
       BIT 7,A                 Tester bit 7 de l'accumulateur
       JR Z,NXT:               Le bit 7 est-il mis ?
       CALL &BB5A              Afficher contenu de A sur écran
       RET
```

Ce programme charge zéro dans A et l'incrémente continuellement jusqu'à la mise du bit 7. Lancez-le; on ne remarque pas grand chose puisque la mise du bit 7 correspond à 128 en décimal, un espace en ASCII. Essayez d'insérer soit
l'instruction INC A soit l'instruction DEC A juste avant l'instruction CALL &BB5A, pour imprimer quelque chose de visible sur l'écran.

Deux autres instructions de bit existent: l'une pour mettre le bit et l'autre pour annuler un bit individuel dans un octet.

SET b,r Met le bit en position b dans l'opérande r.

RES b,r Annule (RESet) le bit en position b dans l'opérande r.

Pour illustrer ces instructions, observons le code ASCII de 'C', en binaire 01000011. Si le bit 0 est annulé, c'est la même chose que soustraire un, ce qui donne la représentation ASCII de B.

PROGRAMME 6.7

```
LD A, 67          A=ASCII pour 'C'
CALL &BB5A        Affiche 'C'
RES 0,A           Annule bit 0
CALL &BB5A        Affiche 'B'
SET 0,A           Met le bit 0
CALL &BB5A        Affiche 'C'
RET
```

EXERCICE 6.5

Ecrivez un programme qui charge 255 dans l'accumulateur, affiche le contenu sur l'écran, annule le bit 4, affiche le résultat sur l'écran, met le bit 1, annule le bit 3 et puis affiche le résultat final sur l'écran.

Une réponse possible est donnée au chapitres solutions.

Voilà un autre chapitre terminé, à présent vous devriez avoir une assez bonne idée de comment écrire un programme relativement complexe.

Et pour résumé:

RESUME

Les notions et les groupes d'instructions suivants devraient à présent vous être assez familiers :

1. Multiplication et division binaires
2. Le groupe décalage et rotation
3. La groupe mise, annulation et test de bit.

CHAPITRE 7

LA PILE

La Pile est un bloc de mémoire situé à partir de &C000. Elle est utilisée pour le transfert rapide de données, et remplie à partir de &C000, la position libre suivante étant enregistrée par un registre appelé le 'Pointeur de Pile ' (Stack Pointer) ou SP. L'analogie courante est une pile d'assiettes, dont seule l'assiette supérieure est accessible puisqu'elle est la dernière qui a été posée. Mais la pile est remplie en partant du haut, soit à partir de &C000 jusqu'à zéro, donc les assiettes sont mises ou enlevées par le bas, une manière de faire aux antipodes de la logique ! Cette façon de remplir et de vider la plie est appelée 'Last In, First Out ou LIFO (dernier entré, premier sorti). Donc la Pile est une mémoire LIFO.

L'une des fonctions de la pile est d'enregistrer des adresses pendant les appels de sous-programmes que le Z80 fait automatiquement. Quand le Z80 voit une instruction telle que CALL &BBBA, il lui faut d'abord enregistrer l'endroit où se trouve l'instruction suivante dans la routine d'appel (programme principal), de façon à la retrouver après l'exécution du sous-programme. Puis il place le 'BBBA' dans le PC (compteur de programme). Supposez que le Z80 soit prêt à exécuter l'instruction suivante :

```
CALL &BBBA
```

Le Z80 accomplit les choses suivantes :

1. Incrémente PC
2. Appelle l'octet de commande (CALL')
3. Incrémente PC
4. Appelle le premier octet de l'opérande (&BA)
5. Incrémente PC
6. Appelle le second octet de l'opérande (&BB)
7. Stocke l'octet fort de PC sur la pile
8. SP = SP-1
9. Stocke l'octet faible de PC sur la pile
10. SP=SP-1
11. Met &BBBA dans PC
12. Exécute le sous-programme. Jusqu'à RET
13. Incrémente PC
14. Appelle l'octet de commande CRET')
15. Charge l'octet faible de PC à partir de la pile
16. SP=SP+1
17. Charge l'octet fort de PC a partir de la Pile
18. SP=SP+1
19. Incrémente PC
20. Continue avec le programme principal

Dans cet exemple, le sous-programme aurait bien pu rencontrer d'autres sous-programmes, des sous-programmes 'imbriqués', et à chaque exécution de CALL l'adresse de retour aurait été empilée sur la pile. Puis, alors que le programme revenait (RET) de ces sous-programmes, les adresses de retour auraient été enlevées successivement pour ramener le Z80 au point initial de départ. La zone de mémoire utilisée pour la pile est supérieure à 256 octets, donc il y a tout à fait assez de place.

Heureusement, l'opération de l'enregistrement des adresses par la pile, lors de l'exécution des sous-programmes, est entièrement automatique et le programmeur peut donc laisser le Z80 faire ce travail. Néanmoins, lors de l'utilisation de sous-programmes incorporés, la plie ne stocke pas automatiquement le contenu des registres mais elle doit être programmée pour le faire, les instructions à cet effet sont :

PUSH et POP

PUSH qq Placer le registre double qq sur la pile

Ici, qq est AF, BC, DE ou HL. PUSH permet aussi de par sa forme le stockage de IX et IY :

PUSH IX Placer le registre IX sur la pile

PUSH IY Placer le registre IY sur la Pile

Ces instructions recopient sur la pile le contenu des registres spécifiés. Après avoir traité le sous-programme ou autre chose, pour restaurer le contenu précédant des registres, on utilise l'Instruction POP :

POP qq Enlever le contenu de registre de la pile

Là aussi qq est AF, BC, DE ou HL.

POP IX Enlever IX de la pile
POP IY Enlever IY de la pile

Que vous utilisiez PUSH ou POP, ce n'est pas la peine de remettre à Jour le pointeur de pile, cela se fait automatiquement.

Lors de la mémorisation de contenus de registres sur la Pile, il est Important de se rappeler que la Pile est une structure LIFO : vous enlèverez normalement les registres dans l'ordre inverse de leur stockage, Et puis, lors de l'écriture d'un programme avec des PUSH et des POP, chaque PUSH doit être assorti d'un POP, sinon l'ordinateur pourrait "être induit en erreur : il pourrait penser par exemple, qu'un nombre de la Pile est une adresse RETour, alors qu'il s'agit du registre double BC qui n'a pas été enlevé. Evidemment, cela conduirait à un 'plantage' du système. Le programme 7.1 est un exemple de l'utilisation correcte de PUSH et POP pour préserver A et HL. Il utilise deux des routines incorporées de l'ordinateur; l'une est la routine de sortie de texte &BB5A déjà bien connue. L'autre est à l'adresse &BB75 et fixe la position du curseur texte (Voir appendice 4 pour plus de détails). Pour l'utiliser, H doit contenir la position colonne et L la position ligne qui sont requises, Notez que &BB75 altère les registres, c'est donc une excellente idée que de les stocker sur la pile !

PROGRAMME 7.1

```
LD HL,&0604          Charge dans H colonne 4 et dans L ligne 6
LD A,&4D             Charge dans A ASCII de 'M'
PUSH HL              Recopie HL sur la plie
PUSH AF              Recopie A sur la plie
CALL &BB75           Fixe le curseur en utilisant HL
POP AF               Rappelle A
CALL &BB5A           Ecrit 'M' sur le curseur en utilisant A
LD BC,&0101          Fixe décalage du curseur dans BC
ADD HL,BC            Ajoute décalage à HL
PUSH AF              Sauvegarde AF sur la pile
CALL &BB75           Remet à jour la position du curseur (HL)
POP AF               Rappelle A
CALL &BB5A           Ecrit 'M' à la nouvelle position du curseur
RET                  Retour au BASIC
```

Ceci devrait produire deux 'M' dans des positions diagonales consécutives. Deux points sont a noter dans le programme 7.1. Tout d'abord, bien que les choses soient poussées sur la pile, l'information demeure dans les registres appropriés Jusqu'à qu'ils soient changés, puisque PUSH recopie seulement les registres sur la pile; dans le programme ci-dessus, les contenus des registres sont changés ('altérés') de manière imprévisible par l'utilisation de routines incorporées. Néanmoins, il était tout à fait correct de recourir a PUSH AF puis à CALL qui appelait une routine utilisant le contenu du registre A, puisque celui-ci n'avait pas été changé jusqu'ici. Un autre point a noter: bien que seul A devait être préservé, AF devait être poussé sur la pile, puisque PUSH ne fonctionne qu'avec des registres doubles. Il en va de même pour POP.

EXERCICE 7.1

Ecrivez un petit programme pour :

a) fixer le curseur graphique a (0,0)
b) mettre 100 dans DE et 200 dans HL
c) sauvegarder DE et HL sur la pile
d) tracer une ligne jusqu'à (100,200)
e) remettre le curseur graphique à (0,0)
f) retirer DE et HL mais en mettant l'ancien contenu de DE dans HL et vice-
   versa
g) tracer une ligne Jusqu'à (200,100)

Indications : voir appendice 4; &BBCO pour fixer le curseur et &BBF6 pour tracer les lignes. Une réponse possible est donnée dans le chapitre solutions.

Si vous avez fait l'exercice ci-dessus, vous avez peut être remarqué dans la solution une façon intéressante de permuter les contenus de DE et HL :

```
PUSH DE
PUSH HL
    .
    .
POP DE
POP HL
```

Les registres sont retirés dans le 'mauvais' ordre.

Il existe une autre manière de permuter le contenu des registres doubles DE et HL :

EX DE,HL Echange (EXchange) le contenu de DE avec HL

Cette instruction ne marche que si elle est écrite comme ci-dessus; c'est à dire que EX HL,DE ou EX BC,HL etc ne marchera pas.

LE POINTEUR DE PILE (SP)

Pour garder trace des éléments à enlever (POP) ensuite, ou pour garder trace de l'endroit où placer l'élément suivant à placer (PUSH) sur la pile, le Z80 utilise un registre spécial appelé le 'POINTEUR DE PILE' ou SP (STACK POINTER). Celui-ci pointe communément à la dernière position dans la pile - soit le dernier octet dans la Pile.

Il existe diverses instructions qui permettent au programmeur d'accéder au pointeur de pile, et de le changer. Lorsqu'on les utilise, il est possible de créer une 'pile utilisateur'. Après avoir ajusté SP pour qu'il pointe sur une nouvelle adresse mémoire, le Z80 utilisera la nouvelle adresse comme départ d'une nouvelle pile, ignorant totalement l'ancienne. Par conséquent, si une nouvelle pile est créée dans un sous-programme, il est important de préserver toutes les adresses RETour en les plaçant (PUSH) sur la nouvelle pile.

Trois instructions LD peuvent être utilisées pour altérer SP :

LD SP,HL   Charge dans SP le contenu de HL LD SP,IX

Charge dans SP le contenu de IX LD SP,IY

Charge dans SP le contenu de IY

Pour trouver la valeur actuelle de SP, on utilise l'instruction suivante:

LD (nn),dd Charge dans la case mémoire nn le contenu du registre double dd

dd est soit BC, DE, HL ou, dans le cas présent, SP.

Simplement pour vous prouver que la création de votre pile fonctionnera, essayez le programme suivant :

PROGRAMME 7.2

```
LD A,43                 Met '+' dans A
PUSH AF                 Stocke A dans la pile actuelle
CALL &BB5A              Met '+' sur l'écran
LD (&714A),SP           Se rappelle de la valeur actuelle de SP
LD HLS7148              Se prépare pour :
LD SP,HL                Créer la nouvelle pile en 87148
LD A,61                 Met '=' dans A
PUSH AF                 Stocke A dans la nouvelle plie
CALL &BB5A              Met '=' sur l'écran
LD HL,(&714A)           Trouve position initiale de SP
LD SP,HL                Revient à la pile initiale
POP AF                  Enlève '+'
CALL &BB5A              Met '+' sur l'écran
RET
```

Le programme 7.2 mettra '+=+' sur l'écran. Si le pointeur de pile n'avait pas été amené à une nouvelle position/ alors POP AF aurait enlevé '=' a la place, et l'écran aurait affiché '+=='. Pour prouver qu'il y a une nouvelle pile en &7148, essayez ce petit exercice !

EXERCICE 7.2

Prolonger le programme ci-dessus afin qu'il enlève le '=' et l'affiche puis remette SP là ou il était pour enlever et afficher à nouveau un autre '+', l'écran devrait alors afficher '+=+=+'.

Indications : Rappelez-vous que SP doit pointer sur la dernière position dans la pile, par conséquent l'amener à &7148 ne marchera pas. Utilisez 87146.

Une solution est donnée au chapitre solutions.

Pour conclure ce chapitre, il faut mentionner les autres commandes qui agissent sur la Pile. Celles-ci permettent au programmeur avancé et 'prudent' de manipuler le contenu de la pile sans utiliser chaque fois PUSH ou POP pour que le SP pointe a l'élément souhaité dans la pile.

Tout d'abord, il existe trois autres instructions d'échange (EXchange) :

EX (SP),HL Echange le contenu de HL avec le dessus de la plie. EX (SP),IX Echange le contenu de IX avec le dessus de la pile EX (SP),IY Echange le contenu de IY avec le dessus de la plie.

Les autres instructions qui peuvent agir sur SP sont :

INC ss INCrémente le registre double ss

DEC ss DECrémente le registre double ss

ss est BC, DE, HL ou SP.

ADD IX,pp Ajoute le contenu du registre double PP à IX

Ici PP est BC, DE, IX ou SP.

ADD IY,rr Ajoute le contenu du registre double rr à IY

Ici rr est BC, DE, IY, ou SP.

RESUME

Après avoir lu ce chapitre, vous devez connaître ceci :

LIFO EX DE,HL
SP Quelle partie de la pile SP pointe-t-il
PUSH Pile utilisateur
POP LD (nn),dd
EX (SP),HL

CHAPITRE 8

MOUVEMENTS DE BLOCS ET COMPARAISONS

Le Z80 possède plusieurs instructions faites pour permettre une manipulation facile de sections de mémoire, et sans que le programme ait à inclure les adresses de chaque adresse mémoire. Ces instructions peuvent être partagées en deux catégories. Les mouvements de blocs permettent a des zones de mémoire d'être recopiées d'un endroit a un autre. Les comparaisons de blocs sont utilisées afin de rechercher une zone de mémoire pour un élément de donnée spécifique.

MOUVEMENTS DE BLOCS

La première instruction de mouvement de bloc que nous allons étudier dans ce chapitre est LDI :

LDI Charge (LoaD) dans la mémoire et Incrémente les pointeurs de données.

Pour utiliser cette instruction , HL doit contenir l'adresse de la case mémoire à partir de laquelle les données doivent arriver, et DE doit contenir l'adresse de la case mémoire où les données doivent être recopiées. Après exécution de cette instruction, le Z80 incrémente a la fois DE et HL. Il décrémente également BC; ce qui rend LDI particulièrement pratique dans les boucles - BC peut être utilisé comme un compteur de boucle. Quand BC est décrémenté a 1, le flag parité/dépassement est mis a zéro - pour d'autres valeurs de BC, LDI met ce flag a 1, afin qu'il puisse être testé pour terminer une boucle s'il est mis, en utilisant la condition PO. Nous détaillons cela ci-après; mais voyons d'abord le programme.

Comme exemple d'utilisation de LDL le programme 8.1 recopie de la mémoire sur l'écran.

Il recopie des données, d'une partie de la pile sur l'écran, ces données occupant les positions mémoire &B100 à &BFFF, qui correspondent aux positions mémoire &C000 à &FFFF inclues.

PROGRAMME 8.1

```
            LD HL,&B992     Charge dans HL adresse de départ des données
            LD DE,&C000     Charge dans DE la destination
            LD BC,&A1       Charge dans BC la quantité de données + 1
LOOP: LDI           Recopie un octet de données
            JP PO,FINISH    Sort de la boucle si BC = 1
            JP LOOP         Sinon continue la boucle
FINISH:     RET
```

Cela devrait produire une ligne horizontale multicolore en travers de l'écran, avec une petite ligne multicolore aux extrémités opposées de chaque côté.

PARITE

Comme nous l'avons mentionné plus tôt, le flag de parité/dépassement est mis lorsqu'il est décrémenté à 1 plutôt qu'à 0, il est donc nécessaire, pour recopier correctement la mémoire, de charger dans BC le nombre d'octets à recopier + 1. Pour tester le flag parité/dépassement, les conditions sont PO -'parité impaire' (Odd) et PE 'parité paire' (Even)'.

'Parité' se réfère au nombre de bits mis dans l'octet ou au flag testé; parité paire signifie qu'un nombre pair de bits sont mis et parité impaire signifie qu'un nombre impair de bits sont mis. Donc, lorsque le flag de parité est mis, il contient un nombre impair (1) de bits mis, et peut être testé sur la condition PO, comme dans le programme 8.1 ci-dessus:

JP PO,FINISH:

Si Jamais il vous faut tester la parité d'un octet de données, vous pouvez le faire en chargeant 255 dans h, et en lui faisant une opération ET (AND) avec l'octet en question. Le flag de parité sera mis de façon appropriée et on peut le tester avec une instruction telle que le JP

conditionnel ci-dessus. Les tests de parité sont généralement utilisés en systèmes de communications : un bit de chaque octet sera mis de côté pour 'être utilisé comme 'bit de parité'/ qui sera mis ou enlevé de façon appropriée pour chaque octet, afin que tous les octets transmis et reçus aient une parité identique. Si une interférence dénature les données, il y a de fortes chances que la parité de quelques octets aient été changée, et ceci peut être détecté.

EXERCICE 8.1

Ecrivez un programme similaire au programme 8,1 pour recopier &3FFF octets en commençant en &B100, et pour les afficher sur l'écran en commençant en &C000, ATTENTION : une erreur conduirait au 'plantage' de l'ordinateur, ce qui vous obligerait à recharger l'assembleur. Une réponse possible est donnée dans le chapitre solutions.
La solution de l'exercice 8.1 remplit sans problème l'écran de n'importe quoi, mais ce serait bien de voir tout cela se faire plus lentement. Essayez le programme 8.2 :

PROGRAMME 8.2

```
         LD HL,&B100
         LD DE,&C000
         LD BC,&4000
LOOP:    LDI
         JP PO,FINISH:
         LD A,&FF
DELAY:   DEC A              Instructions 'gaspillant' du temps
         JP NZ,DELAY:
         JR LOOP:
FINISH: RET
```

Une fois le programme lancé, l'écran se remplit ligne par ligne, les lignes tout d'abord séparées, puis les intervalles entre les lignes sont remplis.

La raison de ce type de remplissage de l'écran plutôt qu'un autre, par exemple, ligne par ligne sans séparation initiale entre les lignes, la raison en est la manière dont l'ordinateur contrôle l'écran. Des octets consécutifs de mémoire-écran (soit de &C000 à &FFFF) ne contrôlent pas des cases écran consécutives sur le moniteur. Les détails de l'organisation de la mémoire écran sont liés au 'système d'exploitation' de l'ordinateur plutôt qu'au code macnine ou au langage assembleur.

Le programme 8.2 illustre, indirectement, l'utilité principale de l'instruction de mouvement de bloc LDI : vous pouvez insérer d'autres instructions entre les opérations successives de l'instruction LDI.
Le programme 8.1 illustre l'inconvénient principal de LDI : si vous n'avez pas besoin d'autres instructions entre chaque opération LDI, il vous faut malgré tout un retour à LDI pour qu'il se répète. Cela gaspille de l'espace mémoire et du temps. Le Z80 a un truc pour cela :

LDIR Charge la mémoire dans la mémoire, Incrémente les pointeurs de
données et se Répète.

Cette instruction agit de la même manière que LDI, sauf qu'elle se répète toute seule jusqu'à ce que BC ait été décrémenté à 0. Et aussi, elle met le flag de parité/dépassement à 0 quelle que soit la valeur actuelle de BC. De toute façon cela n'a bien sûr pas d'importance, puisqu'avec la répétition automatique, il n'est pas nécessaire de faire de tests pour savoir si elle a terminé ou pas. Le programme 8.3 est une autre version du programme 8.1, où l'on a remplacé LDI et JP par LDIR.

PROGRAMME 8,3

```
LD HL,&B992
LD DE,&C000
LD BC,&A0
LDIR
RET
```

Notez que dans ce cas, il n'est pas nécessaire de charger dans BC le nombre d'octets plus un à transférer, puisque LDIR s'arrête lorsque BC est à zéro, alors que dans le programme 8.1 et 8.2 la boucle se terminait quand BC=1. Il suffit de charger dans BC le nombre actuel d'octets à transférer. Il existe encore deux autres instructions de mouvement de bloc :

LDD Charge la mémoire de la mémoire. Décrémente les pointeurs de pile LDDR Charge la mémoire dans la mémoire. Décrémente les pointeurs de plies et se répète

Celles-ci sont respectivement semblables à LDI et LDIR, sauf que DE et HL sont décrémentés au lieu d'être incrémentés. Par exemple, essayez le programme 8.4 qui remplira l'écran de bas en haut :

PROGRAMME 8.4

```
          LD HL,&BFFF
          LD DE,&FFFF
          LD BC,&4000
LOOP:     LDD
          JP PO,FINISH:
          LD A,8FF
DELAY:    DEC A
          JP NZ,DELAY:
           JP LOOP:
FINISH:   RET
```

COMPARAISONS

C'est souvent pratique dans un programme de pouvoir comparer deux valeurs, puis de se décider en fonction du résultat - comme dans la construction BASIC IF...THEN. L'équivalent en langage assembleur est:

CP s ComPare s à A en soustrayant s de A et en laissant A inchangé

Ici, s est A, B, C, D, E, H, L, (IX,d), ou (HL), Après soustraction de s de A, le résultat n'est pas gardé (c'est-à-dire que A n'est pas altéré) bien que les conséquences de la soustraction aient été enregistrées par le flag de retenue (Carry), le flag zéro, le flag de dépassement, le flag de signe et le flag de demi-retenue. Les flags de signe, zéro et de demi-retenue sont étudiés dans un autre chapitre du livre, Ils sont utilisés lorsque l'on veut comparer des nombres en complément a deux.

Etant donné que la comparaison soustrait s de A, le flag de retenue sera mis si s>A, et annulé dans l'autre cas. Si s=A alors le flag zéro sera mis, sinon annulé. Le programme 8.5 illustre l'utilisation de CP.

PROGRAMME 8.5

```
          LD A,&21            Met un nombre dans A
          LD B,&40            Met un nombre dans B
          CP B                Compare A à B
          JP C,BIGGER:        Si flag Carry mis, B>A
```

```
            JP Z,EQUAL:          Si zéro mis, B=A
            LD A,&73             Sinon B<A; mettre 'S' ASCII dans A
            CALL &BB5A           Met sur l'écran
            LD A,83C             '<' ASCII
            CALL &BB5A           Met sur l'écran
            ID,&41                'A'  ASCII
            CALL &BB5A           Met sur l'écran
            RET                  Sortie programme
BIGGER:     LD A,&73             'S' ASCII
            CALL &BB5A           Met sur l'écran
            LD A,&3E             '>' ASCII
            CALL &BB5A           Met sur l'écran
            LD A,&41             'A' ASCII
            CALL &BB5A           Met sur l'écran
            RET                  Sortie programme
EQUAL: LD A,&73            'S' ASCII
            CALL &BB5A           Met sur l'écran
            LD A,&3D             /=/ ASCII
            CALL &BB5A           Met sur l'écran
            LD A,&41             'A' ASCII
            CALL &BB5A           Met sur l'écran
            RET
```

Essayez encore ce programme avec les valeurs &40 en A et &21 en B, et avec &21 dans A et B pour vérifier que cela fonctionne comme vous le souhaitez.

Non seulement le Z80 possède l'instruction de comparaison ci-dessus, mais il possède aussi des instructions de comparaison de blocs. La première de celles-ci est CPI:

CPI Compare A avec la mémoire, Incréments le pointeur de données.

Comme avec LDI, cette instruction décrémente BC, le compteur d'octets (Byte Counter). Le flag de parité est également mis lorsque BC est décrémenté à 1, sinon il est annulé. Cette instruction n'utilise pas le flag de retenue, mais elle agit malgré tout sur le flag zéro. Cela veut dire que CPI ne peut être utilisée que pour tester l'égalité, pas supérieur à ou inférieur à. Le programme 8.6 cherche dans la mémoire une parenthèse '(' et met une '(' sur l'écran pour chaque parenthèse trouvée dans la mémoire.

PROGRAMME 8.6

```
            LD A,&28          Met '(' dans A
            LD HL,&0          Commence au début de la mémoire
            LD BC,&0          Recherche sur 64k (BC=octets+1,0-1=FFFF)
LOOP: CPI
            JP PO,FINISH:     Fin si flag de parité mis
            CALL Z,FOUND:     Appelle 'found' si zéro mis
            JP LOOP:          Boucle encore
FINISH:     CALL Z,FOUND:     Vérifie flag zéro avant fin
            RET               Retour à l'assembleur
FOUND:      CALL &BB5A        Met '(' sur écran
            RET               RETour du sous-programme
```

Il faut noter dans ce programme que le sous-programme 'FOUND' doit être appelé a partir de 'FINISH' parce que si Z était testé avant PO, et si un '(' était trouvé, appeler BB5A altérerait le flag de parité. Si PO était testé d'abord, ce problème ne se produirait pas.

Une autre instruction de comparaison de blocs est :

CPIR ComPare A avec la mémoire, Incrémente le pointeur de données et se
      Répète.

Cette instruction est semblable à CPI, sauf qu'elle continue automatiquement jusqu'à ce qu'une identité soit rencontrée ou que BC=0, Le programme 8.6 peut être réécrit en remplaçant simplement CPI par CPIR, Le nouveau programme sera plus rapide avec CPIR parce que les conditions n'ont cas a être testées après chaque comparaison.

Les deux dernières comparaisons de blocs sont les mêmes que CPI et CPIR sauf que le pointeur de données, HL est décrémenté au lieu d'être incrémenté :

CPD   ComPare A avec la mémoire. Décrémente le pointeur de données
CPDR ComPare A avec la mémoire. Décrémente le pointeur de données et se
      répète

A l'inverse de LDD et LDDR, ces instructions agissent sur le flag de parité.

Au cas où vous en ayez besoin, vous trouverez dans les annexes une table des effets des comparaisons sur les flags pouvant être testés C (retenue), S (signe) et V (débordement). Il faudra vous y référer si vous devez comparer des nombres en complément à deux (soit des nombres supérieurs à 128)

RESUME

Après ce chapitre, vous devriez connaître

```
LDI                          LDD
PE                           LDDR
PO                           CP s
parité                       CPI
une boucle  de temporisation  CPIR
LDIR                         CPD
                             CPDR
```

CHAPITRE 9

OPERATIONS SPECIALES ET INTERRUPTIONS

Ce chapitre traite quelques propriétés parmi les moins utiles du Z80 en langage assembleur, tout au moins d'un point de vue purement logiciel. Beaucoup de ces opérations et propriétés spéciales décrites dans ce chapitre ne sont pratiquement pas utilisées du tout par la plupart des programmeurs.

Cependant, il est intéressant de les connaître. Les interruptions, expliquées ci-dessous, sont très utiles, mais en faire une explication détaillée ne sera pas notre but car elles sont plutôt un sujet pour un livre axé sur l'électronique.

Les interruptions

Quand on exécute un travail qui doit être fait, personne n'aime les interruptions tant que le travail n'est pas fini. Le Z80 est aussi comme ça, Quand il exécute un bout de programme, il a tous ses registres sous contrôle et tous ses flags mis correctement. Une Interruption, cependant, est un sous-programme qui demande a être exécuté quand il est prêt, pas quand le Z80 est prêt. En clair, une interruption vient de l'extérieur du champ de contrôle direct du Z80 –soit d'un appareil externe, soit du clavier. Dès lors, les flags doivent "être stockés par le Z80 - le plus souvent sur la pile, Alors le Z80 prendra en charge l'Interruption - c'est-à-dire faire tout ce que doit faire l'interruption, et il doit ensuite restaurer tous les registres et flags et continuer avec le programme originel. Manier des interruptions doit être possible dans tous les programmes qui attendent des interruptions, particulièrement si le programme effectue certains travaux qui ne doivent pas être interrompus. Si, par exemple un autre appareil envole une suite de données en mémoire/ alors une procédure "hand-shaking" (poignée de mains) est enclenchée entre les deux appareils. Tout simplement c'est un échange de messages du genre: "Je suis prêt à envoyer les données/ êtes-vous prêts à les recevoir?" "Oui" "Voilà les données ... fin des données." "Merci!"

Si un tel échange est Interrompu, les données risquent bien d'être mutilées, et donc d'être inutilisables, Pendant de telles périodes où aucune interruption n'est possible, le programme peut bloquer la plupart des interruptions - pas toutes - pour permettre à une procédure particulière d'être achevée. Les interruptions qui peuvent être bloquées sont appelées interruptions masquables, et les interruptions qui ne peuvent être bloquées sont appelées interruptions non-masquables, ou NMI.

L'instruction qui bloque toutes les interruptions masquables est DI:

DI DIsable maskable interrupts (Empêche les interruptions masquables)

Une fois achevée la partie de programme, de préférence courte, qui ne peut être interrompue, on peut à nouveau rendre possible les interruptions en utilisant EI

EI Enable maskable Interrupts (Permet les interruptions masquables)

Quand le Z80 prend en charge une interruption, il le fait au moyen d'un sous-programme en code machine. Comme les autres sous-programmes, ils doivent se terminer par une instruction de retour. Pour les interruptions non-masquables l'instruction est:

RETN RETurn from Non-maskable interrupt (Retour d'interruption non-masquable)

Pour les interruptions masquable c'est:

RETI RETurn from Interrupt (Retour d'interruption)

Si vous réfléchissez à ce qui précède, vous avez peut-être compris que les interruptions peuvent être interrompues. Cependant, il y a des priorités: par exemple, une interruption masquable ne pourra normalement pas interrompre une interruption non-masquable - la plupart du temps elle devra attendre.

L'interruption la plus prioritaire de toutes est une "demande du bus" ou BUSRQ. Avec les autres interruptions masquables et non-masquables, le Z80 finira au moins d'exécuter l'instruction en cours avant de s'occuper de l'interruption. Avec un BUSRQ, au contraire, le Z80 réagit au battement suivant de son horloge interne - c'est-à-dire au cycle suivant - qu'il ait accompli l'instruction suivante ou non.

La réaction du Z80 à une interruption masquable dépend du mode actuel d'interruption. Dans tous les cas, il empêche en premier lieu des interruptions (masquables) supplémentaires, et sauvegarde le PC sur la pile. Il est toujours nécessaire de permettre à nouveau les interruptions (Ei) avant de sortir de vos programmes de service des interruptions.

MODES D'INTERRUPTION POUR INTERRUPTIONS MASQUABLES

Le mode implicite est le mode 0. Il peut être fixé a l'intérieur d'un programme en utilisant l'instruction:

IM 0 Met mode d'interruption 0

Avant de recevoir et d'accepter une interruption en mode 0, le Z80 attendra que le périphérique externe lui donne une instruction au cycle d'horloge suivant au moyen de sa ligne d'entrée générale, le "bus de données".

Cette instruction est généralement un appel (CALL) d'un programme gérant des interruptions que le programmeur a placé quelque part en mémoire. C'est aussi souvent une instruction de redémarrage (ReSTart):

RST n Relancer à adresse n

L'adresse n est une adresse d'un seul octet et se trouve obligatoirement parmi les adresses suivantes: &0, 88, &10, 818, 820, 828, &30, et 838, Comme RST est une instruction d'un seul octet, elle est souvent utilisée quand on a besoin d'une réponse rapide à l'interruption. Le problème avec RST est qu'il y a seulement de la place dans ces cases mémoires pour un saut à une autre adresse - particulièrement s'il va plusieurs programmes gérant des interruptions, chacun utilisé par des appareils différents utilisant des adresses RST différentes.

CALL et RST entraînent tous deux la sauvegarde du PC sur la pile. Le Z80 empêche des interruptions supplémentaires et commence la routine de service des interruptions. Si le programmeur a besoin de préserver les registres et les flags, le programme gérant les interruptions doit commencer à peu près comme ça:

PROGRAMME 9.1

```
PUSH AF
PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY
```

Et devrait finir à peu près comme ceci:

PROGRAMME 9.2

```
POP IY
POP IX
POP HL
POP DE
POP BC
POP AF
EI
RETI
```

Remarquez le "AF". C'est l'accumulateur et les flags. Le prochain mode d'interruption est le mode 1.

IM 1 Met mode d'interruption 1

Ce mode est identique au mode d'interruption 0 à la différence que le Z80 exécute automatiquement un RST &38 avant d'accepter une interruption masquable. De plus, le Z80 ignorera le contenu de son bus de données dans le cycle d'horloge suivant l'interruption. Le dernier mode est le mode 2:

IM 2 Met le mode d'interruption 2

Dans ce mode, après avoir fini l'instruction en cours, sauvegardé le PC et empêché des interruptions masquables supplémentaires, le Z80 sautera à n'importe quelle case mémoire d'adresse paire - c'est-à-dire dont le bit le moins significatif est 0. Les huit bits les plus significatifs devront "être mis à l'avance sur un registre spécial appelé registre de vecteur d'interruption, ou registre "I". Les huit bits les moins significatifs de l'adresse sont envoyés par le périphérique interrupteur. Cela permet au Z80 de sauter à n'importe laquelle des 128 adresses figurant dans une table en mémoire commençant à l'adresse octet fort figurant dans le registre I.

LES REGISTRES DE REMPLACEMENT

Vous êtes maintenant parfaitement au courant du fait que le Z80 a beaucoup de registres différents, en particulier. A, B, C, D, E, H, L, et les flags. Eh bien, le Z80 a un deuxième Jeu de ces registres particuliers, connu sous le nom de registres de remplacement, A', B', C', D', E', H', L' et les flags de remplacement F'.

Même s'ils peuvent être utilisés par le programmeur, il est déconseillé de les utiliser sur l'Amstrad.

L'utilisation de programmes intégrés les altérerait, comme le ferait n'importe quelle interruption. Les instructions pour les utiliser sont les suivantes:

EXX Intervertit les couples de registres

Cette instruction intervertit les contenus de BC, DE, et HL avec leurs remplaçants. Notez que toute instruction suivante utilisant par exemple DE utilisera maintenant le nouveau contenu de DE, qui était
auparavant dans DE'.

Si on utilise les registres de remplacement B' ou C', ils doivent être restaurés avant d'être basculés à nouveau sur le jeu de registres originel et de permettre les interruptions. Un programme pour utiliser les registres de remplacement ressemblera donc à ceci:

```
DI           Empêche les interruptions
EXX          Intervertit les couples de registres
PUSH BC      Sauvegarde nouveau BC
POP BC       Restaure BC
EXX         Revient sur registres initiaux
EI           Empêche les interruptions
```

Les programmes de ce genre doivent être courts, car diverses interruptions automatiques du système d'exploitation telles que les horloges, les examens du clavier etc. ont besoin des registres de remplacement, et qu'un programme trop long peut poser des problèmes. C'est vrai quelque soit le moment où on empêche les interruptions. (Pour plus de détails voir Soft 158).

Une autre instruction permet d'utiliser A' et F':

 EX AF,AF' Intervertit les contenus de AF et AF'

F' doit être restauré avant une nouvelle interversion. Encore une fois, les interruptions doivent être empêchées tandis qu'on utilise ces registres.

Notez qu'on utilise la même instruction pour réintervertir les registres - c'est-à-dire EX AF,AF' et non EX AF',AF.

D'AUTRES INSTRUCTIONS D'ECHANGE

Il y a deux autres formes de EX qui valent la peine d'être mentionnées:

EX DE,HL Intervertit les contenus de DE et de HL

et

```
EX (SP),HL Intervertit le haut de la   pile et le contenu de HL
EX (SP),IX Intervertit le haut de la   Pile et le contenu de IX
EX (SP),IY Intervertit le haut de la   pile et le contenu de IY
```

Le premier octet sur la pile va dans L et vice-versa, et l'octet suivant sur la pile va dans H (et vice versa). Procéder de la même façon avec IX et IY.

ARRET

HALT Arrête l'exécution du programme

Cette instruction arrête le Z80 jusqu'à ce qu'une Interruption soit reçue. La seule chose qu'elle fasse est de 'rafraîchir' la mémoire. Comme vous le savez, si vous coupez l'ordinateur, il oublie tous les programmes ou les données figurant dans sa mémoire. Ceci parce que la mémoire est une mémoire RAM dynamique ou Random Access Memory. Si la mémoire n'est pas mise à jour ou rafraîchie en permanence - c'est-à-dire si le voltage interne n'est pas corrigé en permanence, alors ces voltages disparaissent et les données stockées avec eux. Le Z80 utilise un registre de rafraîchissement qui lui dit quelle mémoire il doit rafraîchir et à quel moment.

LES ENTREES ET LES SORTIES DU Z80

Le Z80 a toute une famille d'instructions conçues pour permettre des entrées et des sorties a partir et vers des appareils externes. Encore une fois, ces instructions sont orientées électronique, et seront seulement expliquées brièvement ici.

IN A,(n) Entre le contenu du port n dans l'accumulateur

Un port est un connecteur avec un appareil externe. Tel numéro renvoie à tel port déterminé par l'électronique - c'est-à-dire par le câblage et par les circuits de l'ordinateur.

Le port, pour ce qui concerne l'ordinateur, est un magasin de données de huit bits.

IN A,(n) n'affecte aucun des flags.

Il existe une forme de l'instruction IN qui permet au contenu d'un port dont le numéro figure dans le registre C d'être placé dans n'importe lequel des registres A,B,C,D,E,H, ou L:

IN r,(C) Entre le contenu du port adressé par C dans registre r

Cette instruction annule le flag addition/soustraction. Le flag carry n'est pas affecté, mais les autres flags sont altérés suivant la valeur entrée.

Si elles sont nécessaires, le Z80 a d'autres formes de l'instruction IN qui mettent en parallèle LDI, LDIR etc.

La première est:

INI Entre le contenu du port adressé par C, décrémente B, incrémente HL

Avec cette instruction, B peut être utilisé comme compteur de boucle si c'est nécessaire. Notez, cependant, que B seulement est utilisé par INI, plutôt que BC comme dans LDI par exemple. C doit contenir le numéro de port, et HL la case mémoire dans laquelle la donnée sera placée. INI altère le signe, les flags demi-carry et parité/dépassement, et met le flag addition/soustraction. Le flag carry n'est pas affecté. Le flag zéro est mis quand B=1.

INIR Entre le contenu du port adressé par C, décrémente B, incrémente HL,
recommence jusqu'à ce que B=0

Cette fonction est équivalente à INI sauf qu'elle recommence automatiquement jusqu'à ce que B soit décrémenté jusqu'à zéro. Les flags sont affectés de la même manière qu'avec INI sauf que le flag zéro est toujours mis.

IND Entre le contenu du port adressé par C, décrémente B, décrémente HL

INDR Entre contenu du port adressé par C, décrémente B, décrémente HL, recommence jusqu'à ce que B=0

Ces instructions sont respectivement identiques à INI et INIR, sauf que HL est décrémenté au lieu d'être incrémente.

OUT (n),A Sort le contenu de l'accumulateur sur le port n

OUT (C),r Sort le contenu du registre r sur le port adressé par C

La dernière de ces deux instructions est différente de IN r,(C) par le fait qu'elle n'affecte aucun flag.

OUTI Sort sur le port adressé par C le contenu de la mémoire adressée par HL, décrémente B, incrémente HL

OTIR Sort sur le port adressé par C le contenu de la mémoire adressée par HL, décrémente B, incrémente HL, recommence jusqu'à ce que B=0

OUTD Sort sur le port adressé par C le contenu de la mémoire adressé par HL, décrémente B, décrémente HL

OTDR Sort sur le port adressé par C le contenu de la mémoire adressée par HL décrémente B, décrémente HL, recommence jusqu'à ce que B=0

Ces instructions affectent les flags de la même façon que les instructions IN correspondantes.

Une Instruction peu utile n'a pas encore été mentionnée:

NOP
NOP No Opération (Pas d'opération)

Elle dit au Z80 de ne rien faire pendant un cycle d'horloge. Cela peut "être très utile pour une bonne mise au point de la durée des boucles de temporisation. De même, si vous utilisez des adresses absolues plutôt que
des étiquettes symboliques, elle peut être utilisée pour compléter votre programme afin d'autoriser des erreurs d'adressage.

RESUME

Ce chapitre a couvert les instructions suivantes:

DI	EX AF,AF'
EI	EX DE,HL
RETN	EX (SP),HL
RETI	EX (SP),IX
IM0	EX (SP),IY
IM1	HALT
IM2	IN A,(n)
RST n	IN r,(C)
EXX	INI
OUI (n),A	INIR
DUT (C),r	IND
OUTI	INDR
OTIR	NOP
OUTD	
OTDR	

CHAPITRE 10

INSTRUCTIONS EXTERNES ET EXTENSIONS GRAPHIQUES

Ce chapitre explique comment des instructions supplémentaires peuvent être ajoutées à celles déjà connues du BASIC. Par exemple, une Instruction dessinant des cercles sera ajoutée au BASIC d'Amstrad. Ces programmes seront entièrement écrits en langage assembleur et Interfaces avec l'interpréteur BASIC en utilisant des instructions externes. Ce chapitre utilise certaines des propriétés les plus avancées de l'assembleur. Il est recommandé de lire d'abord l'annexe 6.

INSTRUCTION EXTERNE (RSX)

Toutes les instructions du BASIC (LIST, GOTO, RND etc.) sont dites "instructions internes". Quand un programme BASIC est exécuté et que l'interpréteur rencontre une instruction interne il la recherche dans la ROM (Read Only Memory = mémoire de lecture uniquement) tandis que, si une instruction externe est rencontrée, il recherche aussi dans la RAM (mémoire de lecture/écriture). Une instruction externe ou extension du système résident (RSX) est une chaîne de caractères alphanumériques précédée d'une barre verticale (Shift et @).

Voici les instructions externes légales:

|CIRCLE |TRIANGLE |BOX

Essayez de taper |BOX en BASIC. L'ordinateur répondra par "Unknown command". Cela parce que l'interpréteur ne peut trouver l'instruction BOX en ROM ni en RAM, Il est assez simple de dire à l'ordinateur qu'une Instruction externe existe; voici comment:

Phase 1

L'ordinateur a d'abord besoin de savoir que des instructions externes vont être ajoutées. On accomplit cela en créant une table d'instructions nouvelles. Cette table d'Instructions agit comme un bloc de sauts à des tables supplémentaires, apportant des informations explicites sur les nouvelles instructions, c'est-à-dire syntaxe, emplacement etc. Ces informations sont transmises au système d'exploitation,

en appelant un sous-programme en ROM, après avoir chargé dans les registres appropriés les données et l'adresse d'un buffer de quatre octets requis par le système d'exploitation.

Utilité de la routine: Fait enregistrer une commande externe par le système d'exploitation (c'est-à-dire dit au système que l'instruction existe.

Adresse d'appel: &BCD1

Conditions d'entrée: BC= adresse mémoire de départ de la table d'instructions externes

HL= adresse mémoire de départ du buffer de quatre octets.

Maintenant, toute cette procédure va être illustrée par un exemple. La table d'instructions externes commencera a l'adresse mémoire représentée par l'étiquette EXCOMT (EXternal COMmand Table). La méthode la plus facile pour réserver quatre octet pour un buffer est d'utiliser la directive d'assembleur "DEFA". On affectera à ce buffer le label BUFF:

Donc:-

```
BUFF: DEFS &04
```

En combinant ces différents éléments:

PROGRAMME 10.1(ne pas entrer tout de suite)

```
BUFF: DEFS &04          Installe buffer
LD BC,EXCOMT:           BC=départ de table
LD HL,BUFF:              HL=départ de buffer
CALL &BCD1              Table d'enregistrement
RET                     Retour au BASIC.
```

N'entrez pas ce programme tout de suite car comme l'ordinateur connaît l'emplacement de la table d'instructions externes il ne trouverait pas de nom d'instruction nouvelle en la parcourant, Dès lors la table d'instructions externes nouvelles doit être remplie avec des noms d'instruction nouveaux.

Phase 2

Des détails supplémentaires sur la table d'instructions nouvelles doivent figurer dans une table additionnelle. L'adresse de cette table est donnée par 1'étiquette NENAM (NEw NAMe). La première ligne de cette section est donc:-

```
EXCOMT: DEFW NENAM:
```

On ajoute alors les nouveaux noms d'instruction en utilisant les instructions JP comme suit.

```
JP BOX:
```

Cela affectera une adresse de saut à la nouvelle instruction BOX. En admettant qu'une seule instruction externe va être ajoutée, la deuxième partie du programme apparaît comme ci-dessous:

```
EXCOMT: DEFW NENAM: JP BOX:
```

Maintenant, tout ce dont on a besoin est d'ajouter les caractéristiques du nom de la nouvelle instruction.

Phase 3

Le nom de la nouvelle instruction est entré comme une chaîne de caractères ASCII en commençant par l'adresse mémoire désignée par NENAM. Le système d'exploitation de l'Amstrad a besoin que le bit 7 soit mis dans l'octet représentant le dernier caractère ASCII de la chaîne du nom de l'instruction. Il est très facile d'accomplir cela en ajoutant 880 au code ASCII de la dernière lettre, Dès lors, dans l'exemple, &80 doit être ajouté à l'ASCII de "X". Cette partie du programme se présente donc comme ci-dessous:

```
NENAM: DEFM  "BO"
       DEFB  "X"+880 DEFB  80
```

Le 80 indique la fin de la table.

Combiner ces trois parties ensembles:

PROGRAMME 10.2 (ne pas assembler tout de suite)

```
            ORG 40000
BUFF:       DEFS &04
            LD BC,EXCOMT:
            LD HUBUFF:
            CALL &BCD1
            RET
EXCOMT:     DEFW NENAM :
            JP BOX:
NENAM:      DEFM "BO"
            DEFB "X"+&80
            DEFB 80
```

Notez que ORG a été mis sur 40000. En mettant alors MEMORY sur 39999, empêchant ainsi le BASIC d'utiliser une case mémoire au-dessus de 39999, l'instruction externe ne peut être altérée.

Quand on tapera ùBOX, a partir du BASIC, l'ordinateur sautera à la case mémoire adressée par l'étiquette BOX.

Pour illustrer la nouvelle instruction externe tapez le Programme 10.3 immédiatement après le programme 10.2.

PROGRAMME 10.3

```
BOX:  LD A,66
            CALL PRINT:
            LD A,79
            CALL PRINT:
            LD A,88
            CALL PRINT:
```

```
            RET
            PRINT: EQU &BB5A
```

Remarquez l'utilisation de l'étiquette mémoire PRINT

Maintenant assemblez le programme entier; une fois listé il apparaîtra comme ci-dessous.

```
            ORG 10000
BUFF:       DEFS 804
            LD BC,EXCOMT:
            LD HL,BUFF:
            CALL &BCD1
            RET
EXCOMT:     DEFW NENAM:
            JP BOX:
NENAM:      DEFN "BO"
            DEFB "X"+&80
            DEFB &0
BOX:  LD A,66
            CALL PRINT:
            LD A,79
            CALL PRINT:
            LD A,88
            CALL PRINT:
            RET
PRINT:      EQU &BB5A
```

Exécutez le programme avec un 'CALL 40000' à partir du BASIC. Bien que rien de visible ne se soit produit, l'instruction externe BOX a été prise en charge Par le système d'exploitation de l'Amstrad. Revenez en BASIC et tapez:

```
ùBOX
```

BOX apparaît sur l'écran! Tout programme en langage assembleur adressé avec l'étiquette BOX peut donc être appelé avec cette instruction externe. On peut par exemple dessiner un rectangle sur l'écran, en utilisant les sous-programmes de graphisme intégrés. Pour préciser la taille du rectangle. Il faut transmettre certains paramètres à l'assembleur. Considérons le diagramme suivant:

FIGURE 10.1

En définissant deux angles opposés diagonalement, on peut décrire un rectangle unique. Il serait utile si, de même que la taille, on pouvait changer la couleur du rectangle. C'est donc cinq paramètres qu'il faut transmettre au sous-programme en langage assembleur. Une solution envisageable serait d'écrire les données dans un bloc de mémoire, créant un "bloc de données". Bien qu'encombrant, cela marcherait. Heureusement, le système d'instructions externes crée ce bloc de données pour nous, Quand elle est appelée, la case de départ du bloc de données est stockée dans le couple de registres IX, et le nombre d'éléments de données est stocké dans l'accumulateur. Considérons l'instruction suivante qui dessinera un rectangle de 100 Par 100 unités sur l'écran en 200,100 avec ink 2

```
ùBOX, 200,100,300,200,2
```

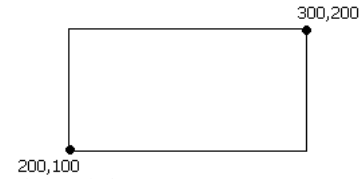


FIGURE 10.2

Les paramètres du rectangle sont convertis en hexadécimal avant d'être stockés en mémoire.

Phase 1

```
100 = &0064
200 = &00C8
300 = &012C
  2 = &0002
```

Phase 2

Les paramètres sont stockés avec l'octet faible en premier suivi immédiatement par l'octet fort. Le dernier paramètre entré est désigné par IX, les paramètres de l'exemple sont donc stockés comme suit:

Case mémoire	Contenu Hex	Décimal
IX+0	02	2
IX+1	00	
IX+2	00	200
IX+3	C8	
IX+4	01	300
IX+5	2C	
IX+6	00	100
IX+7	64	
IX+8	00	200

IX+9 C8

En utilisant un décalage approprié on peut avoir accès a n'importe lequel de ces paramètres. Remarquez que chaque paramètre est stocké sur deux octets, un paramètre peut se trouver dans l'intervalle 0-65535.

Maintenant, des coordonnées transmises par l'instruction externe, quatre couples de coordonnées doivent être dérivées pour décrire le rectangle.

Considérons la Figure 10.3:

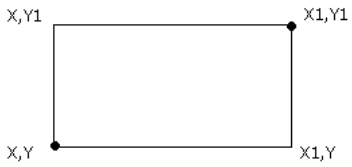


FIGURE 10.3

Comme les coordonnées X,Y et X1,Y1 sont transmises par l'Instruction externe on doit en déduire les coordonnées des deux autres angles et tracer le rectangle. Ceci est représenté par l'organigramme suivant:

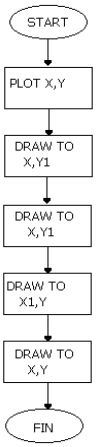


FIGURE 10.4

Exprimons cet organigramme dans un programme (remarquez l'utilisation de la Pile pour stocker les coordonnées):

PROGRAMME 10.4

```
10     DRAW:      EQU   &BBF6
20     PLOT:      EQU   &BBEA
30     INK :      EQU   &BBDE
40     BOX: LD    A,(IX+0)
50          CALL  INK:
60          LD    D,(IX+8)
70          LD    D,(IX+9)
80          PUSH  DE
90          LD    E,(IX+6)
100         LD    D,(IX+7)
110         PUSH  DE
120         LD    E,(IX+4)
130         LD    D,(IX+5)
140         PUSH  DE
150         LD    E,(IX+2)
160         LD    D,(IX+3)
170         PUSH  DE
180         LD    E,(IX+8)
190         LD    D,(IX+9)
200         LD    E,(IX+6)
210         LD    H,(IX+7)
220         PUSH  DE
230         CALL  PLOT:
240         POP   DE
250         POP   HL
260         PUSH  HL
270         CALL  DRAW:
280         POP   HL
290         POP   DE
300         PUSH  DE
310         CALL  DRAW:
320         POP   DE
330         POP   HL
340         PUSH  HL
350         CALL  DRAW:
360         POP   HL
370         POP   DE
380         CALL  DRAW:
390         RET
```

L'utilisation des étiquettes pour les adresses de mémoire augmente grandement la lisibilité du programme. Maintenant remplacez le sous-programme commençant avec l'étiquette BOX dans le programme précédent par le programme 10.4.

Assemblez-le et exécutez-le.

Maintenant revenez en BASIC et essayez le programme 10.5, après avoir intégré l'instruction avec un "CALL 40000".

Remarque: Comme l'assembleur utilise les lignes BASIC au-dessus de 64000, ne tapez pas NEW ou vous devrez recharger l'assembleur.

PROGRAMME 10.5

```
10 MODE O:CALL 40000
20 X=RND(1)*550
30 Y=RND(1)*350
40 X1=RND(1)*50
50 Y1=RND(1)*50
60 C=RND(1)*15
70 ùBOX,X,Y,X+X1,Y+Y1,C
80 GOTO 20
```

Ceci conclut la partie sur les instructions externes. Le reste de ce chapitre expliquera comment on peut ajouter des instructions graphiques additionnelles.

QUELQUES INSTRUCTIONS GRAPHIQUES SUPPLEMENTAIRES

1.BOX,X,Y,X1,Y1,C

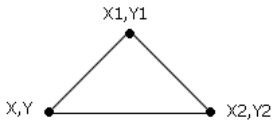
Dessine un rectangle sur l'écran en utilisant le PEN C

2. BOXF,X,Y,X1,Y1,C

où les arguments sont exactement les mêmes que pour BOX. La seule différence est que le rectangle est plein.

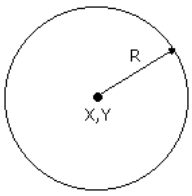
3. TRI,X,,Y,X1,Y1,X2,Y2,C

Dessine un triangle sur l'écran. Avec:



Notez que Y doit être égal à Y2

4. CIRCLE,X,Y,R,C Avec:



Les instructions externes résident en mémoire en commençant à la case mémoire 40000. Pour s'assurer qu'elles ne soient pas altérées on ramène le pointeur de mémoire BASIC sur 39999 avec l'instruction suivante:

MEMORY 39999

Il est donc sage de s'assurer que les deux premières lignes de programme BASIC utilisant ces instructions sont les suivantes:

```
1 MEMORY 39999
2 CALL 40000
    :
```

Les instructions de remplissage du rectangle

Ces instructions dessinent un rectangle plein sur l'écran; il est dessiné comme une série de lignes horizontales consécutives.

Considérons la figure 10.5

FIGURE 10.5

L'algorithme pour remplir ce rectangle se présente ainsi:

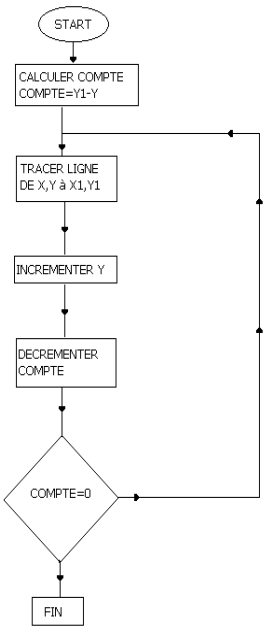


FIGURE 10.6
Exprimons cet algorithme dans un programme

PROGRAMME 10.6

```
10     COUNT:  DEFS 4
20     DRAW:   EQU      &BBF6
30     PLOT:   EQU      &BBEA
40     INK:    EQU      &BBDE
50     BOXF:   LD A,(IX+0)
60               CALL INK:
70                LD    L,(IX+2)
80                LD    H,(IX+3)
90                LD    E,(IX+6)
100               LD    D,(IX+7)
110               AND   A
120               SBC   HL,DE
130               JP    C,END:
140               LD    (COUNT:),HL
150               LD    E,(IX+8)
160               LD    D,(IX+9)
170               LD    L,(IX+6)
180               LD    H,(IX+7)
190               PUSH  HL
200               PUSH  DE
210               CALL  PLOT:
220               LD    E,(IX+4)
230               LD    D,(IX+5)
240               POP   IX
250               POP   IY
260    NXT:        PUSH DE
270               PUSH  IY
280               POP   HL
290               CALL  DRAW:
300               PUSH  IX
310               POP   DE
320               POP   IX
330               INC   IY
340               LD    HL,(COUNT:)
350               PUSH  DE
360               LD    DE,1
370               AND   A
380               SBC   HL,DE
390               LD    (COUNT:),HL
400               POP   DE
410               JR    NZ,NXT:
420    END:       RET
```

La cassette d'assembleur (face B) contient un fichier appelé GRAPHICS-EXT qui contient toutes les instructions graphiques additionnelles présentées dans ce chapitre, Pour l'utiliser chargez l'assembleur puis chargez le fichier GRAPHICS-EXT et assemblez-le. Si vous désirez utiliser une instruction supplémentaire dans vos programmes BASIC, sauvegardez une copie du code objet produit en utilisant l'identificateur de fichier '-b'. Sauvegardez-la par exemple sous le nom GRAPHICS-B, Ce fichier peut alors être chargé a partir du BASIC avec l'instruction : LOAD "GRAPHICS-B",40000.

Assurez-vous cependant que le pointeur de mémoire BASIC a d'abord été ramené sur 39999, en utilisant a la suite l'instruction MEMORY 39999. Ensuite pour intégrer les instructions graphiques, tapez CALL 40000 à partir du BASIC.

Maintenant toutes les instructions sont disponibles. Etudiez le programme de présentation des instructions supplémentaires GRAPHICS-DEMO si vous n'êtes pas sûr de vous.

L'INSTRUCTION TRIANGLE

Cette instruction produit un triangle plein sur l'écran. La méthode utilisée est itérative et est choisie pour sa simplicité plutôt que son efficacité. Dans cette méthode une procédure donnée est répétée jusqu'à ce qu'une condition soit remplie. Ici la différence entre deux coordonnées de la base X est utilisée comme compteur. Voir la figure 10.7

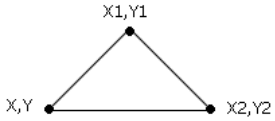


FIGURE 10.7

Une série de lignes sont dessinées du sommet (X1,Y1) à la base. Cela se traduit par le fait que le pixel (point) du sommet est réécrit un nombre de fois égal au compteur. Les pixels à proximité du sommet sont aussi réécrits, mais moins fréquemment. C'est un algorithme très peu efficace; l'idéal serait que chaque pixel du triangle soit écrit une fois. Cela peut être accompli en utilisant des algorithmes plus complexes (par exemple un algorithme scan-line), Mais ceci dépasse le cadre de ce livre.

Voici le listing de l'instruction triangle:

```
10 COUNT:         DEFS 4
20 DRAW:          EQLI  &BBF6
30 PLOT:          EQU   &BBEA
40 INK:           EQU   &BBDE
50 TPI:           LD    A,(IX+0)
60                CALL  INK:
70                LD    E,(IX+12)
80                LD    D,(IX+13)
90                LD    L,(IX+4)
100               LD    H,(IX+5)
110               AND   A
120               SBC   HL,DE
130               JP    M,END:
140               LD    (COUNT:),HL
150               LD    E,(IX+8)
l60               LD    D,(IX+9)
170               LD    L,(IX+6)
130               LD    H,(IX+7)
190               PUSH  DE
200               PUSH  HL
210               LD    L,(IX+10)
220               LD    H,(IX+11)
230               PUSH  HL
240               POP   IY
250               POP   IY
2*0               LD    E,(IX+12)
270               LD    D,(IX+13)
230               PUSH  DE
290               POP   IX
500   NXTT:       POP   HL
310               POP   DE
320               PUSH  DE
330               PUSH  HL
340               CALL  PLOT:
350               PUSH  IX
360               POP   DE
370               PUSH  IY
330               POP   HL
390               CALL  DRAW:
400               INC   IX
410               LD    HL,(COUNT:)
420               LD    DE,1
430               AND   A
4d0               SBC   HL,DE
450               LD    (COUNT:),HL
460               JR    NZ,NXTT:
470               POP   DE
4S0               POP   DE
490   END:        RET
```

L'instruction CIRCLE

La méthode la plus simple pour réaliser un cercle sur l'écran utilise les fonctions SIN et COS de l'Amstrad. Considérons la figure 10.8:

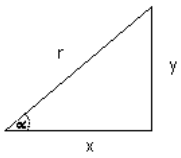


FIGURE 10.8

En incrémentant dans l'intervalle 0<alpha<pi/2 et en fixant un point aux coordonnées (X,Y) on obtient un quart de cercle. Ce quart peut être développé en un cercle plein. En admettant que le centre du cercle est le point d'origine du graphique, considérons la figure 10,9 pour le point X,Y.

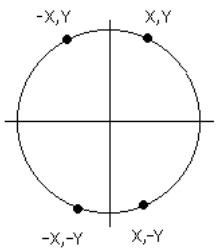
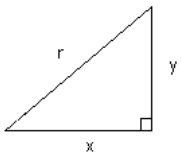


FIGURE 10.9

Même si cette méthode fonctionne, elle est pratiquement inefficace parce que les fonctions SIN et COS nécessitent un temps de calcul

considérable qui ralentit l'algorithme. Un algorithme légèrement plus efficace peut-être dérivé du théorème de Pythagore, équation 10.1 ci-dessous:

Théorème de Pythagore



R^2 = X^2 + Y^2 Equ 10.1

Réajustons l'équation 10.1:

Pour dessiner un quart de cercle comme auparavant, X doit être incrémenté dans l'intervalle 0<=X<=R, on trouve alors le Y correspondant à chaque valeur de X, et le Pixel de coordonnées X,Y est mis.

Essayez le programme BASIC suivant:

PROGRAMME 10.8

```
10 MODE 0
20 DEFINTx,y,r
30 x=0
40 r=100
50 WHILE X<r
60 y=SQR(r*r-x*x)
70 PLOT x,y
80 x=x+1
90 WEND
100 END
```

Ce programme réalise un quart de cercle comme ci-dessous dans la figure 10.10

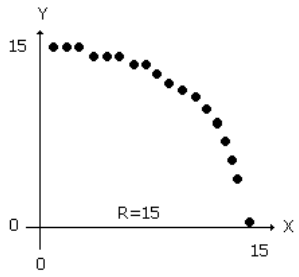


FIGURE 10.10

Il y a deux limites principales à l'utilisation de cette méthode.

1. La qualité de l'arc laisse à désirer à mesure que X s'approche de R.

2. Bien que plus rapide que SIN ou COS, la recherche de la racine carrée d'un nombre demande quand même un temps de calcul considérable.

Il y a différentes méthodes qui permettent d'amoindrir ces problèmes. Dans ce chapitre nous nous attarderons sur une solution possible. Elle est basée sur l'algorithme de Bresenham développé à l'origine pour les traceurs mécaniques. Cet algorithme est considérablement plus efficace que toutes les méthodes mentionnées précédemment car toutes les opérations arithmétiques peuvent être accomplies facilement grâce à quelques additions, soustractions et décalages.

UNE ADAPTATION DE L'ALGORITHME DE BRESENHAM POUR DESSINER LES CERCLES

Au lieu d'incrémenter X dans l'intervalle 0<=X<=R cette méthode utilise l'intervalle 0<=X<=pi/4 produisant donc un segment à 45 degrés. On obtient le cercle complet en réfléchissant les points calculés. Le coeur de l'algorithme est un sous-programme qui sélectionne le pixel le plus proche du cercle véritable au point considéré. La distance entre le cercle véritable et le pixel sélectionné est appelée "terne d'erreur" et est déduite de la manière suivante:-

Utilisons le théorème de Pythagore

R^2 = x^2 + y^2

admettons que le pixel est fixé en X,Y, le terme d'erreur est alors donné par

E= (x^2+y)-r^2

En diminuant E à chaque phase, on obtient l'approximation d'un cercle la plus proche possible sur une grille de pixels discontinue.

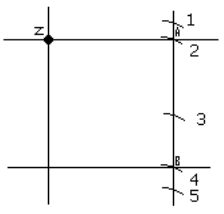


FIGURE 10.11

Si on admet que le pixel noir Z vient d'être mis, le prochain pixel à "être mis sera soit A, soit B. Si on définit le terme d'erreur comme

étant la différence entre le carré des distances du centre du cercle aux pixels A ou B et au cercle actuel à ce point, on peut calculer les équations suivantes.

Pour Pixel A

E =(x^2+y^2)-r^2 Equ 10,2

Pour pixel B

E =(x^2-y^2)-r^2 Equ 10.3

Donc si la valeur absolue de E de A est supérieure ou égale à la valeur absolue de E de B, c'est le pixel B qui est mis et si la valeur absolue de E de A est inférieure à la valeur absolue de E de B, c'est le pixel A qui sera mis.

Combinons les équations 10.2 et 10.3 pour former le terme d'erreur totale E:

E = E + E
 T A B

Maintenant si E >=0, le pixel B est mis, sinon E <0 et le Pixel A est mis.
 T T

Reconsidérons la figure 10.11

Case 1
E <0 le pixel A est mis T
 T

Case 2
E <0 le pixel A est mis T
 T

Case 3
E <0 le pixel A est mis T
 T

Case 4
E >=0 le pixel B est mis
 T

Case 5
E >=0 le pixel B est mis T

Comme on peut le voir la méthode fonctionne, cependant il est quand même nécessaire de calculer le carré et la racine carré des données pour calculer le terme d'erreur. Grâce a une série d'opérations arithmétiques, on peut montrer que l'erreur initiale se présente ainsi:

E = 3-2r Equ 10.4
 T

La valeur de E(T) change tout au long du programme suivant le choix du pixel précédent:

Si le Pixel A est choisi quand E <0 alors le nouveau E est donné par
 T T

E = E +4X+6 Equ 10.5
 T+1 T

ou si le pixel B est sélectionné quand E >=0 alors
 T

E = E +4(x-y)+10 Equ 10.7
 T+1 T

L'équation 10,5 a besoin de deux additions et deux décalages, l'équation 10.6 de deux additions, une soustraction et deux décalages; un progrès considérable sur les algorithmes précédents. On a besoin d'une méthode pour réfléchir ces points afin de former un cercle complet. Si on peut trouver les valeurs de X et Y pour un quart de cercle, on peut alors produire un cercle complet en réfléchissant ces coordonnées. Voir figure 10.12

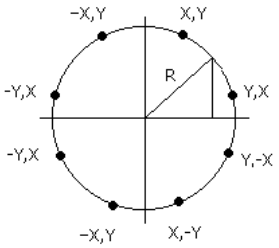


FIGURE 10.12

Cet algorithme est démontré dans le programme BASIC suivant:

PROGRAMME 10.9

```
10 MODE 1
20 radius=100
30 x=0
40 y=radius
50 origin 320,200
60 diff=3-2*radius
70 while x<y
80 gosub 50
```

```
90 if d<0 then d=d+4*x+6:goto 120
100 d=d+4*(x-y)+10
110 y=y-1
120 x=x+1
130 wend
140 end
150 plot x,y
160 plot y,x
170 plot y,-x
180 plot x,-y
190 plot x,-y
200 plot -x,-y
210 plot -y,-x
220 plot -y,x
230 plot -x,y
240 return
```

Le programme produit une distribution des pixels comme dans la figure 10.13, pour un rayon de 15.

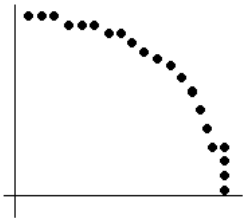


FIGURE 10.13

Ici l'algorithme est converti en langage assembleur: Pour économiser vos doigts, toutes les instructions externes sont incluses dans la cassette sous le nom de GRAPHICS-EXT.

PROGRAMME 10.10

```
10     PLOT:       EQU &BBEA
20     INK:        EQU &B6DE
30     ORGIN:      EQU SBBCC
40     ORGET:      EQU &BBCC
50     DIFF:       DEFS 2
60     RAD:        DEFS 2
70     X:          DEFS 2
80     Y:          DEFS 2
90     XO:         DEFS 2
100    YO:         DEFS 2
ne     CIRCLE:     CALL ORGET:
120               LD(XO:),DE
130               LD A,(IX+0)
140               CALL INK:
150               LD (YO:),HL
160               LD D,(IX+7)
170               LD E ,(IX+6)
180               LD H,(IX+5)
190               LD L,(IX+4)
200               PUSH HL
210               LD H,(IX+3)
220               LD L,(IX+0}
230               LD (RAD:),HL
240               POP HL
250               CALL ORGIN:
260               LD BC,0000
270               LD (X:),BC
2S0               LD HL,(RAD:)
290               LD (Y:),HL
300               SLA L
310               RL H
320               PUSH HL
330               POP DE
340               LD HL,3
350               XOR A
360               SBC HL,DE
370               LD (DIFF),HL
380 CALC:         LD HL,(X:)
390               LD DE,(Y:)
400               PUSH HL
410               PUSH DE
420               CALL MIRIM:
430               POP DE
440               POP HL
450               XOR A
460               SBC HL,DE
470               JP P,END2:
430               LD HL,(DIFF:)
490               LD BC,0000
500               SBC HL,BC
510               JP P,LESS:
520               LD DE,(X:)
530               SLA E
540               RL D
550               SLA E
560               RL D
570               LD HL,6
530               ADD HL,DE
590               LD DE,(DIFF:)
600               ADD HL,DE
610               JP NXT3:
620 LESS:         LD HL,(IX:)
630               LD DE,(Y:)
```

```
640               XOR A
650               SBC HL,DE
660               SLA L
670               RL H
680               SLA L
690               RL   H
700               LD   DE, 10
710               ADD  HL,DE
720               LD  DE,(DIFF:)
730               ADD  HL,DE
740               LD   DE,(Y:)
750               DEC  DE
760               LD  (Y:),DE
770 NXT3:         LD (DIFF:),HL
780               LD HL,(X:)
790               INC HL
800               LD (X:),HL
810              JP CALC:
820 MIRIM:        LD DE,(X:)
830               LD HL,(Y:)
840              CALL PLOT:
850               LD DE,(Y:)
860               LD HL,(X:)
870               CALL PLOT:
880               LD HL,0000
890               LD BC,(X:)
900               XORA
910               SBC HL,DE
920               PUSH HL
930               PUSH HL
940               LD DE,(Y:)
950               CALL PLOT:
960               POP DE
970               LD HL,(Y:)
930               CALL PLOT:
990               LD HL,0000
1000              LD BC,(Y:)
1010              XOR A
1020              SBC  HL,BC
1030              PUSH HL
1040              PUSH HL
1050              LD DE,(X:)
1060              CALL PLOT:
1070              POP DE
1080              LD HL,(X:)
1090              CALL PLOT:
1100              POP HL
1110              POP DE
1120              PUSH HL
1130              PUSH DE
1140              CALL PLOT:
1150              POP HL
1160              POP DE
1170              CALL PLOT:
1180              RET
1190   END2:            LD DE,(X0:)
1200              LD HL,(Y0:)
1210              CALL ORIGIN:
1220              RET
```

Ça y'est, vous y êtes arrivé. Bien Joué. Nous espérons que le livre vous a plu et que sa lecture vous a été profitable. Joyeuse programmation !

A N N E X E 1

Le jeu d'instructions du Z80

Abréviations utilisées dans les tables suivantes:

Registres

```
r,r'       = indique n'importe lequel des   registres A,B,C,D,E,H,L.
dd         = indique n'importe lequel des   coudes de registres BC,DE,HL,SP
qq         = indique n'importe lequel des   couples de registres AF,BC,DE,HL
pp         = indique n'importe lequel des   couples de registres BC,DE,IX,SP
rr         = indique n'importe lequel des   couples de registres HC,DE,IX,SP
e          = indique un (décalage) offset   de complément à deux
s          = indique soit r,n,(HL), (IX+d), IY+d) ou dans une instruction
             d'index un (décalage) offset   de complément à deux
H          = Octet fort: L=0ctet faible
S          = Où b=bit 0 7
```

Modes d'Adressage

```
RR         = Registre - Registre
Im         = Immédiat
IDX        = Indexé
D          = Direct
In         = Indirect
```

Flags

```
C           = Carry / flag de report
Z           = Flag zéro
S           = Flag signe
P/V         = Flag parité ou dépassement
H           = Flag demi-carry
N           = Flag addition/soustraction
```

Modification des flags

? Flag fixé en fonction du résultat de 1'opération
0 Flag annulé
1 Flag mis
* Flag non affecté
- Modification du flag imprévisible
V Flag mis si dépassement (overflow)
P Flag mis si parité
F Le flag P/V reçoit le contenu du flip-flop d'interruption (IFF)

Le flag P/V

Si l'opération indiquée par un V dans la colonne P/V se traduit par un dépassement, le flag V sera mis (1), sinon il sera annulé (0). Si l'opération a pour but de tester la parité, ce qui est indiqué par un P, le flag sera mis si la parité est paire et annulé si la parité est impaire.

Modes d'adressage

LD r,r' Registre registre
LD r,n Immédiat
LD r,(IX+d) Indexé
LD r,(nn) Direct
LD r,(dd) Indirect

Où

r ou r' est un registre 8 bits
n est un registre 8 bits
d est un décalage de complément à
nn est un nombre 16 bits
dd est BC, DE, HL ou SP.

GROUPE LOAD 8 BITS

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
LD r,r'	r ←r'	1	1	RR	*	*	*	*	*	*
LD r,n	r ←n	2	2	Im	*	*	*	*	*	*
LD r,(HL)	r ←(HL)	1	2	In	*	*	*	*	*	*
LD r,(IX+d)	r ←(IX+d)	3	5	IDX	*	*	*	*	*	*
LD r,(IY+d)	r ←(IY+d)	3	5	IDX	*	*	*	*	*	*
LD (HL),r	(HL)←r	1	2	In	*	*	*	*	*	*
LD (IX+d),r	(IX+d)←r	3	5	IDX	*	*	*	*	*	*
LD (IY+d),r	(IY+d)←r	3	5	IDX	*	*	*	*	*	*
LD (HL),n	(HL)←n	2	3	In	*	*	*	*	*	*
LD (IX+d),n	(IX+d)←n	4	5	IDX	*	*	*	*	*	*
LD (IY+d),n	(IY+d)←n	4	5	IDX	*	*	*	*	*	*
LD A,(BC)	A←(BC)	1	2	In	*	*	*	*	*	*
LD A,(DE)	A←(DE)	1	2	In	*	*	*	*	*	*
LD A,(nn)	A←(nn)	3	4	D	*	*	*	*	*	*
LD (BC),A	(BC)←A	1	2	In	*	*	*	*	*	*
LD (DE),A	(DE)←A	1	2	In	*	*	*	*	*	*
LD (nn),A	(nn)←A	3	4	D	*	*	*	*	*	*
LD A,I	A←I	2	2	RR	*	?	F	?	0	0
LD A,R	A←R	2	2	RR	*	?	F	?	0	0
LD I,A	I←A	2	2	RR	*	*	*	*	*	*
LD R,A	R←A	2	2	RR	*	*	*	*	*	*

GROUPE LOAD 16 BITS

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
LD dd,nn	dd←nn	3	3	Im	*	*	*	*	*	*
LD IX,nn	IX←nn	4	4	Im	*	*	*	*	*	*
LD IY,nn	IY←nn	4	4	Im	*	*	*	*	*	*
LD HL,(nn)	H←(nn+1)	3	5	D	*	*	*	*	*	*
	L←(nn)				*	*	*	*	*	*
LD dd,(nn)	dd←(nn+1)	4	6	D	*	*	*	*	*	*
	dd←(nn)				*	*	*	*	*	*
LD IX,(nn)	IX←(nn+1)	4	6	D	*	*	*	*	*	*
	IX←(nn)				*	*	*	*	*	*
LD IY,(nn)	IY←(nn+1)	4	6	D	*	*	*	*	*	*
	IY←(nn)				*	*	*	*	*	*
LD (nn),HL	(nn+1)←H	3	5	D	*	*	*	*	*	*
	(nn)←L				*	*	*	*	*	*
LD (nn),dd	(nn+1)←dd	4	6	D	*	*	*	*	*	*
	(nn)←dd				*	*	*	*	*	*
LD (nn),IX	(nn+1)←IX	4	6	D	*	*	*	*	*	*
	(nn)←IX				*	*	*	*	*	*
LD (nn),IY	(nn+1)←IY	4	6	D	*	*	*	*	*	*
	(nn)←IY				*	*	*	*	*	*
LD SP,HL	SP←HL	1	1	RR	*	*	*	*	*	*
LD SP,IX	SP←IX	2	2	RR	*	*	*	*	*	*
LD SP,IY	SP←IY	2	2	RR	*	*	*	*	*	*
PUSH qq	(SP-1)←qq	1	3	Im	*	*	*	*	*	*
	(SP-2)←qq				*	*	*	*	*	*
PUSH IX	(SP-1)←IX	2	4	Im	*	*	*	*	*	*
	(SP-2)←IX				*	*	*	*	*	*
PUSH IY	(SP-1)←IY	2	4	Im	*	*	*	*	*	*
	(SP-2)←IY				*	*	*	*	*	*
POP qq	qq←(SP)	1	3	Im	*	*	*	*	*	*

	qq←(SP+1)				*	*	*	*	*	*
POP IX	IX←(SP)	2	4	Im	*	*	*	*	*	*
	IX←(SP+1)				*	*	*	*	*	*
POP IY	IY←(SP)	2	4	Im	*	*	*	*	*	*
	IY←(SP+1)				*	*	*	*	*	*

GROUPE ECHANGE – GROUPE TRANSFERT – RECHERCHE DE BLOC

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
EX DE,HL	DE←→HL	1	1	RR	*	*	*	*	*	*
EX AF,AF'	AF←→AF'	1	1	RR	*	*	*	*	*	*
EXX	(BC (BC' DE ←→ DE' HL) HL')	1	1	RR	*	*	*	*	*	*
EX (SP),HL	H←→(SP+1) L←→(SP)	1	5	RR	*	*	*	*	*	*
EX (SP),IX	IX←→(SP+1) IX←→(SP)	2	6	RR	*	*	*	*	*	*
EX (SP),IY	IY←→(SP+1) IY←→(SP)	2	6	RR	*	*	*	*	*	*
LDI	(DE)←(HL) DE←DE+1 HL←HL+1 BC←BC-1	2	4	In	*	*	?	*	0	0
LDIR	(DE)←(HL) DE←DE+1 HL←HL+1 BC←BC-1 JUSQUE BC=0	2	5	In	*	*	0	*	0	0
LDD	(DE)←(HL) DE←DE-1 HL←HL-1 BC←BC-1	2	4	In	*	*	?	*	0	0
LDDR	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 JUSQUE BC=0	2	5	In	*	*	0	*	0	0
CPI	A-(HL) HL←HL+1 BC←BC-1	2	4	In	*	?	?	?	?	?
CPIR	A-(HL) HL←HL+1 BC←BC-1 JUSQUE A=(HL) OU BC=0	2	5	In	*	?	?	?	?	?
CPD	A-(HL) HL←HL-1 BC←BC-1	2	4	In	*	?	?	?	?	?
CPDR	A-(HL) HL←HL-1 BC←BC-1 JUSQUE A=(HL) OU BC=0	2	5	In	*	?	?	?	?	?

GROUPE ARITHMETIQUE ET LOGIQUE 8 BITS

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
ADD A,r	A←A+r	1	1	Im	?	?	V	?	0	?
ADD A,n	A←A+n	2	2	Im	?	?	V	?	0	?
ADD A,(HL)	A←A+(HL)	1	2	In	?	?	V	?	0	?
ADD A,(IX+d)	A←A+(IX+d)	3	5	IDX	?	?	V	?	0	?
ADD A,(IY+d)	A←A+(IY+d)	3	5	IDX	?	?	V	?	0	?
ADC A,s	A←A+s+CY			Im	?	?	V	?	0	?
SUB A,s	A←A-s			Im	?	?	V	?	1	?
SBC A,s	A←A-s-CY			Im	?	?	V	?	1	?
AND s	A←A s			Im	0	?	P	?	0	1
OR s	A←A s			Im	0	?	P	?	0	1
XOR s	A←A,s			Im	0	?	P	?	0	1
CP s	A-s			Im	?	?	V	?	1	?
INC	r-r+1	1	1	Im	*	?	V	?	0	?
INC(HL)	(HL)←(HL)+1	1	3	In	*	?	V	?	0	?
INC(IX+d)	(IX+d)←(IX+d)+1	3	6	IDX	*	?	V	?	0	?
INC(IY+d)	(IY+d)←(IY+d)+1	3	6	IDX	*	?	V	?	0	?
DEC d	d←d-1			Im	*	?	V	?	1	1

GROUPE ARITHMETIQUE GENERALE ET CONTROLE UNITE CENTRALE

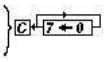
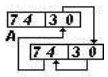
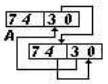
Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
DAA cnvertit A en BCD arès addition ou sous-traction sur oper. BCD		1	1	Im	?	?	P	?	*	*
CPL	A←$\bar{A}$	1	1	Im	*	*	*	*	1	1
NEG	A←0-A	2	2	Im	?	?	V	?	1	?

CCF	CY←CY	1	1	Im	?	*	*	*	0	*
SCF	CY←1	1	1	Im	1	*	*	*	0	0
NOP	Pas d'opération	1	1	Im	*	*	*	*	*	*
HALT	CPU halte	1	1	Im	*	*	*	*	*	*
DI	IFF←0	1	1	Im	*	*	*	*	*	*
EI	IFF←1	1	1	Im	*	*	*	*	*	*
IM0	Mettre Interruption Mode 0	2	2	Im	*	*	*	*	*	*
IM1	Interruption Mode 1	2	2	Im	*	*	*	*	*	*
IM2	Interruption Mode 2	2	2	Im	*	*	*	*	*	*

GROUPE ARITHMETIQUE 16 BITS

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
ADD HL,dd	HL←HL+dd	1	3	Im	?	*	*	*	0	-
ACD HL,dd	HL←HL+dd+CY	2	4	Im	?	?	V	?	0	-
SBC HL,dd	HL←HL-dd-CY	2	4	Im	?	?	V	?	1	-
ADD IX,pp	IX←IX+pp	2	4	Im	?	*	*	*	0	-
ADD IY,rr	IY←IY+rr	2	4	Im	?	*	*	*	0	-
INC dd	dd←dd+1	1	1	Im	*	*	*	*	*	*
INC IX	IX←IX+1	2	2	Im	*	*	*	*	*	*
INC IY	IY←IY+1	2	2	Im	*	*	*	*	*	*
DEC dd	dd←dd-1	1	1	Im	*	*	*	*	*	*
DEC IX	IX←IX-1	2	2	Im	*	*	*	*	*	*
DEC IY	IY←IY-1	2	2	Im	*	*	*	*	*	*

GROUPE ROTATION ET DECALAGE

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
RLCA		1	1	Im	?	*	*	*	0	0
RLA		1	1	Im	?	*	*	*	0	0
RRCA		1	1	Im	?	*	*	*	0	0
RRA		1	1	Im	?	*	*	*	0	0
RLCr		2	2	Im	?	?	P	?	0	0
RLC(HL)		2	4	Im	?	?	P	?	0	0
RLC(IX+d)		4	6	Im	?	?	P	?	0	0
RLC(IY+d)		4	6	Im	?	?	P	?	0	0
RL s		-	-	Im	?	?	P	?	0	0
RRC s		-	-	Im	?	?	P	?	0	0
RR s		-	-	Im	?	?	P	?	0	0
SLA s		-	-	Im	?	?	P	?	0	0
SRA s		-	-	Im	?	?	P	?	0	0
SRL s		-	-	Im	?	?	P	?	0	0
RLD		2	5	Im	*	?	P	?	0	0
RRD		2	5	Im	*	?	P	?	0	0

GROUPE MISE, ANNULATION ET TEST DE BIT

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
Bit 6,r	Z<--r	2	2	RR	*	?	-	-	0	1
Bit 6,(HL)	Z<--(HL)	2	3	In	*	?	-	-	0	1
Bit 6,(IY+d)	Z<--(IY+d)	4	5	IDX	*	?	-	-	0	1
Bit 6,(IY+d)	Z<--(IY+d)	4	5	IDX	*	?	-	-	0	1
Set 6,r	r<--1	2	2	RR	*	*	*	*	*	*
Set 6,(HL)	(HL)<--1	2	4	In	*	*	*	*	*	*
Set 6,(IX+d)	(IX+d)<--1	4	6	IDX	*	*	*	*	*	*
Set 6,(IY+d)	(IY+d)<--1	4	6	IDX	*	*	*	*	*	*
Set 6,s	s<--0	-	-	RR	*	*	*	*	*	*

GROUPE DE JUMP (SAUT)

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
JP nn	PX<--nn	3	3	Im	*	*	*	*	*	*
JP cc,nn	Si condition cc vraie PC<--nn sinon continue	3	3	Im	*	*	*	*	*	*
JR e	PC<--PC+e	2	3	Im	*	*	*	*	*	
JR C e	Si c=0 continue	2	2	Im	*	*	*	*	*	
JR NC e	Si c=1 continue	2	2	Im	*	*	*	*	*	

JR Z e	Si z=0 continue	2	2	Im	*	*	*	*	*	
JR NZ e	Si z=1 continue	2	2	Im	*	*	*	*	*	*
JP(HL)	PC<--HL	1	1	In	*	*	*	*	*	*
JP(IX)	PC<--IX	2	2	IDX	*	*	*	*	*	*
JP(IY)	PC<--IY	2	2	IDX	*	*	*	*	*	*
DJNZ e	B<--B-1 Si B=0 continue B=0 PC<--PC+e	2	3	Im	*	*	*	*	*	*

GROUPE CALL ET RETURN

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
CALL n	(SP-1)<--PC (SP-2)<--PC	3	5	Im	*	*	*	*	*	*
CALL cc,nn	Si condition cc fausse, continuer sinon comme CALL nn	3	3	Im	*	*	*	*	*	*
RET	PC<--(SP) PC<--(SP+1)	1	3	Im	*	*	*	*	*	*
RET cc	Si condition cc fausse, continuer sinon comme RET	1	1	Im	*	*	*	*	*	*
RETI	Retour d'interruption	2	4	Im	*	*	*	*	*	*
RETN	Retour d'interruption non masquable	2	4	Im	*	*	*	*	*	*
RET P	(SP-1)<--PC (SP-2)<--PC PC<--0 PC<--P	1	3	Im	*	*	*	*	*	*

GROUPE ENTREE ET SORTIE

Mnémonique	Opération Symbolique	Nbre d'octets	Nbre de cycles	Mode d' adressage	Flags affectés C	Z	P/V	S	N	H
IN A,(n)	A<--(n)	2	3	Im	*	*	*	*	*	*
IN r,(c)	r<--(c)	2	3	In	*	?	P	?	0	?
INI	(HL)<--(c) B<--B-1 HL<--HL+1	2	4	In	*	?	-	-	1	-
INIR	(HL)<--(c) B<--B-1 HL<--HL+1 JUSQUE B=0	2	5	In	*	1	-	-	1	-
IND	(HL)<--(c) B<--B-1 HL<--HL-1	2	4	In	*	?	-	-	1	-
INDR	(HL)<--(c) B<--B-1 HL<--HL-1 JUSQUE B=0	2	5	In	*	1	-	-	1	-
OUT (n),A	(n)<--A	2	3	Im	*	*	*	*	*	*
OUT (c),r	(c)<--r	2	3	In	*	*	*	*	*	*
OUTI	(c)<--(HL) B<--B-1 HL<--HL+1	2	4	In	*	?	-	-	1	-
OUTIR	(c)<--(HL) B<--B-1 HL<--HL+1 JUSQUE B=0	2	5	In	*	1	-	-	1	-
OUTD	(c)<--(HL) B<--B-1 HL<--HL-1	2	4	In	*	?	-	-	1	-
OTDR	(c)<--(HL) B<--B-1 HL<--HL-1 JUSQUE B=0	2	5	In	*	1	-	-	1	-

ANNEXE 2

EFFETS DES INSTRUCTIONS SUR LES FLAGS

Les Flags

Bit	Flag	
0	C	Carry
1	N	Addition/Soustraction
2	P/V	Parité/Dépassement
3	-	-
4	H	Demi-carry
5	-	-
6	Z	Zéro
7	S	Signe

Effets des Instructions Sur Le Flag Signe

Groupe	Instruction	Effets

Charge 8 bits	LD A,I	Mis si registre I est négatif, sinon annulé
	LD A,R	Mis si registre R est négatif, sinon annulé
Compare	CPI,CPIR CPD,CPDR	Mis si résultat est négatif, sinon annulé
arithmétique 8-bits	ADD A,S ADC A,S SUB S SBC A,S AND S OR S XOR S CP S INC S DEC S	Mis si résultat est négatif, sinon annulé
arithmétique générale	DAA	Mis si bit supérieur de A=1, sinon annulé
	NEG	Mis si résultat est négatif, sinon annulé
arithmétique 16-bits	ADC HL,SS SBC HL,SS	Mis si résultat est négatif, sinon annulé
Décalage et rotation	RLC S RL S RRC S RR S SLA S SRA S SRL S	Mis si résultat est négatif, sinon annulé
	RLD RRD	Mis si A est négatif acres décalage, sinon annulé

Bit	BIT B,S	Altéré
Entrées et Sorties	IN R, (C)	Mis si donnée entrée est négative, sinon annulé
	IN,INIR IND, INDR OUTI,OTIR OUTD,OTDR	Altéré

Effets Des Instructions Sur Le Flag Zéro

Charge 8-bits	LD A,I	Met Z si registre I=0, sinon annule Z
	LDA,R	Met Z si registre R=0, sinon annule Z
Compare	CPL,CPIR, CPD,CPDR	Met Z si A=(HL), sinon annule Z
Arithmétique 8-bits	ADD A,S ADC,A,S SUB S SBC A,S OR S XOR S CP S INC S DEC S	Met Z si résultat=0, sinon annule Z
arithmétique générale	DAA NEG	Met Z si résultat=0, sinon annule Z
arithmétique 16-bits	ADC HL,SS SBC HL,SS	Met Z si résultat=0, sinon annule Z
Décalage et rotation	RLC S RL RRC S RR S SLA S SRA S SRL S RLD RRD	Met Z si résultat=0, sinon annule Z
Bit	BIT B,S	Met Z si bit spécifié=0, sinon annule Z

Entrées et Sorties	INR,(C)	Met	Z si donnée entrée=0 sinon annule Z
	INLIND,	Met	Z si B-1=0, sinon annule Z
	INIR,INDR	Mis	
	OUTLOUTD	Met	Z si B-1=0, sinon annule Z
	OTIR,OTDR	Met	Z

Effets Des Instructions Sur Les Flags Demi-Carry et Addition-Soustraction

Groupe	Instruction	Effets Sur Demi-Carry	Addition / Soustraction
Charge 8-bits	LD A,I LD A,R	Annulé	Annulé
	LDI,LDIR, LDD,LDDR	Annulé	Annulé
Compare	CPI,CPIR,CPD CPDR	Mis si pas de retenue de bit 4 sinon annulé	Mis
Arithmétique 8-bits	ADD A,S ADC A,S	Mis si pas de carry de bit 3, sinon annulé	Annulé
	SUB S SBC A,S	Mis si pas de retenue de bit 4, sinon annulé	Mis
	AND S OR S XOR S	Mis	Annulé
	CP S	Mis si pas de retenue De bit 4, sinon annulé	Mis
	INC S	Mis si carry de bit 3 sinon annulé	Annulé
	DEC S	Mis si pas de retenue de bit 4, sinon annulé	Mis
Arithmétique générale	DAA	Altéré	Pas d'effet
	CPL	Mis	Mis
	NEG	Mis si pas de retenue de bit 4, sinon annulé	Mis
	CCF	Pas d’effet	Annulé
	SCF	Annulé	Mis
Arithmétique 16-bits	ADD HL,SS ADD HL,SS	Mis si carry venant de bit 11, sinon annulé	Annulé
	SBC HL,SS	Mis si pas de retenue de bit 12, sinon annulé	Mis
	ADD IX,PP ADD IY,RR	Mis si carry venant de bit 11, sinon annulé	Annulé
Décalage et rotation	RLCA RLA RRCA RRA RLC S RLS RRCS RR S SLA S SRA S SRI S RLD RRD	Annulé	Annulé
Bit	BIT B,S	Mis	Annulé
Entrées Sorties	INR,(C)	Annulé	Annulé
	INI,INIR, IND,INDR, OUTI,OTIR, OUTD,OTDR	Altéré	Mis

Effets de l'instruction sur le Flag Parité/Dépassement

Groupe	Instruction	Effets
Charge 8-bits	LD A,I LD A,R	Copie de la bascule d'interruption 2
Instructions de blocs	LDI,LDD CPI,CPIR CPD,CPDR	Mis si BC<>1, sinon annulé
	LDIR,LDDR	Annulé
	ADD A,S ADC A,S SUB S SBC A,S	Mis si fin de page, sinon annulé
	AND S	

arithmé-tique 8-bits	OR S XOR S	Mis si parité paire, sinon annulé
	CPS	Mis si fin de page, sinon annulé
	INC S	Mis si opérande était égale à &7F avant incrémentage, sinon annulé
	DEC S	Mis si opérande était égale à 80H avant incrémentage, sinon annulé
arithmé-tique générale	DAA	Mis si parité de (A) paire, sinon annulé
	NEG	Mis si(A) était égal à &80 avant inversion, sinon annulé
arithmé-tique 16-bits	ADC HL,SS SBC HL,SS	Mis si fin de page, sinon annulé
Décalage et rotation	RLCS RL S RRC S RR S SLA S SRA S SRL S RLD S RRD S	Mis si parité paire, sinon annulé
Bit	BIT B,S	Altéré
Entrées et Sorties	IN R,(C)	Mis si parité paire, sinon annulé
	INI,INIR, IND, INDR, OUTI,OTIR, OUTD,OTDR	Altéré

Effets Des Instructions Sur Le Flag Carry

Groupe	Instructions	Effets sur le Carry
Charge 8-bits	LD A,I LD A,R	Pas d'effet
Instruc-tions de bloc	CPI,CPIR, CPD,CPDR, LDI,LDIR LDD,LDDR	Pas d'effet
arithmé-tique 8-bits	ADD A,S ADC A,S	Mis si retenue du bit 7, sinon annulé
	SUB S SBC S	Mis si pas de retenue, sinon annulé
	AND S OR S XOR S	Annulé
	CP S	Mis si pas de retenue, sinon annulé
Arithmé-tique générale	DAA	Mis si retenue bcd, sinon annulé
	NEG	Mis si A différent de 0 avant inversion, sinon annulé
	CCF	Mis si CCF égal à 0 avant CCF, sinon annulé
	SCF	Mis
Arithmé-tique 16-bits	ADD HL,SS ADD LH,SS	Mis si retenue du bit 15, sinon annulé
	SBC HL,SS	Mis si pas de retenue, sinon annulé
	ADD IX,PP ADD IY,RR	Mis si retenue du bit 15, sinon annulé
Décalage et rotation	RLCA RLA	Copie A bit 7
	RRCA RRA	Copie A bit 0
	RLC S RLS	Copie bit 7 de l'opérande
	RRC S RR S	Copie bit 0 de l'opérande
	SLA S	Copie bit 7 de l'opérande

SRA S Copie bit 0 de l'opérande

SRL S Copie bit 0 de l'opérande

ANNEXE 3

Les Effets Des Comparaisons Sur Les Flags Dépassement, Signe Et Carry

		Interprétation				
A	S	8-bits	2's comp	V	S	C
30	20	A>s	A>s	0	0	0
20	C0	A<s	A>s	0	0	1
70	C0	A<s	A>s	1	1	1
C0	70	A>s	A<s	1	0	0
C0	20	A>s	A<s	0	1	0
20	30	A<s	A<s	0	1	1

s = A,B,C,D,E,H,L,(HL),(IX+d),OU (IY+d)

1 = flag mis 0 = flag annulé 2's comp=complément à 2

Conditions:

Flag	Mis	Annulé
V	PO (Parité Impaire)	PE (Parité Paire)
S	P (Positif)	M (Négatif)
C	C (Retenue)	NC (Pas de retenue)

Dans le premier tableau, si les flags signes et dépassement sont identiques, alors A>s en complément à deux. S'ils sont différents, alors A<s, si bien sûr le flag zéro n'est pas mis.

ANNEXE 4

QUELQUES SOUS-PROGRAMMES INTEGRES

&BB00 Initialisation gestion clavier

Entrée | Sortie

Rien | A,B,C,D,E,H,L et Flags altérés.

Efface buffer du clavier. Met répétition des touches sur valeurs par défaut. Shift et caos locks sur off. Interdit les événements d'interruption

---

&BB03 Restaurer la gestion clavier

Entrée | Sortie

Rien | A,B,C,D,E,H,L et Flags altérés.

Efface buffer de clavier. Met répétition des touches sur valeurs défaut. Autorise les interruptions. Interdit les événements d'interruptions.

&BB06 Attend caractère

Entrée | Sortie

Rien | A: Le caractère (ASCII)
Carry:Vrai Autres Flags:Altérés

Etend les tokens simples d'extension.

&BB09 Extraction caractère

Obtient caractère du buffer de clavier. Si vide, ne pas attendre

---

Entrée | Sortie

Rien | A: Le caractère, ou altération
Carry; Vrai si caractère,
sinon faux Autres Flags:Altérés

Etend les tokens simples d'extension.

&BB18 Attente caractère (non-étendu)

Obtient pressage touche du buffer de clavier. Attendre si absent.

Entrée | Sortie

Rien | A: Le caractère/token (ASCII)
Carry: Vrai
Autres Flags altérés

N'étend pas les tokens d'extension.

&BB1B Extraction caractère (non-étendu)

Obtient pressage touche du buffer de clavier. Sinon, ne pas attendre
Entrée

Entrée | Sortie

Rien	A: Le caractère ou altération Carry: Vrai si caractère disponible Autres Flags:Altérés

N'étend pas les tokens d'extension.

&BB1E Test touche

Regarder si une touche spécifique ou le bouton joystick sont appuyés.

Entrée	Sortie
A: numéro de touche (ASCII)	A,HL: Altérés C: 128 = ctrl appuyé 32 = shift appuyé 160 = shift et ctrl appuyés Carry: Faux Zéro:Faux si appuyé, sinon vrai Autres Flags: Altérés

&BB21 Etat clavier

Regarder si shift lock et caps locks sont appuyés

Entrée	Sortie
Rien	A: Altéré L: 0 = shift lock enlevé 255 = lock mis H; H : 0 = caps lock enlevé 255 = caps lock mis Flags: Altérés

&BB24 Etat Joystick

Entrée	Sortie
Rien	A: Etat de joystick H: Etat de joystick 0 L: Etat de joystick 1

Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Réservé	Fire 1	Fire 2	Droite	Gauche	Bas	Bouton Haut

Bit 7 = 0; Bits ci-dessus mis = état correspondant vrai

&BB39 Mise Touche Répétition

Autorise/interdit auto-répétition sur touche spécifique

Entrée	Sortie
A: numéro de touche (ASCII) B: 0 = Pas de répétition 255 = Répétition possible	A,B,C,H,L: Altérés Flags: Altérés

&BB3C Test Répétition

Trouver état répétition sur touche spécifique

Entrée	Sortie
A : Numéro de touche	A,HL: Altérés Zéro: Faux si répétition, sinon vrai Carry: Faux

&BB3F Mise En Place De Répétition

Fixer retard de démarrage et vitesse de répétition

Entrée	Sortie
H: Démarrage retard L: Vitesse de répétition	A: Altéré Flags: Altérés

Les retards et les répétitions sont attendus en 50 èmes de seconde,
50 = 1 seconde

&BB4E Initialisation Du Texte VDU

Entrée	Sortie
Rien	A,B,C,D,E,H,L et Flags altérés.

Tous les canaux de texte sont fixés sur valeurs défaut (ink, paper, window, etc). Caractères définis par l'utilisateur perdus. Codes contrôles et indirections de texte mis sur valeurs défaut.

&BB51 Restaure Texte VDU

Entrée | Sortie

Rien | A,B,C,D,E,H,L et Flags altérés.

Met codes contrôle et indirections de texte sur valeurs défaut

---

&BB54 Autorisation De Texte VDU

Permet de placer des caractères sur l'écran.

Entrée | Sortie

Rien | A:Altéré
Flags: Altérés

Efface buffer de code contrôle

&BB57 Interdiction De Texte VDU

Entrée | Sortie

Rien | A: Altéré
Flags: Altérés

Vide buffer de code contrôle

&BB5A Sortie De Texte

Sort caractère sur canal actuel

Entrée | Sortie

A: caractère à envoyer(ASCII)

&BB5D Ecriture De Caractère De Texte Sur Ecran

Entrée | Sortie

A: Caractère à imprimer | A,B,C,D,E,HL et Flags altérés.

Met à jour position du curseur

&BB60 Lecture de caractère de texte à partir de l'écran

Entrée | Sortie

Rien | A: Le Caractère, ou zéro
Carry: Vrai si caractère,
sinon faux
Autres flags : Altérés

Position de curseur doit être fixée d'abord.

&BB63 Mise En Place Texte Graphique

Entrée | Sortie

A: <> 0 autorise
= 0 interdit | A: Altéré
Flags: Altérés

Quand ils sont autorisés, les codes contrôles sont imprimés, pas exécutés

---

&BB66 Définition d'une fenêtre

Fixe les dimensions d'une fenêtre de texte

Entrée | Sortie

H: Valeur colonne pour un côté
D: Valeur colonne pour autre côté
L: Valeur ligne pour un côté
E: Valeur ligne pour autre côté | A,B,C,D,E,H,L et Flags altérés.

Le plus petit de H et D est la colonne de gauche. Le plus petit de L et E est la première ligne. Coin supérieur gauche de l'écran = 1,1 Met curseur sur coin gauche de la fenêtre

---

&BB6C Efface Fenêtre De Texte

Entrée | Sortie

Rien | A,B,C,D,E,H,L et Flags altérés.

Met curseur sur coin supérieur gauche de la fenêtre

&BB6F Fixe Colonne Texte

Entrée | Sortie

A: Nouvelle colonne | A,H,L: Altérés
Flags: Altérés.

---

1 = Colonne la Plus a gauche de la fenêtre

&BB72 Fixe Ligne Texte

Entrée	Sortie
A: Nouvelle ligne	A,H,L et Flags altérés.

1 = Première ligne de la fenêtre

&BB78 Trouver Curseur Texte

Entrée	Sortie
Rien	H: Colonne curseur L: Ligne curseur A: Compteur de scrolling Flags: Altérés

&BB87 Teste Position Curseur

Trouve où imprimer le prochaine caractère

Entrée	Sortie
H: Colonne à contrôler L: Rangée à contrôler	A: Altéré B: = 0 si fenêtre se déplace vers le bas = 255 si fenêtre se déplace vers le haut sinon altéré H: = colonne où aura lieu l'impression L: = Rangée où aura lieu l'impression Carry = Faux si fenêtre se déplace, sinon vrai Autres flags: Altérés

1,1 = coin supérieur gauche de la fenêtre.

Fixe Couleur Pen Fixe couleur texte premier-plan

Entrée	Sortie
A: numéro couleur	A,H,L et Flags altérés.

&BB96 Fixe Couleur Racler

Fixe couleur fond texte

Entrée	Sortie
A: Numéro couleur papier	A,H,L et Flags altérés.

&BB9C Echange Couleurs Texte

Echange premier-Plan et fond de texte

Entrée	Sortie
Rien	A,H,L et Flags altérés.

&BBBA Initialisation de VDU graphique

Entrée	Sortie
Rien	A,B,C,D,E,H,L et Flags altérés.

Met indirections VDU graphique sur défaut. Annule fenêtre graphique sur tout l'écran. N'efface pas la fenêtre.

&BBBD Annulation VDU graphique

Entrée	Sortie
Rien	A,B,C,D,E,H,L et Flags altérés.

Fixe VDU graphique sur valeurs défaut.

&BBCO Déplacement graphique absolu

Entrée	Sortie
DE: Coordonnée absolue de X HL: Coordonnée absolue de Y	A,B,C,D,E,H,L et Flags altérés.

&BBC3 Déplacement graphique relatif

Entrée	Sortie
DE: Coordonnée relative de X HL: Coordonnée relative de Y	A,B,C,D,E,H,L et Flags altérés

&BBC6 Trouver position graphique

Entrée	Sortie
Rien	DE:Coordonnée de X HL:Coordonnée de Y A: Altéré Flags: Altérés

&BBC9 Fixe position graphique

Entrée	Sortie
DE: Coordonnée X HL: Coordonnée Y	A,B,C,D,E,H,L et Flags altérés.

(0,0) = coin bas gauche de l'écran= origine graphique défaut

&BBCC Trouver origine graphique

Entrée	Sortie
Rien	DE: Coordonnée X HL: Coordonnée Y

(0<0) = coin bas gauche de l'écran = origine graphique défaut

&BBCF Fixe largeur de fenêtre graphique

Entrée	Sortie
DE: Coordonnée X d'un cOté HL: Coordonnée X autre coté	A,B,C,D,E,H,L et Flags altérés.

Coté gauche = le plus petit de DE et HL (0,0) = coin bas gauche de l'écran

&BBD2 Fixe hauteur de fenêtre graphique

Entrée	Sortie
DE: Coordonnée Y d'un côté HL: Coordonnée Y autre côté	A,B,C,D,E,H,L et Flags altérés.

(0,0) = bas gauche
Coté en bas = Plus bas de DE et HL

&BBDB Effacage fenêtre graphique

Fixe fenêtre sur couleur de fond

Entrée	Sortie
Rien	A,B,C,D,E,H,L et Flags altérés.

&BBDE Fixe couleur crémier Plan graphique

Entrée	Sortie
A: Numéro couleur	A et Flags altérés

&BBE4 Fixe couleur fond graphique

Entrée	Sortie
A: Numéro couleur	A et Flags altérés

&BBE7 Trouver couleur fond graphique

Entrée	Sortie
Rien	A:numéro couleur fond Flags: altérés

&BBEA Plot graphique absolu

Entrée	Sortie

DE: Coordonnée X
HL:Coordonnée Y

A,B,C,D,E,H,L et
Flags altérés.

&BBED Plot graphique relatif

Entrée

Sortie

DE: Coordonnée relative X
HL: Coordonnée relative Y

A,B,C,D,E,H,L et Flags altérés.

&BBFO Tester point graphique absolu

Entrée

Sortie

DE: Coordonnée X
HL:Coordonnée Y

A: couleur de fond du point
B,C,D,E,H,L et Flags altérés.

&BBF3 Tester point graphique relatif

Entrée

Sortie

DE: Coordonnée relative X
HL: Coordonnée relative Y

A: couleur de fond du point
B,C,D,H,L, et Flags altérés.

&BBF6 Ligne graphique absolu

Entrée

Sortie

DE: Coordonnée X point final
HL: Coordonnée Y point final

A,B,C,D,E,H,L et Flags altérés.

&BBF9 Ligne graphique relative

Entrée

Sortie

DE: Coordonnée relative X
HL: Coordonnée relative Y

A,B,C,D,E,H,L et Flags altérés.

&BBFC Texte graphique

Ecrit texte dans position graphique actuelle

Entrée

Sortie

A:Caractère

A,B,C,D,E,H,L et Flags altérés.

Coin supérieur gauche du caractère écrit en position graphique actuelle. La position graphique actuelle est mise à jour.

&BCOE Met mode écran

Fixe mode écran, et VDU texte et graphique

Entrée

Sortie

A: Mode requis

A,B,C,D,E,H,L et Flags altérés.

&BC14 Efface écran

Efface mémoire écran

Entrée

Sortie

Rien

A,B,C,D,E,H,L et Flags altérés.

&BC32 Fixe couleur INK

Entrée

Sortie

A : Numéro INK
B : Première couleur
C: Deuxième couleur

A,B,C,D,E,H,L et Flags altérés.

&BC4D Scrolling ligne

Déplace l'écran d'une ligne vers le haut ou vers le bas

Entrée

Sortie

Vers le bas
B = 0
Vers le haut
B <> 0
A: couleur PAPER de nouvelle ligne

A,B,C,D,E,H,L et Flags altérés.

&BC59 Fixe mode plot graphique

Entrée	Sortie
A: Mode écriture	A,B,C,D,E,H,L et Flags altérés.

A = 0: mode FORCE NEW = INK
A = 1: mode XOR NEW = INK XOR OLD
A = 2: mode AND NEW = INK AND OLD
A = 3: mode OR NEW = INK OR OLD
NEW = METTRE PIXEL FINAL; OLD = METTRE PIXEL ACTUEL
INK = INK PLOTTé

&BCA7 Réinitialisation son

Entrée	Sortie
Rien	A,B,C,D,E,H,L et Flags altérés.

&BCAA Canal son

Ajouter un son au canal

Entrée	Sortie
HL: Adresse du programme sonore	HL: Altéré si son ajouté : Préservé si canal plein Carry: Vrai si son ajouté Faux si canal Plein A,B,C,D,E,IX: Altérés Autres flags: Altérés

---

OCTET 0: Canaux et synchronisation.
OCTET 1: Enveloppe d'amplitude à utiliser
OCTET 2: Enveloppe de ton à utiliser.
OCTETS 3,4: Période de ton.
OCTET 5: Période de son.
OCTET 6: Amplitude de départ.
OCTET 7,8: Durée, ou nombre de répétitions.

Configuration de
l'octet 0:

BIT 0: Canal A
BIT 1: Canal B
BIT 2: Canal C
BIT 3: Synchronisé avec Canal A
BIT 4: Synchronisé avec Canal B
BIT 5: Synchronisé avec Canal C
BIT 6: Son tenu
BIT 7: Queue vide

&BCAD Test son

Etat d'une queue son

Entrée	Sortie
A: Bit de test	B,C,D,E,H,L et Flags altérés.

&BCBO Evénement Queue Vide

Prépare le son pour qu'il se produise quand une queue se vide

---

Entrée	Sortie
A : Bit de liaison (Evénement-Canal) HL: Adresse d'événement	A,B,C,D,E,H,L et Flags altérés.

Bit 0: Canal de lien A
Bit 1: Canal de lien B
Bit 2: Canal de lien C

&BCB3 Relâchement du son

Relâche des sons particuliers tenus

Entrée	Sortie
A: Canaux à relâcher	A,B,C,D,E,H,L et Flags altérés.

Bit 0: Relâche canal A
Bit 1: Relâche canal B
Bit 2: Relâche canal C

&BCB6 Son tenu

Tous les sons sont arrêtés immédiatement

Entrée	Sortie
Rien	Carry: Vrai si son actif, sinon faux A,B,C,D,E,H,L et Flags altérés.

Les sons sont automatiquement relancés par CANAL SON, RELACHE SON et CONTINUE SON.

&BCB9 Continue son

Relâche tous les sons tenus

Entrée	Sortie
Rien	A,B,C,D,E,H,L et Flags altérés.

&BCBC Enveloppe d'amplitude

Etablit une des 15 enveloppes d'amplitude

Entrée	Sortie
A: Numéro d'enveloppe HL: Adresse d'amplitude	HL: Contient adresse du bloc de données plus 16 si enveloppe correcte, sinon préservé A,B,C: Préservés si Enveloppe, incorrecte, sinon altérés D,E: Altérés Carry:Vrai si enveloppe correcte, sinon faux Autres flags: Altérés

---

OCTET 0 : Nombre de parties de l'enveloppe
OCTETS 1,2,3 : Première partie de l'enveloppe
OCTETS 4,5,6 : Deuxième partie de l'enveloppe
OCTETS 7,8,9 : Troisième partie de l'enveloppe
OCTETS 10,11,12: Quatrième partie de l'enveloppe
OCTETS 13,14,15: Cinquième partie de l'enveloppe

&BCBF Enveloppe de ton

Etablit une des 15 enveloppes de ton

Entrée	Sortie
A: Numéro d'enveloppe HL:Adresse du bloc de données de ton	HL: Contient l'adresse du bloc de données plus 16 si enveloppe correcte, sinon préservé A,B,C: Préservé: enveloppe incorrecte Carry: Vrai si enveloppe correcte sinon faux Autres flags: Altérés

OCTET 0: Nombre de parties de l'enveloppe
OCTETS 1,2,3: Première partie de l'enveloppe
OCTETS 4,5,6: Deuxième partie de l'enveloppe
OCTETS 7,8,9: Troisième partie de l'enveloppe
OCTETS 10,11,12: Quatrième partie de l'enveloppe
OCTETS 13,14,15: Cinquième partie de l'enveloppe

&BCC2 Adresse d'amplitude

Trouve l'adresse d'une enveloppe d'amplitude

Entrée	Sortie
A: Numéro d'enveloppe	A: Altéré BC: Longueur de l'enveloppe en octets si enveloppe correcte, sinon préservé HL: Adresse de l'enveloppe si enveloppe correcte, sinon altéré Carry; Vrai si enveloppe correcte sinon faux Autres flags: Altérés

---

A doit être entre 1 et 15

&BCC5 Adresse ton

Adresse d'enveloppe de ton

Entrée	Sortie
A: Numéro d'enveloppe	A: Altéré BC: Longueur de l'enveloppe si enveloppe correcte, sinon préservé HL: Adresse de l'enveloppe si enveloppe correcte, sinon altéré

Carry: Vrai si enveloppe
correcte, sinon faux
Autres flags: Altérés

A doit être entre 1 et 15

ANNEXE 5

Notations Binaire, en Code Binaire et en Hexadécimal

Les systèmes numériques modernes, en usage dans le monde entier, utilisent le système décimal. Il a été développé pour Pouvoir compter au delà de 10 et en deçà de 1. Dans ce système, les chiffres de gauche dans un nombre ont une valeur plus grande que ceux de droite; par exemple dans le nombre 66 le premier 6 a une valeur 10 fois supérieure au second, c'est-à-dire

```
66 --> 60
   --> 6
```

Ceci est étendu dans des nombres plus grands dont les chiffres vers la gauche représentent des multiples de puissances de dix croissantes

```
6666 -->   6   x   1000
     -->   6   x   100
     -->   6   x   10
     -->   6   X   1
```

Un système où la position ou emplacement d'un chiffre dans un nombre modifie sa valeur est appelé système numérique à Valeur d'Emplacement. En système décimal, la valeur du même chiffre placé tout de suite à gauche est dix fois Plus grande et c'est ce qu'on appelle la BASE de ce système. D'autres systèmes utilisent des bases différentes mais suivent le même modèle que le système décimal, c'est-à-dire que l'emplacement de gauche est plus grand, étant multiplié par la base.

L'ordinateur, fonctionnant avec le courant électrique, reconnaît seulement deux états, allumé ou éteint, souvent représentés Par "1" et "0". Il utilise donc le système binaire - c'est-à-dire la base 2. N'importe quel nombre en binaire est uniquement constitué de 0 et de 1, ou en électricité, éteint et allumé (ou en électronique, zéro volt et quelques volts). Pour compter après un, le système binaire doit employer la notation avec valeur d'emplacement et, comme dans les autres systèmes, le facteur de la multiplication est la base, c'est-à-dire 2. Dès lors, le nombre 101 en base 2 ou binaire représente:

```
 1x4  0x2  1x1
       ^
---- 1 0 1 -----
```

c'est-à-dire 4+0+1=5. En fait, le grand nombre de bases présente un problème pour représenter les nombres. En système décimal (base 10), "101" représente cent un alors qu'en système binaire (base 2) "101" représente 5, Pour lever cette ambiguïté, il existe une convention pour représenter des nombres, la base est indiquée a la droite du nombre juste en dessous de la ligne. Dès lors nos deux nombres deviennent:

```
101  = Cent un en base dix
  10
101  = Cinq en base deux
   2
```

La génération actuelle d'ordinateurs domestiques utilise des registres ou mémoires de huit bits et peuvent donc représenter des nombres Jusqu'à 11111111, c'est-à-dire 255 en base 10:

```
 128 + 64 + 32 + 16 + 8 + 4 + 1 = 255
                                   10
  1    1    1    1    1   1   1 = Chiffre

                                Equivalent en
 128   64   32   16   8   4   1   base 10
```

FIGURE A.1

Exécutons un autre exemple de conversion - prenons 10100111 (base 2)

```
1 x 128      0 x 64      1 x 4      1 x 2      1 x 1
   ^           ^          ^  ----------^          ^
------------1 0 1 0 0 1 1 1---------------------
          ------^ ^ ^-------
         1 X 32  0 X 16   0x8
```

```
Donc 10100111  = 1x128 +0x64 +1x32 +0x16 +0x8 +1x4 +1x2 + 1x1
               = 128+32+4+2+1
               = 167
                   10
```

Pour être sûr que vous avez bien compris, essayez l'exercice suivant:

EXERCICE A5.1

Calculer la valeur des nombres binaires suivants en base 10:

```
I) 00000011
               2
II) 00000100
               2
III) 10000000
               2
IV) 10000011
               2
V) 10110111
              2
VI) 01110011
              2
```

Les réponses se trouvent dans le chapitre solutions.

Pour rendre plus claire l'idée de conversion de nombres de décimal en binaire essayez le programme "BIN/HEX" sur la cassette. Tapez ce qui suit:

```
RUN "BIN/HEX"
```

L'ordinateur va charger et exécuter automatiquement le programme et faire apparaître l'affichage initial du programme. Examinez-le bien.

Ne vous souciez pas des cases "HEX" et "BCD", nous y viendrons par la suite. Ce qui nous intéresse pour l'instant ce sont les cases "Décimal" et "binaire".

Comme l'affichage l'indique, le programme attend que vous entriez un nombre entre 0 et 255, Tapez "1" et appuyez sur "ENTER". L'écran change:

Toutes les cases ont changé pour afficher la valeur de 1 dans les différentes notations. Les cases d'instruction vous indiquent que la valeur actuelle peut être incrémentée ou décrémentée en utilisant respectivement les touches curseur flèche haut et flèche bas.

Appuyez sur la touche flèche haut et à mesure que les valeurs augmenteront vous pourrez voir comment on compte en binaire. Si maintenant vous appuyez sur "E" vous pourrez entrer un nouveau nombre de départ. Entrez "15" et la case binaire devrait contenir "00001111". Maintenant incrémentez ce nombre de un. Les quatre octets de gauche (les octets les moins significatifs) sont tous passé à zéro et le cinquième octet est devenu un.

Pour comprendre ceci posons l'addition;

```
   1111  A
   + 1   B
```

En ajoutant le 1 a 1 on obtient "2" c'est-à-dire 0, retenue 1. Cette retenue produit un autre "0" + une retenue, et ainsi de suite.

Quand le registre est plein, c'est-à-dire quand on a 8x11111111, l'addition d'un 1 supplémentaire remettra le registre à zéro et 256 sera perdu. Cependant tout n'est pas perdu avec le microprocesseur Z80 car il a un flag carry qui enregistre le fait qu'une retenue s'est produite, (le flag carry est indiqué par "CF" sur l'affichage.)

Pour voir le flag carry fonctionner appuyez sur "E" et ensuite commencez le comptage à 250, c'est-à-dire entrez 250 <RETURN>. Maintenant il est facile d'incrémenter jusqu'à 255 ou &x11111111.

Maintenant surveillez la case binaire lorsque vous passez de 255 à 256. Le système binaire revient à zéro + le bit de retenue. C'est une propriété très pratique du Z80 mais sur laquelle on ne doit compter u pour stocker temporairement la retenue. Il est tout aussi facile de remettre à zéro ce bit que de le mettre sur un.

Le programme acceptera aussi des nombres entrés sous la forme binaire. Cependant, pour faire cela le nombre DOIT être précédé du préfixe "&x". Dès lors pour entrer le nombre binaire 101010 (42) tapez:

&X101010

et ensuite tapez ENTER. Toutes les cases changent pour afficher 42 dans les différentes notations.

Pour voir si vous comprenez bien la notation binaire essayez de convertir les nombres binaires suivants en décimal sur le papier et utilisez ensuite le programme pour contrôler vos réponses.

EXERCICE A5.2

i)   11111
ii)   101001
iii) 1011101
iv)   10001000

Les réponses sont dans le chapitre des solutions.
Gardez le programme BIN/HEX dans l'ordinateur; vous en aurez besoin bientôt!

Alors que les 0 et les 1 sont pratiques pour l'ordinateur, ils le sont beaucoup moins pour l'homme moyen il faut donc trouver un compromis. La notation décimale est peu utilisée car à part 1 en base 10 et 1 en base 2, il n'y a pas d'autre correspondance. Une autre idée serait de prendre les huit bits binaires pour un chiffre (c'est-à-dire jusqu'à 255) et d'utiliser une base de 256! Quelle objection verriez vous à cela, en dehors du fait que l'idée elle-même est un peu tirée par les cheveux?

C'est le moment de réfléchir... La réponse vient de l'examen du cas de la base 10 dans laquelle dix chiffres (0 à 9) sont nécessaires pour représenter les dix étapes jusqu'à 10. En base 2, on a besoin de deux chiffres, il faudrait donc 256 chiffres en base 256!

HEXADECIMAL

Un système de compromis adopté sépare les huit bits en deux parties et représente ceux-ci séparément. Dès lors le plus grand nombre à être représenté est 8x1111 ou 15 et ceci nécessite, en comptant zéro, seize symboles différents. Ceux adoptés dans ce but sont:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 Nombre décimal

0 1 2 3 4 5 6 7 8 9 A B C D E F Symbole (Nombre hexadécimal)

FIGURE A5.2

En utilisant cette notation, n'importe quel nombre de huit bits peut être représenté avec deux symboles, un pour les quatre bits les plus significatifs et un pour les quatre bits les moins significatifs. Pour éviter la description plutôt longue de ces deux moitiés d'octet, on leur donne le nom de quartet. Dès lors un octet consiste en deux quartets, un quartet plus significatif ou quartet fort et un quartet moins significatif ou quartet faible - voir Figure A5.3.

```
     1 1 1 1 ! 1 1 1 1 = 255

quartet fort  quartet faible

15 1 1 1 1     1 1 1 1 = 15

     F             F  =
       16              16
            FF
               16
```

FIGURE A5.3

On donne au système qu'on a décrit le nom d'HEXADECIMAL - le plus souvent appelé HEXA en abrégé. Son avantage principal, pour ce qui concerne les ordinateurs, est qu'il est compatible avec le système binaire. N'importe quel nombre de huit bits peut être représenté par deux caractères hexadécimaux.

Si vous regardez maintenant la case en haut à droite sur l'écran à laquelle on ne voulait pas s'intéresser auparavant, vous pouvez voir qu'elle compte en HEXA. La compatibilité entre binaire et HEXA apparaît même pour une retenue importante - Prenez par exemple &x1111, 15 ou &F : après un incrément cela change les 1 binaires en zéro et ajoute un 1 à gauche, c'est-à-dire; &x10000 ou &10. Ces points primordiaux de correspondance se produisent ainsi:

```
           PHASE 1
                  &x1 =   &1 =  1

          &x0001 0000 =  &10 =  16

&x0000 0001 0000 0000 =  &100 = 256
&X0001 0000 0000 0000 = &1000 = 4096
```

Comme vous avez déjà pu le remarquer, le comptage décimal est présent sur l'affichage DEC/BIN/HEX et on peut y suivre la progression du comptage.

Appuyez sur "E" et entrez la valeur de départ "1" et une nouvelle fois commencez d'incrémenter le comptage. Jusqu'à 9, les caractères hexa correspondent aux caractères décimaux et entre 10 et 15 chaque lettre correspond à un nombre décimal. Après 15, la conversion d'hexadécimal en décimal devient un petit peu plus compliquée, comme l'utilisation de deux nombres ensemble, par exemple &FF = 255, appelle une nouvelle fois une notation à valeur d'emplacement. Cette fois comme la base est 16 la proportion entre chaque emplacement et son voisin est 16.

Les valeurs en base 10 des emplacements en hexadécimal sont:

```
      x 4096  x 256   x 16    x 1

         1      2       3       4   Numéro d'emplacement
```

FIGURE A5.4

L'utilisation des Figures A5.4 et A5.5 illustre la décomposition de E92F en base 16 dans 59695 en base 10.

```
           E   9   2   F

E(14)x4096 + 9x256 + 2x16 F(15)x1 = 59695
```

FIGURE A5.5

Maintenant que vous maîtrisez complètement la notation hexa, essayez ce qui suit les deux premiers sont expliqués en détail dans le chapitre des solutions.

EXERCICE A5.3

Calculez la valeur décimale des nombres suivants:-

```
i)  &09      v) &OA
ii) &13     vi) &1A
iii)&A5    vii) &EA
iv) &AE
```

Faites d'abord l'exercice ci-dessus sur le papier. Le programme accepte aussi l'entrée de nombres en hexadécimal mais ils doivent être précédés du préfixe "&", Dès lors si vous entriez la valeur "&FF" l'équivalent décimal, 255, serait affiché. Vous pouvez contrôler vos réponses à l'exercice A5.3 en utilisant le programme ou en regardant dans le chapitre des solutions. Gardez le programme, vous en aurez besoin dans un moment.

DECIMAL CODE EN BINAIRE

En plus des notations décimale, binaire et hexadécimale on utilise un autre système en informatique, le système décimal codé en binaire (BCD). Comme son nom l'indique c'est une forme hybride avec a la fois des éléments du système binaire et du système décimal. Il est communément utilisé là où on a besoin d'une sortie en format digital, par exemple pour une montre digitale, ou quand on a besoin d'une grande précision et qu'aucun bit ne peut être gaspillé.

En BCD la base décimale normale est retenue, c'est-à-dire qu'un emplacement est un facteur de 10 fois son voisin mais tous les différents chiffres sont représentés en binaire. Le nombre 87 sera donc représenté par:

```
                  8 7   base 10

                1000     0111
```

c'est-à-dire BCD = 1000 0111 (ou en huit bits 10000111)

FIGURE A5.6

Comme le plus grand chiffre utilisé en notation décimale est 9, on a seulement besoin de quatre bits en binaire pour le représenter, c'est-à-dire 9 = &x1001, un chiffre en BCD peut donc être représenté par un quartet et deux chiffres par un octet. La Figure A5.6 nous le montre, où 87 est représenté en BCD par &X10000111. Cela peut créer une ambiguïté dans le fait que &X10000111 en binaire égale 135. Pour la lever , les représentations en BCD prendront la notation suivante 10000111 (BCD).

En utilisant quatre bits du binaire, il est possible de compter jusqu'à 15 (8X1111 = 15) mais en BCD le plus grand chiffre est 9, inévitablement BCD est moins économique dans son utilisation de la place mémoire. Son plus grand chiffre, 9, égale &X1001 et si on lui ajoute un on fait tourner jusqu'à &XOOOO et on retient 1 pour le quartet suivant, c'est-à-dire

```
 8 = 0000 1000  (BCD base 2)
 9 = 0000 1001   "  "  "
10 = 0001 0000   "  "  "
11 = 0001 0001   "  "  "
```

FIGURE A5.7

L'affichage qui nous concerne est celui marqué BCD. Appuyez sur "E" et entrez la valeur de départ de 1. Si vous incrémentez les 9 premiers nombres vous verrez que Jusqu'à 9, binaire et BCD sont identiques. Cependant, quand vous incrémentez surveillez la case BCD et vous pourrez voir le 1 transféré sur le quartet le plus significatif. A partir de 10 BCD devient un véritable hybride représentant un nombre décimal sous une forme binaire.

A mesure que les nombres entrés augmentent, la nature non-économique du BCD devient apparente. Si maintenant vous entrez une nouvelle valeur de départ de 95 et commencez à incrémenter vous pouvez voir le problème quand 99 devient 100. Quand le comptage est exécuté, de toute façon l'accumulateur est faussé et contient des valeurs incorrectes. Le programme y remédie en affichant dans la case le message "TROP GRAND POUR BCD". Si vous décrémentez jusqu'à 99 ou moins, la valeur BCD appropriée est de nouveau affichée mais le flag carry reste mis. (Si vous voulez, vous pouvez annuler les deux flags carry en appuyant sur "R". Cependant, le flag BCD sera remis aussitôt qu'une valeur supérieure à 99 est rencontrée). Quand on incrémente de 99 à 100, le BCD génère une retenue de son quartet le plus significatif sur le flag carry afin que le Z80 ne la perde pas mais mette le flag.

Comme on l'a vu plus haut, cette retenue n'est qu'un expédient à court terme et on doit la prendre en compte le plus tôt possible si on ne veut pas la perdre. La retenue est générée sur les case du BCD à 99 alors que les cases binaires seront stockées jusqu'à 256 et les décimales jusqu'à 999. Le BCD est moins économique mais il a ses utilisations dans des situations particulières.

Maintenant que vous savez tout sur le BCD, essayez l'exercice suivant:-

EXERCICE A5.4

Convertissez les nombres décimaux suivants en BCD:

```
i)    4            v)   53
ii)   10          vi)   102
iii)  77         vii)   953
iv)   97        viii)   2579
```

Les réponses sont dans le chapitre des solutions

EXERCICE A5.5

Convertissez les nombres BCD suivants en décimal:-

i)   0000 0001
ii) 0000 1001
iii) 0001 0101
iv) 0010 0000
v)   0100 1001
vi) 0010 0011
vii) 1001 0111
viii)1000 1000

Les réponses sont dans le chapitre des solutions.

Malheureusement pour vous, le programme n'accepte pas les nombres entrés en notation BCD. Cependant si vous êtes désespéré vous pouvez entrer chaque quartet séparément.

Dans l'explication donnée pour la valeur des emplacements en notation à valeur déplacement une simplification a été adoptée pour rendre ces explications plus claire pour ceux au! n'ont pas un grand penchant pour les mathématiques! Cependant, si vous désirez avoir une explication légèrement plus mathématique, lisez ce qui suit. Sinon - FIN DE L'ANNEXE 5.

Avec les nombres binaires, on a dit que les emplacements augmentaient leur valeur par puissances de 2, mais le bit le moins significatif du nombre binaire est équivalent au même symbole en base 10 (ou dans ce cas en base "S, ou autre). En fait, le facteur multiplicateur est la base, mise a la puissance de son emplacement en commençant par zéro a gauche. Par exemple en binaire:

```
    7    6     5     4     3    2    1    0   Emplacement

   128  64    32    16     8    4    2    1   Facteur multiplicateur
                                               précédemment cité

   2^7  2^6   2^5   2^4   2^3  2^2  2^1  2^0 Facteur mathématiquement
                                           plus précis
```

Le bit le moins significatif est multiplié par 2^0 soit un. (Si vous n'en "êtes pas sûr essayez la commande directe PRINT 2^0.) Le bit suivant est multiplié par 2^1, et ainsi de suite.

Cette règle s'applique pour n'importe quelle base. Appliquons-la en hexa, c'est-à-dire en base 16:

Facteur du bit le moins significatif = 16^0 =1
Facteur du 2ème bit le plus significatif = 16^1 = 16
Facteur du 3ème bit le plus significatif = 16^2 = 256
Facteur du bit le Plus significatif = 16^3 = 4096

ANNEXE 6

L'ASSEMBLEUR

Cette annexe explique toutes les fonctions de l'assembleur et de l'éditeur. Il vous est recommandé de la lire et de vous y référer quand ce sera nécessaire tout au long du livre.

Les instructions d'éditeur suivantes sont affichées par le menu.

```
1   Insérer texte       8  Sauvegarder fichier
2   Lister              9  Charger fichier
3   Remplacer ligne    10  Imprimer fichier
4   Effacer texte      11  Renuméroter
5   Assembler          12 Convertir base
6   Retour au Basic    13 opération arithmétique
7   Appeler programme
```

Ces instructions vont être détaillées dans les pages suivantes.

1. L'option insérer

Cette instruction est utilisée pour entrer un texte dans le fichier de textes. Il est possible de spécifier à la fois le numéro de la ligne de départ et l'incrément de ligne en réponse à la question "Entrez départ et incrément". Le premier nombre sera pris pour numéro de la ligne de départ, le second pour l'incrément de ligne. Ces deux nombres doivent être séparés par un tiret "-" qui se trouve en dessous du signe "=" sur le clavier. Par exemple la séquence d'instruction suivante fera donner par l'assembleur 10 comme numéro de ligne à la première ligne de texte entrée et fixera 20 comme deuxième numéro de ligne c'est-à-dire un incrément de ligne de 10. La demande de l'éditeur sera inscrite en gras.

```
> I
> Entrez départ et incrément? 10-10 <ENTER>
```

Notez qu'en fait vous ne verrez pas le "I" apparaître sur l'écran, on le montre ici pour signifier qu'il a été enfoncé, notez aussi que <ENTER> sera utilisé dans cette annexe pour indiquer qu'on appuie sur la grosse touche bleue ENTER.

Si aucun incrément de ligne n'est spécifié on prendra alors 10. Pour sortir du mode insertion on tape le caractère en premier caractère de ligne. Cela fera revenir l'éditeur en mode commande indiqué par ">". Par exemple, si la dernière ligne de votre programme était RET à la ligne 120, alors " " serait entré comme premier caractère à la ligne 130.

L'écran afficherait:

```
120 RET <ENTER>
130 @ <ENTER>

Taper M pour le menu
>
```

2. L'option lister

Cette instruction permet de lister sur l'écran tout ou partie d'un programme. Quand on la sélectionne la demande "Entrez le numéro de ligne de départ" apparaît. Pour lister tout le programme, appuyer seulement sur la touche ENTER. Une fois que le programme listé a complètement rempli l'écran, la -procédure de listage s'arrête tant qu'aucune autre touche n'est appuyée. 'Si vous avez besoin de commencer le listing à un numéro de ligne précis entrez ce numéro de ligne en réponse à la demande "Entrez le numéro de ligne". Vous pouvez arrêter le listing à n'importe quel moment en appuyant sur la barre espace; appuyer sur n'importe quelle touche permettra au listing de continuer. De même, pendant le listage, appuyer sur la touche "M" mettra fin au listing et imprimera le menu sur l'écran.

3. L'option remplacer

Cette option est utilisée pour remplacer une liane existante par une autre, Supposons que le programme suivant soit dans le fichier de texte:

```
10 LD A,65
20 IN C A
30 CALL &BB5A
40 RET
```

La ligne 20 doit être INC A, et sera changée par les instructions suivantes.

> R
> Entrez le numéro de ligne 20 <ENTER>
> 20 INC A <ENTER>
Notez qu'il n'est pas possible de remplacer plus d'une ligne à la fois. Si vous avez besoin de remplacer une série de lignes utilisez l'option insérer pour insérer le nouveau texte.

4. L'option effacer

Cette instruction permet d'effacer une seule ligne ou un bloc entier de texte du fichier de texte. La syntaxe pour cette instruction est exactement la même que pour l'instruction insérer.

Par exemple:

> D
> Entrer numéro de ligne 20 <ENTER>

Effacera la ligne 20 alors que:

> D
> Entrer numéro de ligne 20-200 <ENTER>

Effacera les lignes 20 à 200 incluse.

Pour effacer un programme entier choisissez des numéros de ligne à l'extérieur de l'intervalle utilisé. Par exemple, supposons qu'un programme utilise l'intervalle de numéros de ligne 10 à 320. La séquence d'instructions suivante efface entièrement le programme de la mémoire.

> D
> Entrer numéros de ligne 1-400 <ENTER>

5. L'option assembler

Après avoir entré un programme dans le fichier de texte il est nécessaire d'assembler ce code source en code machine. On l'accomplit avec 1'option assembler.

Quand l'option assembler a été sélectionnée, (en tapant A en mode commande), un menu apparaîtra comme ci-dessous:

1. Pas de listing
2. Listing sur écran.
3. Listing sur imprimante.
4. A la fois sur écran et imprimante

1. Pas de listing

Cette option ne fera générer aucun listing et conduira donc à l'assemblage le Plus rapide. De toutes façons toutes les erreurs seront signalées.

2. Listing sur écran.

Cette option fera sortir le listing d'assemblage de l'écran.

3. Listing sur imprimante.

C'est exactement la même que l'option 2 à la différence que la sortie est dirigée sur l'imprimante.

4. A la fois sur écran et sur imprimante.

La sortie de l'assemblage est dirigée à la fois sur l'écran et sur l'imprimante.

Si pendant l'assemblage l'assembleur rencontre une erreur quelle Qu'elle soit il sautera à l'éditeur après vous avoir signalé l'erreur.

6. L'option Retour au Basic

Cette instruction sort du programme d'assembleur et retourne au BASIC après avoir effacé l'écran avec une instruction MODE 1. L'Assembleur et l'Editeur sont écrits dans un mélange de BASIC et de code machine. Le petit programme de commande, en BASIC, utilise les numéros de ligne 64000 et au dessus. Dès lors, du moment que les numéros de lignes sont maintenus en dessous de 64000 il est possible d'avoir un petit programme BASIC résident en mémoire, en même temps que l'assembleur. C'est très utile pour tester les algorithmes avant de les essayer en code machine. La taille maximum du programme BASIC additionnel est d'à peu près 4K. Pour entrer à nouveau l'assembleur, en sauvegardant n'importe quel programme figurant dans le fichier, appuyez sur la touche point décimal sur le bloc de touches numériques ou tapez à partir du BASIC "GOTO 64026". Si vous n'avez pas besoin de garder le contenu du fichier de texte, c'est a dire si vous voulez effacer n'importe quel programme de langage assembleur en mémoire, tapez à partir du BASIC "GOTO 64018".

7. L'option appeler programme

Cette option Permet aux programmes en code machine de fonctionner ou d'être appelés à partir de l'assembleur. Pour faire fonctionner un programme en code machine, on appelle la case mémoire contenant la première instruction dans le programme. Remarque: cette option fonctionnera seulement si le programme utilise la directive d'assembleur ENT pour première ligne; n'essayez pas de l'utiliser si votre programme utilise la directive d'assembleur ORG, l'ordinateur pourrait se comporter bizarrement c'est à dire: se "planter".

Dès lors, si ENT a été utilisée, appuyer sur C en mode commande effacera l'écran et sautera ensuite au programme en code machine.

Pour exécuter un programme qui utilise la directive ORG il est nécessaire d'appeler le début du programme à partir du BASIC. Donc supposons que l'instruction ORG dans le programme se présente comme suit:

10 ORG 30000

Cela amènera l'assembleur a stocker le premier octet du code machine en case mémoire 30000. Maintenant tapez ce oui suit:

CALL 30000 <ENTER>

Cela lancera le programme en langage machine commençant en case mémoire 30000. Assurez vous cependant qu'il n'y a pas d'erreur dans le programme en code machine cela pourrait "planter" l'ordinateur.

Comme on l'a vu précédemment il est possible de sauvegarder une copie du code machine généré par l'assembleur (un fichier code obi et), ce type de fichier doit être chargé dans l'ordinateur à partir du BASIC. L'instruction de chargement est exactement la même que celle utilisée pour charger un programme BASIC a la différence qu'on doit dire à l'ordinateur où il doit localiser le fichier en code machine. Cette information, l'adresse de chargement, est donnée après le nom du fichier, séparée d'une virgule.

Par exemple pour charger en mémoire le code objet créé par le programme TEST et sauvegardé sous le nom de fichier TEST-b, à partir de la case mémoire 30000, l'instruction suivante sera entrée a partir du BASIC

LOAD "TEST",30000 <ENTER>

Ensuite pour exécuter le programme

CALL 30000 <ENTER>

sera utilisé.

Les instructions cassette (S et F)

En utilisant ces instructions, des fichiers de texte peuvent être sauvegardés sur une cassette ou chargés à partir d'une cassette sous une forme convenant à l'assembleur. Il est aussi possible de sauvegarder le code machine (code objet). produit quand un fichier est assemblé. C'est très utile car cela permet de développer et d'exécuter un programme en code machine indépendamment de l'assembleur.

8. L'option sauvegarder

Cette option est utilisée soit pour sauvegarder une copie du fichier de texte sur cassette soit pour sauvegarder une copie du code objet.

Sauvegarder le fichier de texte

Comme pour les programmes BASIC, un nom est attribué aux fichiers oui peut consister en n'Importent Quels caractères alphanumériques. Quand 1'option sauvegarder est sélectionnée on vous demande d'entrer le nom de fichier que vous désirez utiliser en réponse à la Question "Entrez le nom de fichier". Donc pour sauvegarder le contenu du fichier de texte sous le nom de fichier TEST on utilisera les instructions suivantes

> S
> Entrer le nom de fichier? TEST <ENTER>

Le message familier: "Appuyer sur REC et PLAY et ensuite sur n'importe quelle touche:" apparaîtra. A ce moment là vous devrez procéder comme en BASIC: introduire une cassette vierge, appuyer sur PLAY et REC et ensuite sur n'importe Quelle touche. Quand le fichier a été sauvegardé le message de mode commande ">" apparaîtra à nouveau.

Sauvegarder le code objet

Il est aussi possible de sauvegarder une copie du code objet généré par le dernier assemblage d'un fichier de texte. Le code objet est le code machine produit par l'assembleur. Dès lors un programme en langage assembleur peut être développé en utilisant l'assembleur et ensuite le code machine résultant peut être sauvegardé sur cassette pour l'utiliser plus tard. (Remarque: vous trouverez plus de détails sur comment exécuter ce fichier code objet dans la section détaillant 1'option "Appeler programme")

Pour sauvegarder un code objet il est nécessaire d'ajouter une marque de type de fichier au nom du fichier. Cette marque est un "b" séparé du fichier par un tiret. Avant que le code objet puisse être sauvegardé, on doit assembler le programme constitué par le fichier de texte. Dès lors pour sauvegarder le code objet créé par le fichier imaginaire TEST on utilisera le nom de fichier suivant:

> s
> Entrer le nom du fichier? TEST-b <ENTER>

9. L'option charger

Cette option est utilisée pour charger un fichier de texte a partir de la cassette. Remarque: il ne peut être utilisé pour recharger une copie du code objet créée par l'assembleur et sauvegardée avec l'identifieur "-b". Voir l'option "Appeler programme" pour cela. Vous êtes guidé par le message "Entrer nom de fichier". Si vous ne pouvez pas vous souvenir sous quel nom de fichier le programme a été sauvegardé appuyez seulement sur ENTER en réponse au message et le premier fichier compatible trouvé sur la cassette sera chargé. Par exemple on utilisera ce oui suit pour charger le fichier test

> F
> Entrer le nom du fichier? TEST <ENTER>

Appuyer sur PLAY puis n'importe Quelle touche: <Appuyer sur PLAY puis ENTER>

10. L'option imprimer

Cette option permet au programme figurant dans le fichier de texte d'être sorti sur l'imprimante. La syntaxe pour cette instruction est exactement la même que pour l'instruction lister, a la différence qu'on entre P a la place de L.

11. L'option renuméroter

Cette instruction renumérote toutes les lignes stockées dans le fichier de texte. Quand elle est sélectionnée, le message "Entrer le nouvel incrément de ligne" apparaît; le numéro entré a ce moment là sera utilisé comme numéro de ligne de départ et comme incrément de ligne. Donc pour renuméroter un fichier de texte avec un incrément de ligne de 10 en commençant par le numéro de ligne 10, on utilisera la séquence d'instruction suivante:

> N
> Entrer le nouvel incrément de ligne 10 <ENTER>

12. L'option convertir base

Quand on écrit des programmes en langage assembleur il est parfois utile de pouvoir convertir un nombre d'une base à une autre. L'Option Convertir Base permet cela, elle acceptera n'importe quel nombre dans l'intervalle 0-65535. Le nombre à convertir peut être représenté dans l'une des trois bases suivantes:

1. Décimale.
2. Hexadécimale.
3. Binaire.

Pour les distinguer, on utilise les préfixes BASIC standards:

Base	Préfixe
Décimale	Pas de préfixe
Hexadécimale	&
Binaire	&X

Le nombre entré est converti dans toutes ces bases. Donc pour exprimer 275 en décimal, hexa et binaire, on utilisera la séquence d'instruction suivante:

> B

Entrer nombre à convertir? 275 <ENTER>

L'écran affichera alors le nombre 275 dans les trois bases. Pour effacer l'écran, appuyez encore sur B. Notez que la partie inférieure de l'écran ne sera pas affectée par toutes les instructions, en dehors de B et 0 qui l'effaceront.

13. l'option opération arithmétique

Cette option permet à deux nombres, dans n'importe laquelle des trois bases, d'être soit ajoutés ou soustraits l'un à l'autre. Par exemple ajoutez 200 à &FF.

> 0

```
Premier nombre? 200  <ENTER>
Deuxième nombre? &FF <ENTER>
A. Add S. Soustr.? A <ENTER>
```

La partie inférieure de l'écran s'efface maintenant et affiche ce oui suit:

```
Binaire = &X11100011
Décimal = 445
Hexadécimal = &1C7
```

Comme pour l'option Convertir Base, cet affichage restera Jusqu'à ce qu'on appuie à nouveau sur 0 ou B.

Directives d'Assembleur

En plus de toutes les instructions du Z80, l'assembleur acceptera certaines directives d'assembleur. Une directive d'assembleur est une instruction à l'assembleur au lieu d'être une instruction au Z80 (pour cette raison on les appelle quelques fois pseudo-instructions).

Les directives d'assembleur augmentent considérablement la lisibilité et la facilité d'écriture des programmes.

La première directive devrait être familière:

```
       ENT Place le code objet immédiatement après le code source, dans une
           forme permettant l'exécution du programme par l'option Appeler
           Programme de l'assembleur.
```

Quand on écrit des programmes qui nécessitent un stockage de données supérieur à ce qui est possible à partir des registres ou facilement à partir du stack, il est nécessaire d'utiliser des cases mémoire pour stocker ces données. Comme on ne connaît pas la longueur exacte du programme il n'est pas facile de calculer l'adresse de la case mémoire située a la fin du programme. La solution du problème est d'utiliser la directive d'assembleur suivante:

```
DEFS n    Réserve n octets de mémoire en commençant par la valeur actuelle du
          compteur d'adresse.
```

Pour permettre une référence facile à cette zone de mémoire on affecte un label (= étiquette) a cette directive. Par exemple pour envoyer un buffer de quatre octets référencé par le label STORE: on utilisera la ligne suivante

```
STORE: DEFS 4
```

Quand l'assembleur rencontrera cette instruction il réservera quatre octets de mémoire et affectera au label "STORE:" l'adresse du premier octet de ce buffer.

Par exemple:

Supposons qu'un programme de multiplication de 8 bits par 8 produise un résultat de 16 bits en HL; si on a besoin de ce résultat plus loin dans le programme il devra être conservé.

En utilisant le programme suivant on peut obtenir ce résultat en définissant le label STORE:

```
       ENT
STORE: DEFS 2
     .
     .
     .

Programme de multiplication
     .
    .
LD (STORE),HL
    .
    .
```

Dès lors le contenu de la case mémoire adressée par STORE contient le résultat de la multiplication. En plus de la création de buffer, les directives d'assembleur peuvent être utilisées pour beaucoup d'autres fonctions. La première à être considérée est l'attribution de valeurs numériques aux labels ou étiquettes.

```
EQU nn Affecte la valeur nn au label précédent.
```

Cette forme de directive est principalement utilisée pour définir des constantes.

Par exemple:

```
LOOP: EQU 20
      LD A,LOOP:
```

Une fois assemblé et exécuté, ceci chargera 20 dans A.

Jusqu'ici la plupart des programmes ont utilisé la directive ENT. C'est parfait pour développer et tester des fonctions mais quand un programme en code machine doit être appelé à partir du BASIC il doit être stocké là où le BASIC ne peut pas le recouvrir. La quantité de mémoire disponible pour le BASIC est fixée par l'instruction BASIC MEMORY. En mettant MEMORY sur 39999 aucune case mémoire au dessus de 39999 ne peut être utilisée pour le stockage d'un programme BASIC: en effet, la mémoire a été réservée pour les programmes en code machine.

Au lieu de laisser l'assembleur décider où loger le programme, il faut qu'on le lui dise. On utilise la directive suivante pour accomplir ceci:

```
ORG nn  Met le compteur d'adresse sur nn
```

Cette propriété de l'assembleur permet aux programmes d'être assemblés pour résider n'importe où en case mémoire. Remarque: la valeur nn doit "être supérieure à la dernière case mémoire utilisée par l'éditeur. Si vous tapez une case erronée l'assembleur vous le signalera.

Beaucoup de programmes ont besoin de cases mémoires spécifiques pour y stocker des données. Il y a trois directives pour accomplir ceci; les deux premières chargent des données numériques et se présentent comme suit:

```
DEFB n   Stocke n à la valeur actuelle du compteur d'adresse.
```

```
DEFW nn Stocke l'octet faible du mot de deux octets nn à la valeur actuelle du compteur d'adresse et l'octet fort à la valeur actuelle du compteur d'adresse + 1.
```

La dernière directive stocke la représentation ASCII d'une chaîne en mémoire. C'est très utile pour imprimer des messages sur l'écran.

```
DFFM "s" Définit le contenu de la mémoire, en commençant par la valeur actuelle du compteur d'adresse/ comme la représentation en ASCII de la chaîne "s".
```

Essayez le programme suivant;

```
      ENT
MESS: DEF "A MESSAGE !"
      LD B,11
```

```
      LD HLMESS:
LOOP: LD A,(HL)
      CALL &BB5A
      INC HL
      DJNZ LOOP:
      RET
```

Utile non?!

Addition d'opérandes 8 chargement direct de caractères ASCII

Si jusqu'à maintenant vous avez travaillé sur le livre et si vous avez lu le chapitre 10, vous avez peut être été frappé que dans le chapitre 10 880 était, ajouté à un caractère ASCII entre guillemets dans une instruction DEFB. L'effet de cette ligne est de charger en mémoire, à la valeur actuelle du compteur d'adresse, la valeur ASCII du caractère entre guillemets une fois qu'on lui a ajouté 880. L'effet d'ajouter 880 à un nombre binaire est de mettre le bit 7.

Dès lors pour charger une représentation ASCII de A dans l'accumulateur il est possible d'utiliser la ligne suivante:

```
LD A,"A"
```
qui chargera 65 dans A. De même, pour charger "B" dans A vous pouvez utiliser:

```
LD A,"A"+1
```

Ces fonctions sont parfois très utiles, surtout pour définir des tableaux de données.

Commentaires

L'assembleur permettra d'inclure des lignes de commentaires dans le fichier de texte si le premier caractère de cette ligne est un point-virgule. Un programme d'exemple contenant des commentaires apparaîtra donc comme ci-dessous:

```
10     ENT
20 ; Charger 65 dans A
30      LD A,65
40 ; Imprimer A sur l'écran
50      CALL &BB5A
60      RET
```

Notez que les commentaires doivent être sur des lignes séparées des instructions; on ne peut pas combiner les deux sur une ligne.

GLOSSAIRE

ADRESSE

L'ordinateur contient un grand nombre de chips, dans lesquels il enregistre les données. Il stocke les données par petits groupes de 8 bits, appelés octets. L'ordinateur a un grand nombre de ces octets dans sa mémoire, on appelle chaque emplacement oui contient un octet une case mémoire. Chaque case mémoire a un numéro spécial oui lui est associé, exactement comme un numéro de maison, ce numéro est appelé l'adresse de cette case mémoire. Elle aide l'ordinateur a trouver l'octet qu'il veut.

ADRESSE DE VECTEUR

Dans une opération d'indirection, l'adresse qu'on doit rechercher est appelée l'adresse de vecteur.

ALGORITHME

Un algorithme est un groupe d'instructions qui exécute un travail particulier. Les recettes de cuisine sont des algorithmes comme le sont les organigrammes et les programmes d'ordinateur.
ALU (ARITHMETIC AND LOGIC UNIT)
Unité arithmétique et logique. C'est la zone dans le processeur central qui effectue toutes les opérations arithmétiques et logiques,

ANNULER

Un flag ou un bit est annulé quand il a la valeur "0". Un pixel est annulé quand il est éteint.

ARCHITECTURE

L'arrangement ou le dessin de la logique dans un système informatique.

ASCII

Une abréviation de American Standard Code for Information Interchange, Dans l'ordinateur, les informations sont stockées sous la forme de nombres binaires/ il a donc été décidé/ cour représenter une lettre dans l'ordinateur/ de donner à chaque lettre de l'alphabet un numéro spécial ou code. Ce code est appelé le code ASCII de la lettre. Les codes ASCII ne représentent pas tous des lettres/ certains représentent des caractères tels que =/+/!/? ou /.II y a aussi des codes qui représentent des nouvelles lignes etc. Beaucoup d'ordinateurs ne se conforment pas strictement au standard ASCII mais la plupart s'y conforment dans une assez large mesure.

ASSEMBLEUR

Un programme oui traduit le langage assemblage en code machine.
BASIC
Beginners Ail Purpose Symbolic Instruction CODE. Un langage de programmation "évolué".

BIT

Un chiffre/ soit 0 soit 1/ dans un nombre binaire,

BUS DE DONNEES

Un groupe de voies d'accès utilisé par le système informatique pour communiquer intérieurement.

CARACTERES DE CONTROLE

Les caractères de contrôle n'affichent rien mais servent à exécuter des fonctions telles que "nouvelle ligne" <RETURN> ou effaçage de l'écran.

CARTE MEMOIRE

Un diagramme montrant comment l'ordinateur utilise la mémoire, par exemple quelle case mémoire il utilise et pourquoi.

CASE MEMOIRE

L'ordinateur a un grand nombre de chips dans lesquels il enregistre les données. Il stocke les données par petits groupes de huit bits, appelés octets. L'ordinateur contient un grand nombre de ces octets dans sa mémoire, chaque emplacement qui contient un octet s'appelle une case mémoire. Chaque case mémoire a un numéro spécial qui lui est associé, exactement comme un numéro de maison, ce numéro s'appelle l'adresse de la case mémoire. Il aide l'ordinateur è trouver l'octet qu'il désire.

CHARGER

Charger un registre ou l'accumulateur signifie qu'on y place un nombre. Ce nombre peut être obtenu de différentes façons.

CODE MACHINE

Ce sont, en fait, les nombres que le chip du microprocesseur Z80 utilise. Pas seulement des données, mais toute instruction que le chip exécute, est représentée par un nombre en code machine, et ce type de nombres qui contrôle le voltage interne, est tout ce que le chip peut utiliser.

CODE OBJET

La version code machine d'un programme en langage assembleur. On dit que l'assembleur convertit le "code source" en "code objet".

CODE D'OPERATION

Une instruction, généralement écrite en langage assembleur ou dans un langage évolué, que l'ordinateur peut suivre (ADD, LD, PRINT, etc.). Notez que le code d'opération, c'est l'instruction moins ses opérandes.
Donc ADD A,B est une instruction, alors que ADD est le code objet.

CODE SOURCE

Le langage assembleur tel qu'on le tape et avant que l'assembleur le convertisse en code objet ou "code machine".

COMPILATEUR

Un programme qui convertit un programme entier écrit en langage évolué comme le BASIC en code machine avant que le programme soit exécuté. On peut obtenir des vitesses comparables à celles des programmes écrits directement en assembleur ou en code machine, Voir aussi "assembleur", "désassembleur" et "interpréteur".

COMPILER

Créer des programmes objet a partir du source.

COMPLEMENT

Une forme "inverse" d'un nombre binaire sous laquelle tous les 1 deviennent des 0 et tous les 0 des 1.

COMPTEUR DE PROGRAMME

C'est un registre spécial de 16 bits, dans le Z80, qui désigne l'instruction suivante à exécuter pour que le Z80 ne perde pas le fil de l'exécution d'un programme!

DESASSEMBLEUR

Un programme oui traduit le code machine en langage assembleur,

DIRECTIVE D'ASSEMBLEUR

Ces instructions disent quelque chose (ORG par exemple) à l'assembleur qui commence alors à stocker le code objet à cette adresse. Voir aussi assembleur.

DOUBLE PRECISION

Un terme assez général oui signifie deux fois plus précis. Dans ce livre, signifie que des opérations mathématiques sont faites sur des nombres longs de deux octets (entre 0 et 65535) plutôt Que sur des nombres longs d'un octet.

ECRAN (ADRESSES)

Une zone spéciale de la mémoire de l'ordinateur est réservée à l'écran. Cette zone de mémoire indique à l'ordinateur où vont les informations sur l'écran. Si on altère le contenu de ces cases, le caractère affiché sur 1'écran changera.

ETIQUETTE Voir LABEL.

EXECUTER

Dans ce livre, signifie lancer un programme en code machine.

FLAG

Un emplacement spécial, généralement d'un seul bit, qui est mis sur une valeur particulière si une condition est remplie et sur une autre valeur si elle ne l'est pas. Il peut changer suivant le résultat, par exemple, d'une opération de comparaison entre deux nombres: s'ils sont égaux il sera mis sur 1 et s'il ne le sont pas il sera mis sur 0.

IMBRICATION

Une structure d'un programme d'ordinateur dans laquelle une boucle est mise dans une autre.

INDIRECTION

Procédure de recherche de l'adresse à utiliser dans une instruction quand elle n'est en fait pas donnée dans l'instruction: l'instruction contient l'adresse à laquelle l'adresse à utiliser est stockée.

INSTRUCTIONS

Les équivalents Anglais les plus proches du code machine. On écrit en fait des instructions quand on entre le langage assembleur. ADD A,B est donc une instruction; 80 est l'équivalent en code machine. Voir aussi "mnémoniques".

INTERFACE

Un périphérique qui permet à deux systèmes électroniques, normalement incompatibles directement, de communiquer ensemble.

INTERPRETEUR

Un programme qui lit un autre programme (un programme BASIC par exemple) et le convertit instruction par instruction en un code machine qui peut "être alors exécuté directement. Notez qu'il est différent d'un assembleur ou d'un compilateur qui convertissent des programmes entiers ("le code source"). En langage interprété comme en BASIC, les programmes fonctionnent plus lentement que les programmes en langage assembleur et les programmes compilés car chaque instruction BASIC doit être interprétée chaque fois qu'elle est exécutée, alors qu'un programme en langage assembleur, par exemple, est seulement converti une fois en code machine avant que le programme soit lancé. Voir aussi "compilateur".

I/O

Abréviation de l'Anglais Input/Output (Entrée/Sortie).

LABEL SYMBOLIQUE (ou ETIQUETTE)

C'est le nom donné à une ligne de programme pour que le programmeur puisse s'y référer facilement. Au lieu de sauter à une case mémoire spécifique dans le programme, on peut sauter au nom de cette case mémoire.

LANGAGE ELEMENTAIRE ET LANGAGE EVOLUE

Le niveau d'évolution d'un langage d'ordinateur se réfère approximativement à sa similarité avec l'anglais ou tout autre langage parlé. Plus il est évolué, plus le langage se rapproche d'un véritable langage parlé. Moins le langage est évolué Plus, il est proche des nombres obscurs du code machine, Les langages évolués sont caractérisés par des instructions simples qui peuvent faire beaucoup - c'est-à-dire des Instructions puissantes. Voici une sélection en ordre approximatif, d'élémentaire à évolué, de langages d'ordinateur: code machine, langage assembleur, FORTRAN, BASIC, ALGOL, PASCAL, COBOL, LISP. Certains langage comme le FORTH, qui est plutôt évolué, ont beaucoup de propriétés à la fois de langage élémentaire et de langage évolué et il est difficile de dire s'ils sont élémentaires ou évolués.

METTRE

Généralement utilisé pour les flags, cela signifie mettre un 1 dans un flag. Un pixel est mis quand il est allumé.

MNEMONIQUES

Tirant leur nom de celui de la déesse grecque de la mémoire, Mnémosyne, les mnémoniques sont simplement des aides-mémoire. On se souvient plus facilement de ce que "ADD A,B" fait que de ce que fait le chip du Z80 avec l'instruction "80". On tendra donc à utiliser le langage assembleur avec ses mnémoniques plutôt que de rentrer les programmes directement en code machine.

OCTET

Un groupe de 8 bits utilisé pour représenter un nombre compris entre 0 et 255.

OPERANDE

L'opérande d'une instruction est tout nombre oui suit cette instruction, comme A et 145 dans LD A,145, Certaines instruction, telles que RET, n'ont pas besoin d'opérande.

OPERATEUR LOGIQUE

Un opérateur ou une fonction qui ne fonctionne qu'avec les valeurs VRAI et FAUX. Les principaux opérateurs logiques Basic sont AND, OR, et MOT. MOT renvoie simplement le complément (l'opposé) de son entrée.

OSCILLATEUR A CRISTAUX

Procédé basé sur la vibration d'un cristal de quartz pour produire un voltage électrique oscillant. Utilisé dans les montres électroniques, montres à quartz, et aussi dans les appareils de temporisation (horloges) de certains systèmes informatiques.

PARAMETRE

La plupart des instructions ont des paramètres. Par exemple dans la ligne BASIC"PRINT a,b"; a et b sont des paramètres.

PERIPHERIQUE

Un appareil lié au système informatique principal, tel qu'une imprimante ou un écran de visualisation.

PIXEL

Contraction de "picture élément" (élément d'Image), un pixel est la plus petite partie de l'écran que l'ordinateur puisse contrôler, c'est-à-dire un point. Toutes les Images, lettres etc. que l'ordinateur envoie sur l'écran sont créées à partir d'une combinaison de Pixels.

"PLANTER"

Si on dit au Z80 d'exécuter un programme en mémoire oui n'est pas tout & fait complet, il peut alors se mettre à exécuter ce qu'il ne devrait pas exécuter!Le Z80 n'est pas très malin et il peut très bien essayer d'exécuter le contenu d'un buffer, d'une chaîne de variables ou de n'importe quelle autre sorte de donnée. Quand cela se produit le Z80 s'embrouille complètement, sautant un peu de partout, exécutant un petit bout de ci-delà. Il peut stocker des choses de partout, mettant généralement un sacré désordre dans ce que vous avez en mémoire. Vous devrez alors éteindre l'ordinateur pour ramener l'ordinateur à la raison, ("planter" l'ordinateur n'abîmera pas le système. Plantez le système pour vous amuser si vous voulez!)

PORTE

Généralement, un périphérique électronique pour exécuter des opérations logiques. Les niveaux de voltage sont utilisés pour indiquer les valeurs VRAI et FAUX, ou 1 et 0. (Il existe aussi des portes logiques pneumatiques, mais pas dans cet ordinateur!)

PROGRAMME OBJET

Le programme objet est en fait la version code machine d'un programme en langage assembleur (le programme source),

PROGRAMME SOURCE

Le programme source est composé des actuelles mnémoniques d'assemblage, telles qu'elles sont utilisés dans ce livre. C'est-à-dire un programme oui a été créé par l'utilisateur mais oui doit d'abord être traduit sous une autre forme avant de pouvoir être exécuté. Un programme en langage assembleur doit être traduit en code machine afin d'être exécuté.

QUARTET

Un quartet est un groupe de 4 bits exactement comme un octet est un groupe de 8 bits. Par commodité on considère quelque fois qu'un octet est composé de deux quartets, le quartet le moins significatif (celui de droite) le quartet le Plus significatif (celui de gauche).

RAM

Abréviation de l'anglais <R>ead <A>nd <M>odify memory (mémoire pouvant "être lue et modifiée) et aussi quelques fois de <R>andom <A>ccess <M>emory (Mémoire à accès sélectif).

REGISTRE

Case mémoire spéciale dans un chip, le Plus souvent dans le processeur central, pour stocker un nombre en binaire. Les registres dans le Z80 ont soit 8 bits soit 16 bits de long.

ROM

Abréviation anglaise de <R>ead <0>nly <M>emory (mémoire de lecture uniquement). C'est la mémoire dans laquelle un programme est stocké en permanence, et elle ne peut être effacée ni en essayant d'y POKEr des données ni en utilisant des instructions du type LD.

ROUTINE DE SERVICE (DES INTERRUPTIONS)

Routine en code machine utilisée pour gérer les interruptions.

SYSTEME D'EXPLOITATION

Un programme intégré conçu pour simplifier les procédures d'intendance à l'intérieur de l'ordinateur. Il s'occupe de fonctions telles que l'exploration du clavier, la création de l'affichage vidéo, la sauvegarde des programmes sur cassette, etc.

UNITE CENTRALE

La partie (ici le Z80) de l'ordinateur qui exécute tous les programmes en code machine. (Tout programme peut être converti en code machine sous une forme ou sous une autre!).

Z80

Z80 est le numéro de référence de l'unité centrale utilisée par cet ordinateur. On se réfère souvent à cette unité centrale en l'appelant "Z80".

S O L U T I O N S

CHAPITRE 1

EXERCICE 1.1

```
ENT
LD A,65
CALL 47962
RET
```

EXERCICE 1.2

```
ENT
LD A,70
CALL 47962
LD A,82
CALL 47962
LD A,69
CALL 47962
LD A,68
CALL 47962
```

CHAPITRE 2

EXERCICE 2.1

```
ORG 30000
LD B,10
LD A,65
CALL 47962
DEC B
JR NZ,30004
RET
```

EXERCICE 2.2

On utilise de préférence l'instruction JR NZ à l'instruction JP NZ car la distance de saut est entre +129 et -126 octets, et l'instruction JR NZ nécessite un temps d'exécution plus court. Elle est des lors plus efficace.

EXERCICE 2.3

```
      ORG 30000
      LD C,26
      LD A,65
NXT:  CALL 47962
      INC A
      DEC C
      JR NZ,NXT:
      RET
```

CHAPITRE 3

EXERCICE 3.1

Case mémoire	Contenu
:	:
:	:
200	E
201	D
202	L
203	H
:	:
:	:

EXERCICE 3.2

```
ENT
LD DE,100
LD (35000),DE
LD HL,400
LD (35002),HL
LD DE,(35000)
LD HL,(35002)
CALL 48118
RET
```

Si vous utilisez la sous-routine plot en code machine en 48106, le programme devient:

```
ENT
LD DE,100
LD (35000),DE
LD HL,400
LD (35002),HL
LD DE,0
LD HL,0
CALL 48106
```

```
LD DE,(35000)
LD HL,(35002)
CALL 48118
RET
```

EXERCICE 3.3

```
ENT
LD DE,200
LD HL,300
CALL 48118
LD DE,400
LD HL,200
CALL 48118
LD DE,0
LD HL,0
CALL 48118
RET
```

Remarque: Il n'est pas vraiment nécessaire, a part pour une illustration, de charger les coordonnées en mémoire. Ce programme est plus rapide qu'un programme qui stocke les coordonnées en mémoire et oui les lit ensuite encore une fois.

FXERCICE 3.4

```
      ENT
      LD BC,35000
      LD A, 65
      LD E,3
NXT:     LD (BC),A
      INC BC
      DEC E
      JR NZ,NXT:
      LD E,3            Nombre de parcours de la boucle
      LD BC,35000       Charger emplacement de départ de "A"
PUT:     LD A, (BC)          Mettre "A" dans l'accumulateur
      INC BC            Ajouter 1 a la case de données
      CALL 47962        Mettre "A" sur l'écran
      DEC E             Décrémenter le numéro de boucle
      JR NZ,PUT:        Si numéro de boucle <> 0, boucler encore
      RET               FIN
```

EXERCICE 3.5

```
      ENT
      LD BC, 35000
      LD A,65
      LD E,26           26 parcours de boucle sont requis
NXT:     LD(BC),A
      INC BC
      INC A             Lettre ASCII suivante
      DEC E
      JR NZ,NXT:
      LD BC,35000
      LD E,26           26 parcours de boucle sont requis
PUT:     LD A,(BC)
      INC BC
      CALL 47962
      DEC E
      JR NZ,PUT:
      RET
```

EXERCICE 3.6

```
ENT
LD IX,35000
LD A,83
LD (IX+0),A
LD (IX+2),A
INC A
INC A
LD (IX+1),A
LD A,65
LD (IX+3),A
LD A,78
LD (IX+4),A
LD A,(IX+2)
CALL 47692
LD A,(IX+3)
CALL 47692
RET
```

CHAPITRE 4

EXERCICE 4.1

```
ENT
LD A,200
ADD A,48
CALL 47692
RET
```

Ceci affiche un petit bonhomme sur l'écran.

EXERCICE 4.2

```
ENT
LD A,&41
ADD A,&10
CALL &BB5A
RET
```

Ceci place un Q sur l'écran.

EXERCICE 4.3

```
ENT
LD C,&FA    octet faible de 250
LD A,&58          octet faible de 600
AND A
ADD A,C
ADD A,65
LD (&7000),A
LD C,&00    octet fort de 250
LD A,&2     octet fort de 600
ADC A,C
ADD A,65
LD (&7001),A
LD A,(&7000)
CALL &BB5A
LD A, (&7001)
CALL &BB5A
RET
```

Ceci produit une forme comme le bras d'un robot.

EXERCICE 4.4

```
ENT
LD C,9
LD A,233
SUB C
CALL &BB5A
RET
```

Il n'est pas nécessaire de soustraire 65 de la réponse parce que CHR$(224) est un caractère imprimable (une figure souriante).

EXERCICE 4.5

```
ENT
LD A,126
ADD A,97
LD C,153
SUB C
CALL &BB5A
RET
```

Cela met "F" sur l'écran.

EXERCICE 4.6

```
ENT
LD DE,4008
LD HL,4248
AND A
SBC HL,DE
LD A,L
CALL &BB5A
RET
```

Cela met la flèche haut sur l'écran.

EXERCICE 4.7

```
ENT
LD HL,35000
LD A,10
LD (HL),A
INC HL
LD A,20
LD (HL),A
LD A,65
ADD A,(HL)
LD (HL),A
DEC HL
LD A,65
ADD A,(HL)
LD (HL),A
CALL &BB5A
INC HL
LD A,(HL)
CALL &BB5A
RET
```

Ceci affiche "KU" sur l'écran.

EXERCICE 4.8

```
ENT
LD DE,100
LD HL,50
CALL 48118
LD D,0
LD E,100
LD A,75
ADD A,E
LD E,A
LD H,0
LD L,50
LD A,75
ADD A,L
LD L,A
CALL 48118
RET
```

CHAPITRE 5

EXERCICE 5.1

```
LD A,&7
ADD A,&C
DAA
ADD A,65
CALL &BB5A
RET
```

Ceci mettra un "Z" sur l'écran.

EXERCICE 5.2

```
LD C,&12
LD A,&35
SUB C
DAA
ADD A,65
CALL &BB5A
RET
```

Ceci mettra un "d" sur l'écran.

EXERCICE 5.3

1. C=0
2. C=1
3. C=0

EXERCICE 5.4

Le masque requis sera 00000011.

EXERCICE 5.5

253 AND 75 = 73 (1001001)

EXERCICE 5.6

1. 1001 OR 1101 = 1101
2. 250 OR 25 = 251
3. (209 OR 20) AND 27 = 17

EXERCICE 5.7

1. 1101 XOR 1110100 = 127

```
LD C,11
LD A,116
XOR C
CALL &BB5A
RET
```

Ceci affiche un échiquier sur l'écran.

2. 77 XOR 200 = 133

```
LD C,77
LD A,200
XOR C
CALL &BB5A
RET
```

Ceci affiche une barre verticale sur l'écran,

3. (25 OR 255) AND 200 = 200

```
LD C,25
LD A,200
OR C
LD C,200
AND C
CALL &BB5A
RET
```

Ceci affiche deux diagonales sur l'écran,

EXERCICE 5.8

1. 0100
2. 0100010
3. 0001

EXERCICE 5.9

```
1.  1010                = 10
    1101                = -3
    0111                =  7


2. 1111
   0111
   0110

3. 0110 = -10
   1000 =8 1110 = -2
```

EXERCICE 5.10

```
LD A, 20
CPL
INC A
ADD A,98
CALL &BB5A
```

```
RET
```

Ceci affiche un "N" sur l'écran.

CHAPITRE 6

EXERCICE 6.1

La solution est la même que pour le programme 6.1 sauf pour ces lignes:

```
LD C,10
LD E,9
```

Le programme devra afficher le caractère 155 sur l'écran, c'est-à-dire T renversé avec une barre courte.

EXERCICE 6.2

```
      LD C,124
      LD E,146
      LD D,0
      LD B,8
      LD HL,O
NXTB:    SRI C
      JR NC,NOADD:
      ADD HL,DE
NOADD:SLA E
      RL D
      DEC B
      JR NZ,NXTB:
      LD A,H
      CALL &BB5A
      LD A,H
      CALL &BB5A
      RET
```

Cela donne <pi>F.

EXERCICE 6.3

Pour réaliser facilement ceci, effacer DEC B et remplacer la ligne JR NZ,NXTB: par:

```
DJNZ NXTB:
```

EXERCICE 6.4

```
LD A,5
RLCA
RLCA
RLCA
RLCA
RLCA
CALL &BB5A
RET
```

Ceci affiche la flèche haut en haut de l'écran; le caractère ASCII pour 160.

```
2.LD A, 254
  RRCA
  CALL &BB5A
  RET
```

Ceci affiche CHR$(127); un échiquier sur l'écran.

EXERCICE 6.5

```
LD A,255
CALL &BB5A
RES 4,A
CALL &BB5A
SET 1,A
RES 3,A
CALL &BB5A
RET
```

Ceci affiche une flèche a deux têtes, une fusée et une pyramide à son côté, sur l'écran.

CHAPITRE 7

EXERCICE 7.1

```
ENT
LD DE,0
LD HL,0     Met curseur graphique sur(0,0)
CALL &BBC0
LD DE,100         Charge 100 dans DE
LD HL,200         Charge 200 dans HL
PUSH DE     Sauvegarde DE sur la pile
PUSH HL     Sauvegarde HL sur la pile
CALL &BBF6        Dessine ligne jusqu'à (100,200)
LD DE,0
LD HL,0     Met curseur sur (0,0)
CALL &BBC0
POP DE      Met ancien contenu de HL dans DE
POP HL      Met ancien contenu de DE dans HL
CALL &BBF6        Dessine ligne jusqu'à (100,200)
RET
```

EXERCICE 7.2

```
ENT
LD A,43     Met "+" dans A
PUSH AF     Stocke A sur la Pile actuelle
CALL &BB5A        Met "+" sur écran
LD (&7148),SP     Enregistre valeur actuelle de SP
```

```
LD HL,&7148       Se prépare pour:
LD SP,HL    Fixer a nouveau la pile en 30000
LD A,61     Met "=" dans A
PUSH AF     Stocke A dans nouvelle pile
CALL &BB5A        Met "=" sur écran
LD HL,(&714A)     Trouve emplacement originel de SP
LD SP,HL          Revient sur la pile originelle
POP AF      Retrouve "+"
PUSH AF
CALL &BB5A
LD HL,&7146       Parce que A et F sont dans la nouvelle pile
LD SP,HL
POP AF
CALL &BB5A
LD HL,(&714A)
LD SP,HL
POP AF
CALL &BB5A
RET
```

CHAPITRE 8

EXERCICE 8.1

```
       LD HL,&B100
       LD DE,&C000
       LD BC,&3FFF
LOOP:  LDI
       JP PO,FINISH:
       JP LOOP:
FINISH:RET
```

ANNEXE 5

EXERCICE A5.1

i)   3
ii)   4
iii) 128
iv) 131
v)   183
vi) 115

EXERCICE A5.2

i)   31
ii)   41
iii) 189
iv)   136

EXERCICE A5.3

i)   9
ii)   19
iii) 165
iv)   174
v)   14
vi)   26
vii) 234

EXERCICE A5.4

i) 0000 0100   v) 0101 0011
ii) 0001 0000   vi) Trop grand pour représentation
                      BCD sur deux octets
iii) 0111 0111   vii) Trop grand pour représentation
                      BCD sur deux octets
iv) 1001 0111   viii) Trop grand pour représentation
                      BCD sur deux octets

EXERCICE A5.5

i)   1   v) 49
ii)   9   vi) 23
iii)   15   vii) 97
iv)   20   viii) 88

INDEX

M

N

O

P

Q

R

S

T

Table de vérité
Triangle (instruction)

X

XOR (porte logique)
XOR s

Z

Zéro (flag)
Z80